Scatternet Formation in Bluetooth

K. Balaji, Sanjiv Kapoor[1]*, Amit A. Nanavati, Lakshmi Ramachandran*

IBM India Research Laboratory, Block 1, IIT, Hauz Khas, New Delhi 110 016.

***Abstract*—Bluetooth is an emerging technology for pervasive computing, which is expected to become a ubiquitous solution for providing short range, low power, low cost, pico-cellular wireless connectivity. Bluetooth employs a Master-Slave model, where a Master device communicates directly with several Slave devices; devices are organized into piconets, which are star-shaped clusters with a Master at the centre of the star, connected to a limited number of Slave devices. Efficient clustering or topology construction algorithms play a very important role for forming a connected set of piconets, namely, a scatternet. In this paper, we address the problem of organization of a set of devices into a minimum set of star-shaped clusters of limited size, with the connection between stars through non-centre nodes, under the assumption that not all nodes are within radio range of each other. We show that the feasibility and optimisation of scatternet formation are algorithmically hard. For this problem, a greedy centralised algorithm, which assumes that a central entity has complete knowledge of the underlying topology is proposed. This is shown to be 1+ [k_1 min(s,n)/n]-approximation algorithm, which is in terms of size of a s-clique-cover, s, the maximum bound on the number of Slaves that any Master can have, k_1, and the number of nodes in the network, n. We then present a distributed heuristic which assumes that each node has 2-hop information of its neighbourhood. The performance of these algorithms are also analysed using simulation experiments.**

Keywords - **Bluetooth, Scatternet, Piconet, Clustering, Ad-hoc networks**

A. Introduction

Ad-hoc networks which are studied in [3,6,7,8,9,14], are expected to play a significant role in future mobile computing applications. Bluetooth [12, 13] is an emerging technology for pervasive computing that allows for the replacement of the numerous proprietary cables that connect one device to another with short range radio links. This technology has applications in short-range wireless connectivity between pervasive devices, like PDAs, mobile phones, palmtops, laptops, pagers, etc. It is meant to be a cable-replacement solution for desktops, keyboards and other peripheral devices. The other potential applications include smart home appliances for establishing wireless connectivity to backbone data networks. In all of these applications, it is important for the devices to quickly self-organize themselves to communicate with other devices in the ad-hoc network. Our work is intended to provide solution to this problem.

Bluetooth technology allows for the formation of network units called piconets and a connected set of piconets, called scatternets without the involvement of a central infrastructure, in an ad-hoc fashion. The centre of piconet is called the Master node, which is directly connected to a bounded number of Slave nodes. Connections between neighbouring piconets are through Bridge nodes, which could be of two types: Master-Slave Bridge (a Master in one piconet and a Slave in the neighbouring piconet), and Slave-Slave Bridge (a Slave node common to more than one piconet). However, the technology allows a node to belong to only one piconet at any point of time. This means, while a Master-Slave Bridge acts as a Slave in one piconet, all nodes in the piconet for which it is a Master will have to be idle. Clearly, Slave-Slave Bridges are more desirable for good performance. Moreover, the number of piconets to which a Slave-Slave Bridge is common, should be bounded for similar reasons. In this paper, we use the term to generally mean a Slave-Slave Bridge, unless stated otherwise.

---

[1] The author is with the CSE Dept. at IIT, Delhi.

The formation of clusters, and the related leader election problem have been investigated in several papers [1,2,4,5,11]. These algorithms use a fair amount of messaging, which would prove to be expensive in Bluetooth. The problem addressed in this paper can be seen as a variation of the Minimum Connected Dominating Set problem, which has been investigated in [10,14,16]. However, these solutions cannot be used since we require the clusterheads 'Masters' to be bounded and the communication between nodes in different clusters to be through 'Bridge' nodes, which are non-clusterhead nodes common to atleast two clusters. Moreover, for the distributed case, we do not assume full knowledge of the topology, and allow the neighbours to exchanges messages initially before running the algorithm on each node. This initial message exchanges between the nodes results in having 2-hop neighbourhood information at every node and keeps the number of rounds of messaging to a minimum because in order to exchange any information between two devices, they need to become a Master/Slave pair.

In this paper we show that the problem of determining a set of Master and Slave nodes such that the Masters communicate via Bridges only, is NP-Complete. Further we define the problem of minimising the number of Master nodes in the underlying graph, representing allowable communication links, such that the degree of each Master is bounded by k_1. We consider the characterisation of graphs by their clique cover number, i.e. the minimum number of cliques required to cover the vertices and edges. Such a characterisation arises naturally from the communication grouping in ad-hoc networks where each node in a group can communicate with every other node within a specified radius. We show that a centralised algorithm (i.e. one which has knowledge of the topology of the graph) gives an approximation bound of $(1+ [k_1 \min(s,n)/n])$ where s is the clique cover number and n is the number of nodes in the graph. When s is less than n/k_1 the approximation bound is 2, improving the trivial approximation ratio of k_1. The value of s is expected to be less than n/k_1 in most graphs that arise in ad-hoc communication networks, since devices which fall within a certain radius of each other will be in radio range, giving rise to cliques, and some of these devices would be in radio range of more than one clique. We also describe a simple distributed algorithm that attempts to mimic the centralised algorithm. Both these algorithms have been implemented and simulation results validate the good performance of the algorithms. Our algorithms add Master nodes (when they are unable to find bridges) to achieve connectivity.

The rest of the paper is organised as follows. Section B describes the problem formulation and our rationale behind it. Section C gives the proof of NP-Completeness. Section D describes the centralised algorithm and derives approximation bounds on the number of star clusters formed. Section E gives the distributed algorithm. Section F provides the simulation results and analysis. Section G recapitulates the main results. Appendix A presents the algorithms in detail.

B. System Model and Problem Statement

In an ad-hoc network, we assume that all nodes are alike and communicate over wireless links. We model the set of nodes as a graph, with an edge between a pair of nodes if they are in radio range of each other. It is assumed that a device can discover other devices within its radio range using device discovery protocols. The aim of the scatternet formation problem is to get a minimum set of connected star-shaped clusters of bounded size k_1. The connection between the stars is to be made through non-centre nodes, or Bridges, which, along with the clusterheads form the backbone of the network. We also impose a bound of k_2 on the number of stars to which a Bridge can be common. However, if there is no such connected scatternet, then we allow direct communication between two centre nodes or Masters. This is because, in Bluetooth, the Master device controls the traffic to all the Slaves, and a node can be active in only one piconet at a time. This means if a device is an active Slave in one piconet, then during this time, the entire piconet for which this device is the Master, has to be idle. Similarly, a device which is a Slave in more than one piconet is active in only one of the piconets at a time, and hence the number of piconets to which a Slave can be common should be limited.

Definition *(Clique-cover of a graph)* : An s-clique cover of a graph G, is a set of s cliques $C_1, C_2, \ldots, C_s$ (the size[2] of these cliques could be different) such that all the vertices and edges of G belongs to some clique.

Our first objective is to develop a centralised algorithm which assumes that there is a central entity with knowledge of the underlying topology. We derive approximation bounds in terms of clique-cover size. Our next objective is to develop a distributed algorithm which assumes that each node knows its 2-hop neighbourhood. This can be justified in the Bluetooth context since device discovery protocols *(inquiry procedures*) defined in [12, 13] allow nodes within radio range to discover each other, thus obtaining 1-hop neighbourhood information. We make the additional assumption that the devices exchange messages containing neighbourhood information, with all neighbours by making temporary Master-Slave connections. Thus the devices have 2-hop neighbourhood information. We also compare the performance of the above algorithms using simulation experiments.

C. NP-Completeness of the Scatternet Formation Problem

In this section we analyse the algorithmic hardness of the scatternet formation problem. For graph theory related terminology we refer to [15].

The problem of scatternet formation has two different problems associated with it. They are : (a) The Feasibility Problem : Given a graph, to find whether its nodes can be labelled as Master and/or Slave such that every Master has atmost k_1 Slaves attached to it and the set of Masters is connected through Slaves which act as Bridges, the number of Masters to which a Bridge can be connected being bounded by k_2, and (b) The Optimisation Problem : Given a graph to find a feasible solution (as in (a)) which also minimises the number of Masters.

We prove that the Feasibility Problem for general graphs is NP-Complete by reducing the hamiltonian path problem to this problem.

Theorem 1 : The Feasibility Problem of Scatternet Formation is NP-Complete.
Proof outline **:** To prove the Theorem, we consider the problem of finding a bounded degree spanning tree of a given graph such that any two adjacent vertices of the tree are of distinct degrees k_1 and k_2. We call this problem the (k_1,k_2)-bounded degree spanning tree problem. For a given graph G and two vertices x and y of G, the problem of finding hamiltonian path from x to y is equivalent to (k_1,k_2)-bounded degree spanning tree problem of the following graph

[2] Size of a clique is the number of nodes in the clique.

G'': Every vertex v of G is replaced by a path P_v with 4 vertices. Let the vertices of P_v corresponding to the vertex v of G, be labelled[3] as v_1,v_2,v_3 and v_4 as shown in Figure 1. In G' both the end vertices corresponding to P_u for some vertex u in G, are made adjacent to both the end vertices corresponding to P_v if and only if u and v are adjacent in G. Let x and y be two vertices of G. The graph G has a hamiltonian path from x to y if and only if G' has a hamiltonian path from x_1 to y_1. We obtain a graph G'' from G' as follows : (a) For the vertex x_1 (and similarly for y_1) in G', k_1-1 new vertices are added in G''and made adjacent to it , (b) for each vertex v_1 in G' corresponding to a vertex v in G (excluding x and y), k_1-2 new vertices are added in G'' and made adjacent to it, (c) for each vertex v_3 in G' corresponding to a vertex v in G (including x and y) k_1-2 new vertices are added in G'' and made adjacent to it, and (d) for the vertex v_2 (and similarly for v_4) in G' corresponding to a vertex v in G (including x and y) k_2-2 new vertices are added in G'' and made adjacent to it. This construction is illustrated for an example in Figure 2. We note that G'' has a (k_1,k_2)-bounded degree spanning tree if and only if G has a hamiltonian path from x to y. This shows that the problem is NP-Hard. Furthermore, the problem is easily seen to be in NP. Hence the Theorem. □

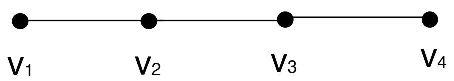


Figure 1: The graph P_v of Theorem 1

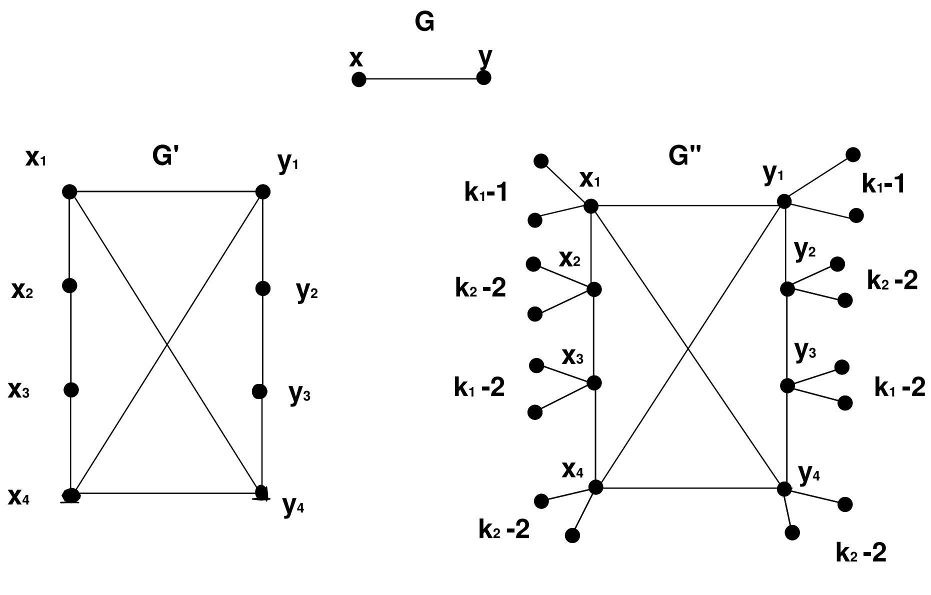


Figure 2 : Illustrative example for the proof of Theorem 1

Theorem 2 : The Optimal Scatternet Formation problem is NP-Hard.
Proof outline **:** Follows from the NP-Hardness of the following problem: Determine if there exists a feasible solution to the scatternet formation problem with n masters. The NP-Hardness of this problem is proved by the same reduction used in Theorem 1. □

[3] The subscript corresponding to any vertex of G should be interpreted similarly.

D. A CENTRALISED GREEDY ALGORITHM

1) Algorithm overview

The greedy algorithm proceeds in two phases. In the first phase it creates a forest and in the second phase it merges the trees in the forest.

First Phase : In the first step of this phase all the nodes are set the label “unmarked” except for the node with the highest degree in the entire graph which is marked as a Master (if there are more than one node with highest degree the node with largest device id is picked). If the degree of the node marked as Master is less than the cluster size k_1 (the maximum number of Slaves that a Master can have), then all its neighbours are marked as its Slaves. Otherwise, the neighbour set of this node is sorted in non-decreasing order of degrees (with the nodes having the same degrees sorted in increasing order of their device id’s) and the first k_1 nodes in the sorted list are marked as its Slaves.

In general, until all the nodes are marked either as “Master” or “Slave”, the algorithm iterates as follows: It picks up the node, say v, with the highest effective degree[4] (if there is a tie the node with largest device id is picked) and marks it a Master.

Suppose the node marked as Master is adjacent to one or more nodes marked as Slave in the previous iterations. Among these Slaves the node with smallest device id is made a Slave (a Bridge) of v, the newly picked Master. Then, the unmarked neighbours of the Master node , v, are sorted in non-decreasing order of their effective degrees (the neighbours with same effective degrees are sorted in increasing order of their device id’s) and the first k_1-1 neighbours in the sorted list are marked as Slaves to this Master.

Alternately, if the node marked as Master is not adjacent to any node marked as a Slave in the previous iterations, then the unmarked neighbours of this Master are sorted in non-decreasing order of their effective degrees (the neighbours with same effective degrees are sorted in increasing order of their device id’s) and first k_1 neighbours in the sorted list are marked as Slaves to this Master.

Second Phase (Cleanup Phase) : The forest obtained in the first phase is merged to get a single tree as follows[5]. As a first step, for every pair of trees T_i and T_j, if there is a node marked as slave in T_i and adjacent to a node labelled Master in T_j which is unsaturated[6], then make the slave in T_i also a Slave of the Master in T_j (the Slave would be a Bridge node between these two Masters in T_i and T_j). By doing this, the trees T_i and T_j are merged. After the first step if the forest reduces to a single tree stop. Otherwise, as a second step for every pair of trees T_i and T_j examine whether there exists an edge between a Master in T_i and a Master in T_j. Then add the Master-Master edge between and merge the trees T_i and T_j. After Step 2, if there are more than one tree left in the forest proceed to Step 3, else stop. In Step 3, to merge two trees T_i and T_j look whether there is a node in T_i labelled as Slave which is adjacent to a node labelled as Master in T_j which is saturated[7] (such a Slave can’t be made a

---

[4] The *effective degree* of an unmarked node is the degree of the node after removing the edges to the nodes that are marked as “Master” or “Slave”.

[5] While merging we prioritise reducing number of masters to maintaining connectivity.

[6] A Master is unsaturated if some more Slaves can be added to it. That is it has less than k_1 Slaves, where k_1 is the number of Slaves a Master can have.

[7] A saturated Master is a Master which is not unsaturated.

Slave of the Master to merge the two trees since the Master is saturated). If such a Slave exists in T_i, then connect the trees by adding the edge between this Slave and the Master node in T_j and, relabel the Slave as "Master". If a single tree is obtained by merging, the algorithm stops, else proceeds to the next step. As a last step, for every pair of trees T_i and T_j do the following. Merge the trees T_i and T_j by adding an edge between a Slave of T_i to a Slave of T_j. Among these two Slaves one of these Slaves is relabelled as Master. This ends the second phase and the algorithm terminates. When the algorithm terminates a single tree is obtained.

2) Time Complexity of the Algorithm

Let Δ be the maximum degree of a graph. In the first phase, the algorithm picks the node with largest effective degree as Master and performs sorting in non-decreasing order after Master assignment. It takes O(n) for finding the largest effective degree node and $\Delta \log \Delta$ time for every sorting since the Master has atmost Δ neighbours. In the second phase, there are atmost $O(n^2)$ pairs to be examined. Therefore, the worst case the runtime of the algorithm is $O(n^2+n\Delta\log\Delta.)$.

3) Approximation Bounds

The following result gives an approximate bound on the number of masters obtained by the Centralised Greedy Algorithm compared to the optimal value that could be obtained by any algorithm. For a real number r, [r] denotes the least integer that is greater than or equal to r.

Theorem 5 : *Let G be a graph with n vertices and let* $C_1, C_2, \ldots, C_s$ be a clique covering of G. Let m be the number of masters obtained by the Centralised Greedy Algorithm in its first phase. Then, $[n/k_1] \leq m \leq [n/k_1] + \min(s,n)$.

Proof outline : For the cliques C_i, we define its intersection nodes to be the nodes that are also present in another clique C_j and interior nodes to be nodes that are present only in that clique. We note that the Greedy Algorithm would label intersection nodes (if any) of a clique C_i, as Master since they would be of higher degree than the interior nodes. The proof of the theorem is by induction on the number of nodes. Assume for simplicity that $s < n$. The case when $s \geq n$ can be handled similarly. Let t be the number of nodes of a graph for which the Greedy Algorithm needs to assign Master and Slave labels at some iteration. We prove the Theorem by assuming that the Theorem is true for graphs with t-1 or less nodes . That is for the graphs with t-i nodes, where $i \geq 1$, the number of nodes labelled as Master by the Greedy Algorithm is atmost $[(t-i)/k_1] + b$, where b is the number of cliques remaining at that iteration, which is obtained from the cliques $C_1, C_2, \ldots, C_s$ by removing the nodes labelled Master and/or Slave in the iterations before that iteration. We prove that the number of masters assigned by the Greedy Algorithm for a graph with t nodes is atmost $[t/k_1]+u$ where u is the number of cliques remaining at that iteration. This holds because of the following: the number of nodes labelled as Master by the Greedy Algorithm with t nodes, $m_t = m_{t-i}+1$ where m_{t-i} is the number of nodes that would be labelled as Master by the Greedy Algorithm with the graph (which has t-i nodes) that would be obtained by deleting i nodes that are labelled Master and Slave in the current iteration. Then, $m_t \leq [(t-i)/k_1]+ (u-j) +1$, where u-j is the number of cliques that would be required in the clique cover after the current iteration. If $[(t-i)/k_1]<[t/k_1]$ then $m_t \leq [t/k_1]+ u$. Otherwise, $j \geq 1$ (that is the number of cliques decreases atleast by 1, since the node assigned as Master in the current iteration, has all the nodes of the cliques that are eliminated from the clique cover as its Slaves) which implies that $m_t \leq [t/k_1]+ u$. □

Corollary: The Centralised Greedy Algorithm is a 1+ [k_1 min(s,n)/n]-approximation if the input graph with n vertices has a clique covering with s cliques.
***Proof outline* :** By following the proof of the above Theorem for any clique their intersection nodes are marked as Masters before their interior nodes. So, any two trees which are created by the algorithm in its first phase can be merged by adding a Master-Master edge in its second phase. Therefore, from the above Theorem the result follows. □

E. A Distributed Algorithm

For a meaningful application of any algorithm for ad-hoc network formation, it has to be completely distributed since usually, no central infrastructure exists in such networks. In order to minimise the number of message exchanges, while providing some useful information to the nodes about their neighbourhood, we let each node exchange messages containing immediate neighbourhood. Once the 2-hop neighbourhood information is known, the following distributed algorithm is run (multiple times) at every node till it gets marked as Master or Slave and gets connected to all the nodes in its 2-hop neighbourhood.

1) Algorithm Overview

Each node u of the ad-hoc network executes the following Distributed Algorithm in two phases. In the first phase every node labels itself Master or Slave and in the second phase they connect themselves to Masters which are not connected to it in the first phase. Each node runs the Algorithm till it gets assigned the label Master/Slave and starts the second phase after labelling itself and ensuring that its 2-hop neighbourhood is completely labelled.

First Phase : Every node gets its 2-hop neighbourhood information by message passing (that is the nodes which are 1-hop and 2-hop away from it and their degrees). After getting the 2-hop neighbourhood information, every node labels itself "unmarked". Then, the algorithm goes through several runs[8] and in each run the nodes either remain unmarked or mark itself Master or Slave by the following rule. If a node finds itself to be the largest effective degree[9] node in its 2-hop neighbourhood then the node marks itself a Master (if there is more than one node with largest effective degree in the 2-hop neighbourhood of a node inclusive of itself, the node marks itself a "Master" if it is of largest device id among these nodes). After a node say v, marks itself a Master it examines whether it is adjacent to a node which is already marked as a Slave. If it finds such a set of nodes S, it requests the node with the smallest device-id in S to be its Slave[10] (to be a Bridge node between this Master, v, and the Masters for which this node is a Slave). Then, v, sorts its unmarked neighbours in non-decreasing order of their effective degrees (if there are more than one node with same effective degree then they are sorted on increasing order of their device id's) and requests the first k_1 -1 nodes to become its Slave. The requested nodes marks themselves as Slaves for the requesting Master. If the node v, after marking itself a Master, doesn't find itself adjacent to any Slave it proceeds as follows. The Master node v, sorts its unmarked neighbours in non-decreasing order of their effective degrees (if there are more than one node with same effective degree

[8] In every "run" a node checks whether it can make itself a master by looking at its 2-hop neighbourhood. If it can't then goes to a sleep state till the effective degree of some node in its 2-hop neighborhood changes. If it can, then as a first step marks itself a Master, as a second step wakes some of its neighbours and requests them to become its slave (following the algorithm).

[9] The definition of effective degree is same as in the case of Centralised.

[10] The requested node marks itself a Slave for the requesting Master.

then they are sorted on increasing order of their device id's) and requests the first k_1 nodes to become its Slave. The requested nodes mark themselves as Slaves for the requesting Master. This ends the first phase. As a result of first phase, a forest is formed. Every node also gets a tree-id in the first phase for identifying whether the node and another node in its 2-hop neighbourhood belong to the same tree or not, as follows. In every run a Slave inherits the tree-id of its Master and every Master either assigns itself a new tree-id (which could be its device-id) or inherits the tree-id of the Bridge.

Second Phase : The second phase of the algorithm connects the different trees in the forest formed in the first phase by the following rule[11]. A Master tries to connect to every other Master in its 2-hop neighbourhood which are of different tree-id's tries to connect through one of its Slaves (that is finds whether one of its Slaves could also become a Slave for the other unsaturated[12] Master). If not, it connects to neighbouring Masters by Master-Master edge. For each remaining Master in its 2-hop neighbourhood, it looks for a Slave that is adjacent to each such Master, and relabels its Slave as Master, introducing a Master-Master edge. For each non-neighbouring Slave (with a different tree-id) in its 2-hop neighbourhood, it picks the node with lowest device id among its Slaves. The Slave picked (i) relabels itself as Master, (ii) adds the Master-Master edge corresponding to its adjacent Master and (iii) requests a Slave (with a different tree-id), to be its Slave (Bridge). The requested Slave marks itself a Slave of the requesting Master. This completes the second phase of the algorithm. At the end of the second phase, every node is connected to every other node in the network through the backbone obtained by this algorithm.

2) Correctness and Analysis

Theorem 4 : The Distributed Algorithm is correct and terminates**.**
Proof outline **:** It follows from the Algorithm that every node gets a unique label Master or Slave without any ambiguity. Also, in each iteration of the first phase of this algorithm atleast one node gets marked itself as Master. These imply that the algorithm stops and every node marks itself Master or Slave. □

Theorem 5 : The nodes that are marked as Masters by the Centralised Algorithm in its first phase are exactly the nodes that are marked as Masters by the Distributed Algorithm in its first phase.
Proof outline **:** In the first iteration, the centralised algorithm picks the global maximum node which is also picked by the Distributed Algorithm, as it is also a local maximum in its 2-hop neighbourhood. In the second iteration, the global maximum node picked by the Centralised Algorithm may be in the 2-hop neighbourhood of the global maximum picked by the Centralised Algorithm in its first iteration or not. If it is not, then it must have been the local maximum and hence should have been picked by the Distributed Algorithm in its first iteration. If it is, then the Distributed Algorithm also picks it in its second iteration. The same argument holds inductively in the subsequent iterations too. So, the nodes which are marked as Master by the Centralised Algorithm in its first phase are also marked as Master by the Distributed Algorithm in its first phase.
Conversely, the nodes which are marked as Master by the Distributed Algorithm in the first phase are also marked as Master by the Centralised Algorithm in the first phase. Let u be a

[11] Minimising the number of masters is prioritised over establishing connectivity. So, the algorithm goes through the following steps one after another.
[12] As defined in the Centralised Algorithm.

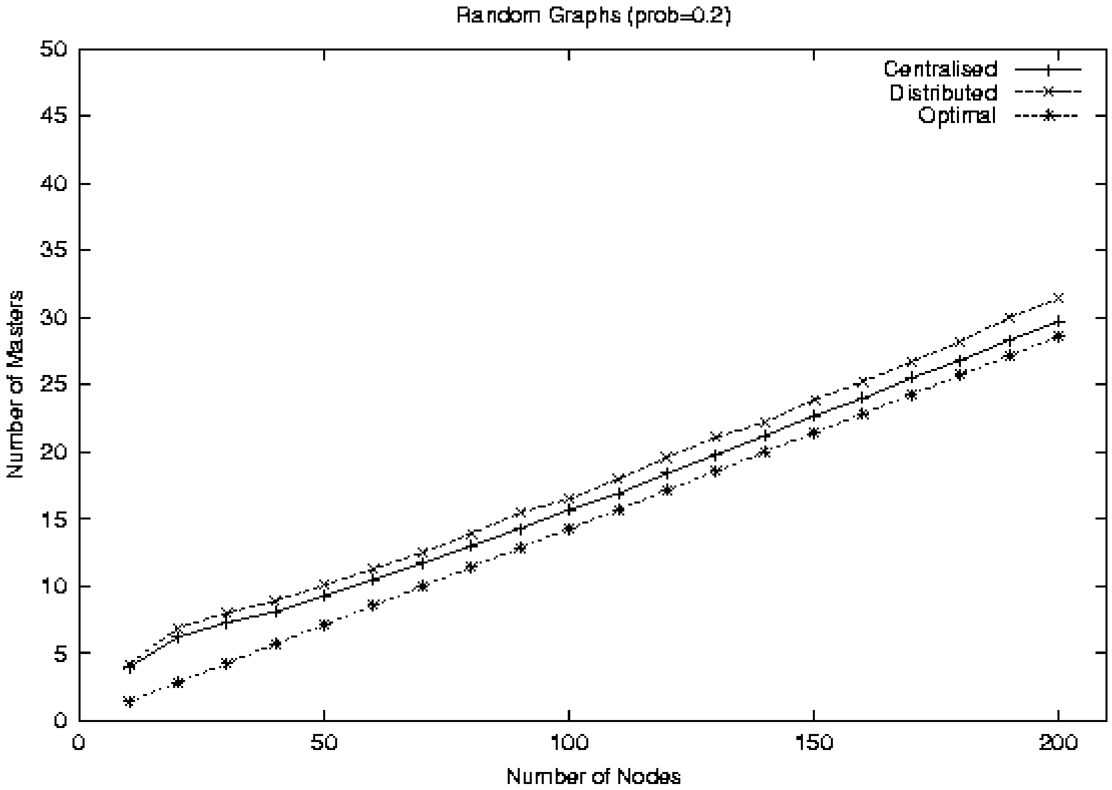
vertex marked as Master by the Distributed algorithm. Suppose, the Centralised Algorithm doesn't mark u as Master (marks it as Slave). Let v be the Master of u. So, the Distributed Algorithm would also mark v as Master (as proved in the converse). This implies that v marks u as Slave. This contradicts that the Distributed Algorithm marks u as Master. □

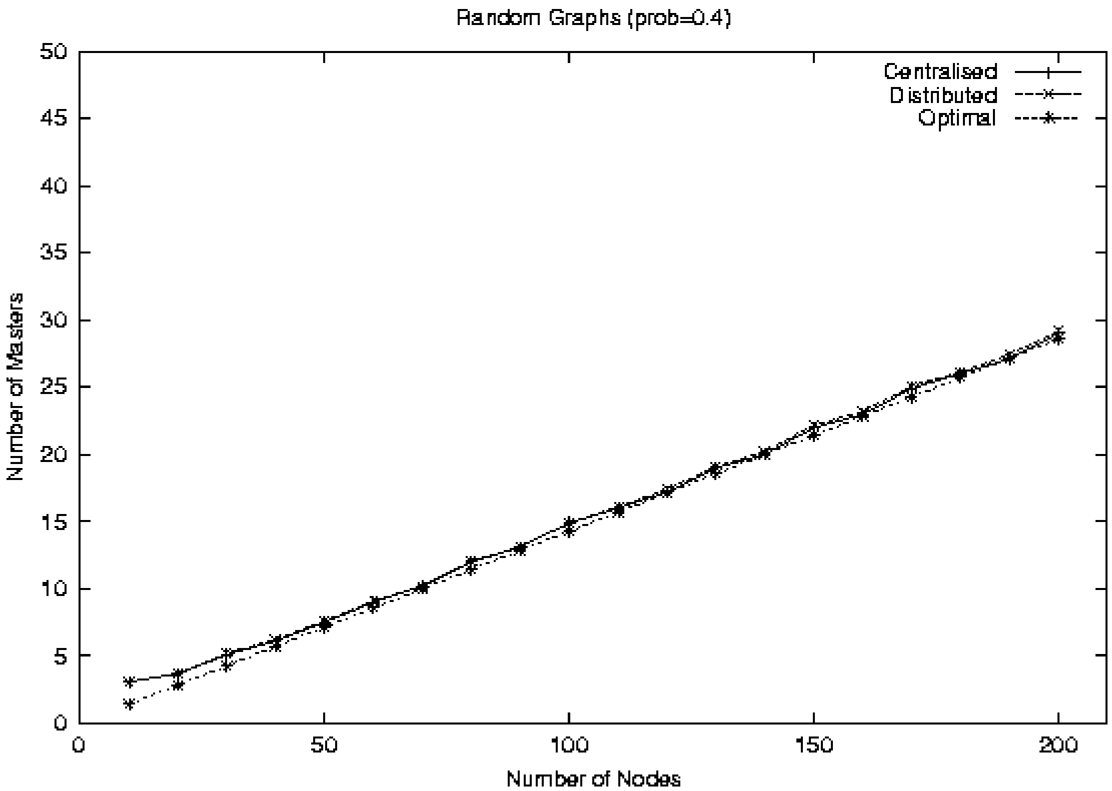
Lemma : The number of nodes labelled as Master by the Distributed Algorithm is atleast the number of nodes labelled as Master by the Centralised Algorithm.
Proof outline **:** The Distributed Algorithm in its second phase assigns atleast as many Masters as that of the second phase of the Centralised Algorithm. This is because, the Centralised Algorithm in its second phase creates atmost one Master, to merge two trees in the forest created in its first phase, whereas, in the Distributed Algorithm, each Master tries to connect to every other Master (with a different tree-id) in its 2-hop neighbourhood. Therefore, each node that is relabelled as Master by the Centralised Algorithm in its second phase is also relabelled as Master in the second phase of the Distributed Algorithm. So, the result follows from Theorem 5. □

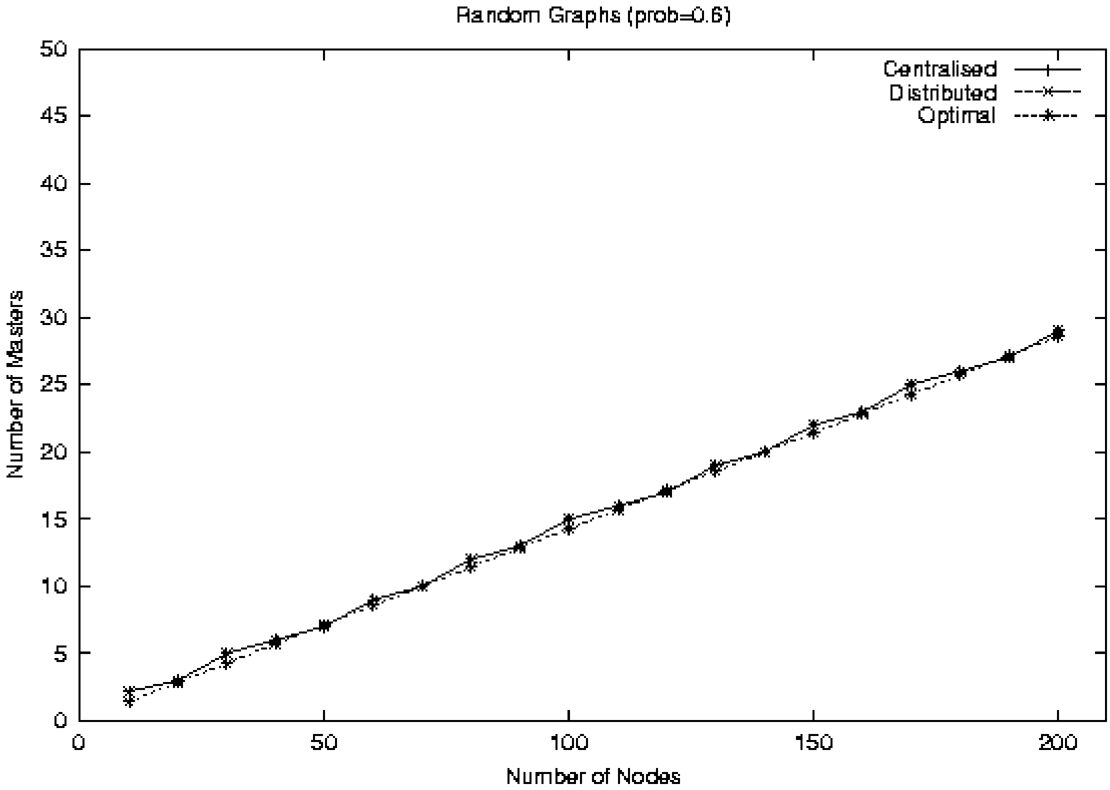
F. SIMULATION RESULTS

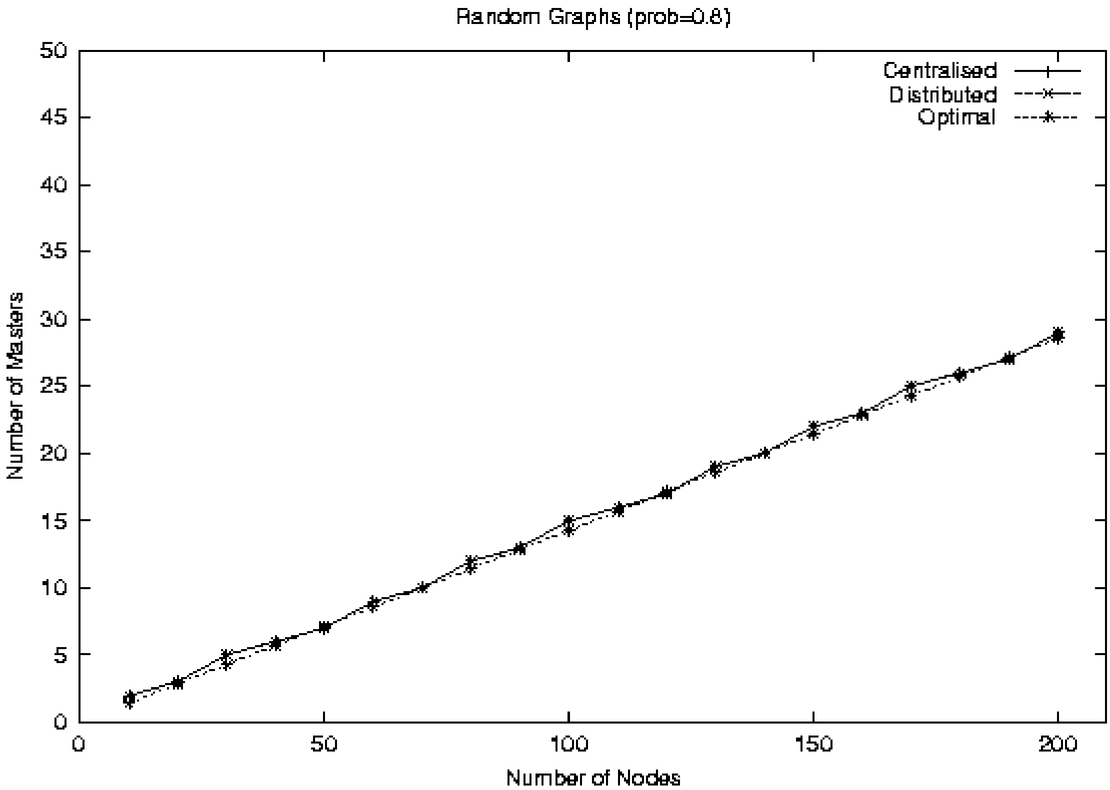
The following plots compare the number of masters created by the Distributed Algorithm with those obtained by the Centralised Algorithm as well as "Optimal" (which means the trivial lower bound n/7 where n is the number of nodes in the graph). GT-ITM (Georgia Tech Internetwork Topology Models) [17] was used to generate the graphs, Random graphs and Locality graphs. The Random graph model distributes vertices at random locations in a plane and then considers each pair of vertices; an edge is added between a pair of vertices with probability α. For the purposes of our simulation, α (referred to as "prob" in the plots) was varied from 0.2 to 1.0 in steps of 0.2. The Locality model is intended to relate edge probability to the distance between vertices. All the vertices that are within a specified radius are created with probability 1, and outside the radius with probability 0. A higher value of "gamma" in the Locality graph plots imply a larger radius, hence more connectivity. For each point in the plots in both classes of graphs, 200 graphs were generated and the average value was used for the plots.

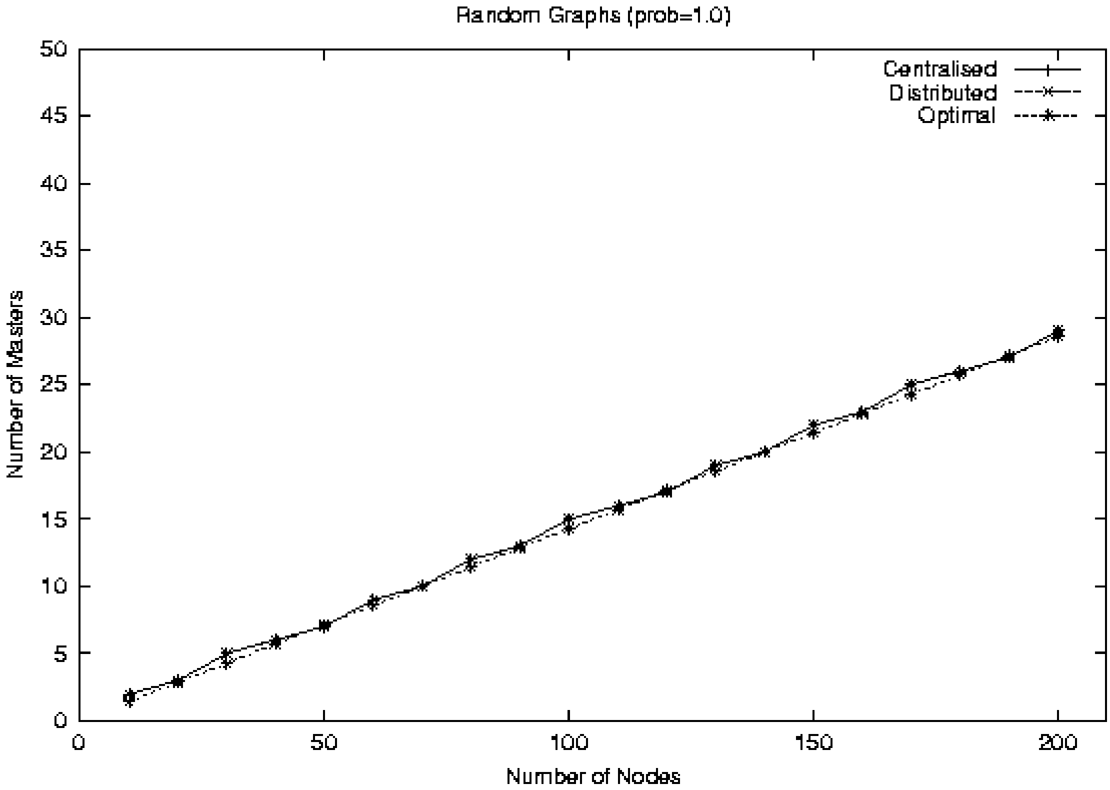
In the case of Random graphs, the number of masters created by the Distributed Algorithm is very close to those created by the Centralised Algorithm and Optimal, especially for dense graphs. The same is true for Locality graphs. However, the Distributed Algorithm does not perform as well for sparse graphs (prob=0.2 or gamma=0.2, respectively), especially for a large number of nodes. The reason for this is perhaps that the number of masters created in the second phase by the Centralised Algorithm is proportional to the number of trees created in its first phase, whereas the number of masters created in the second phase by the Distributed Algorithm is proportional to the number of masters created in its first phase.

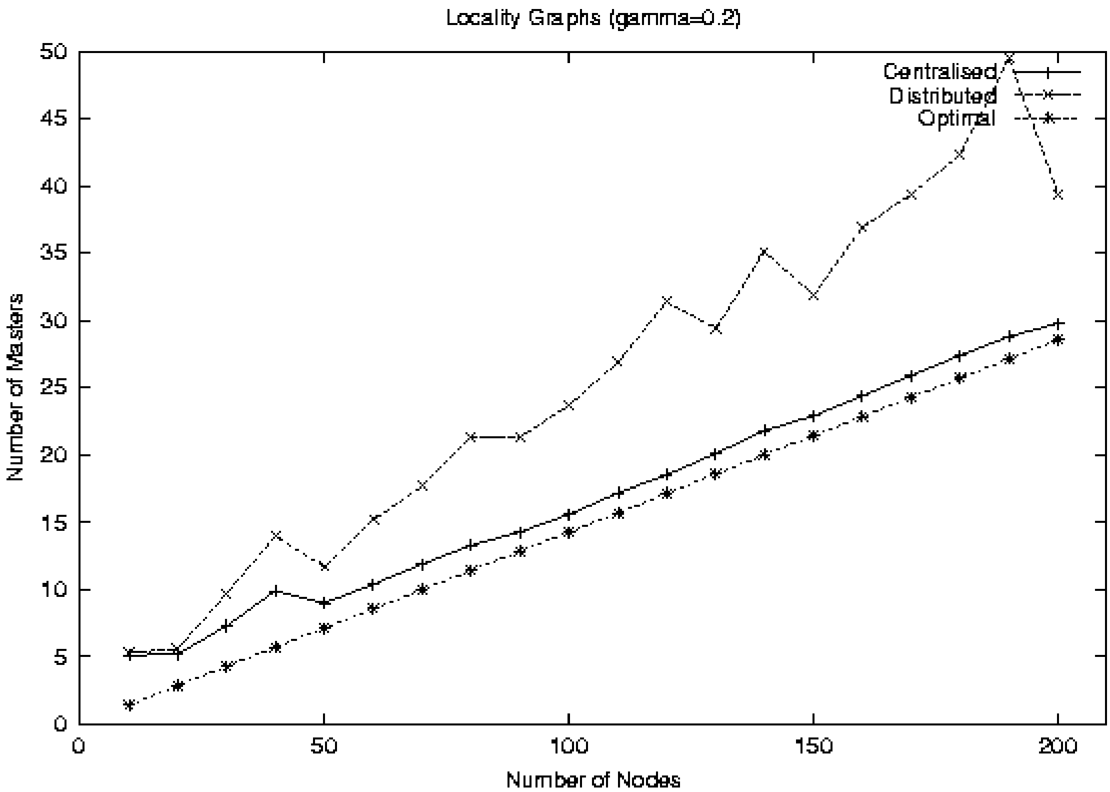

Random Graphs (prob=0.2)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200

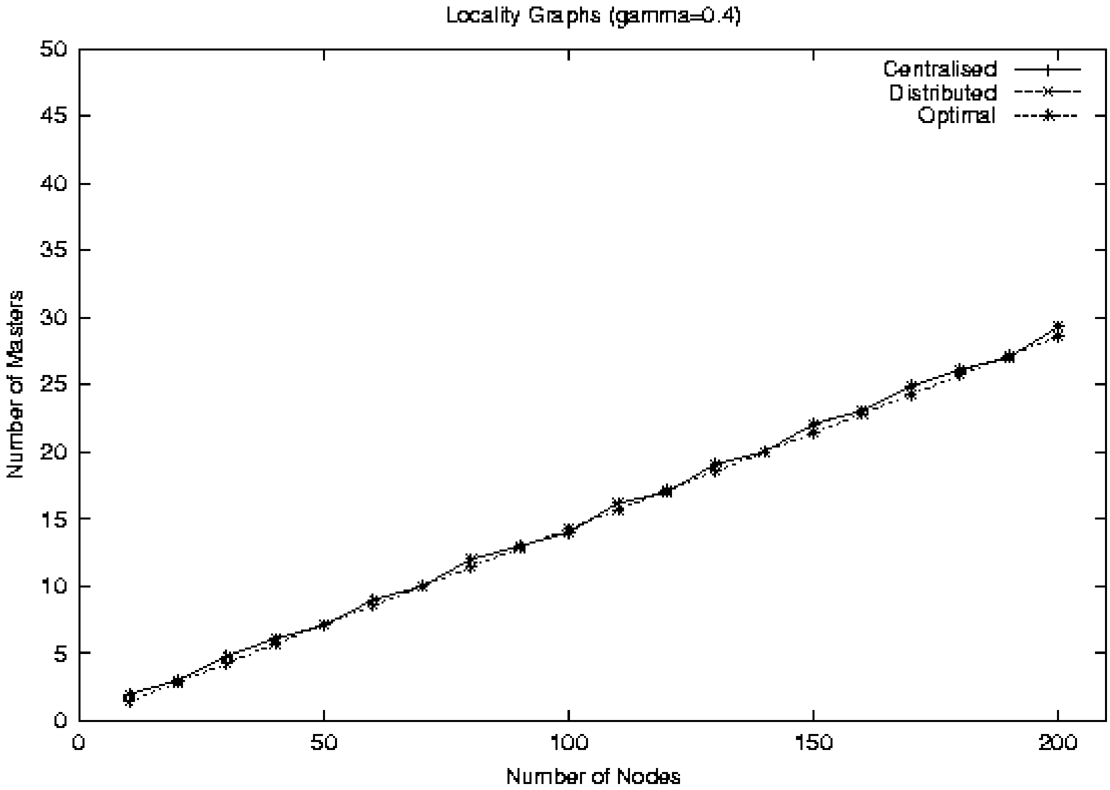

Random Graphs (prob=0.4)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200

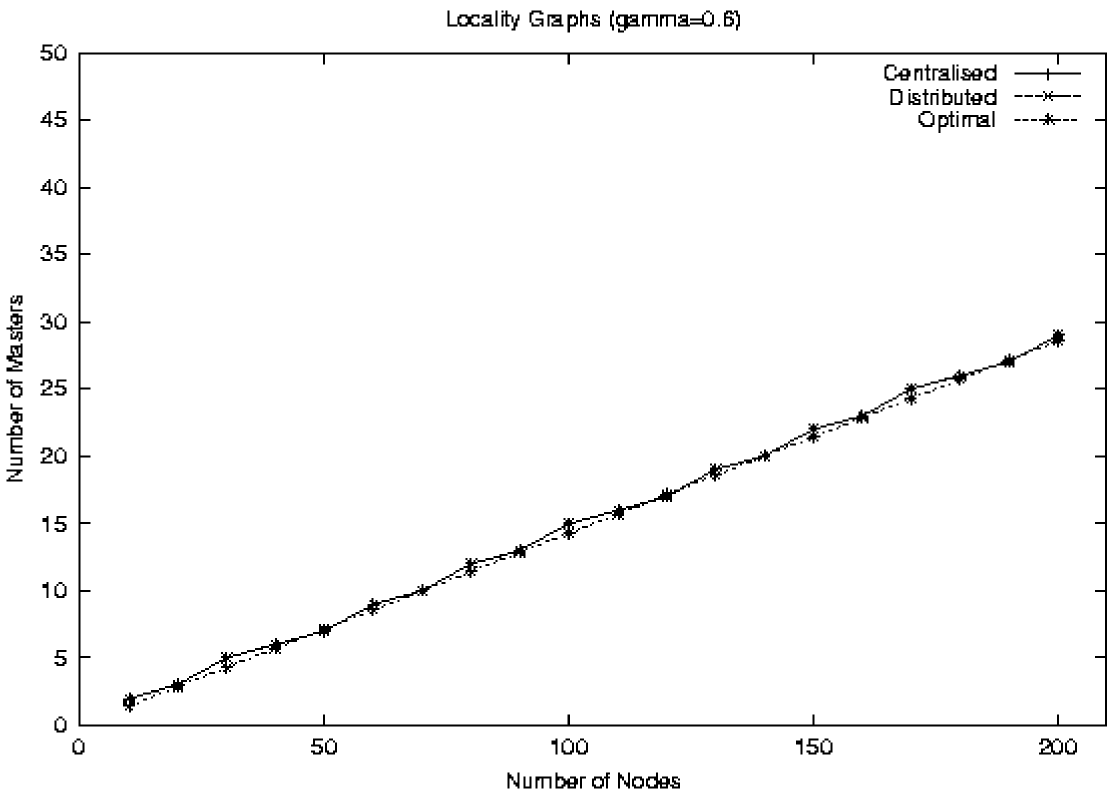

Random Graphs (prob=0.6)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200

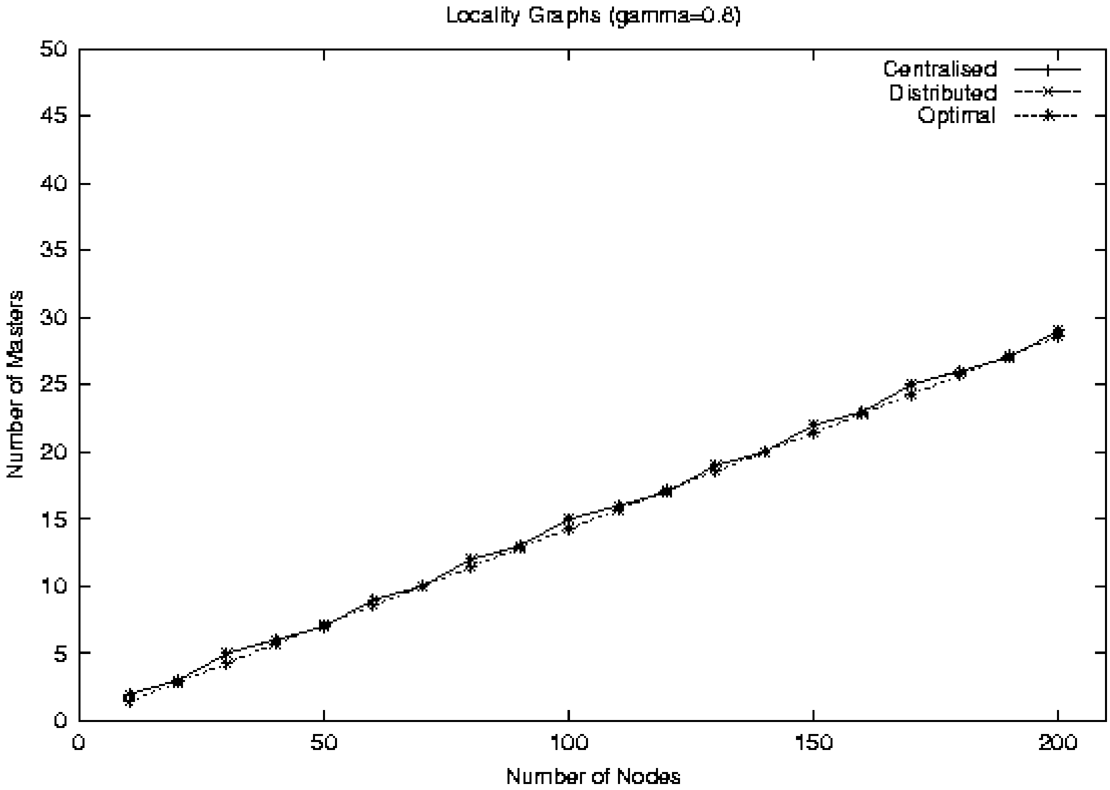

Random Graphs (prob=0.8)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200

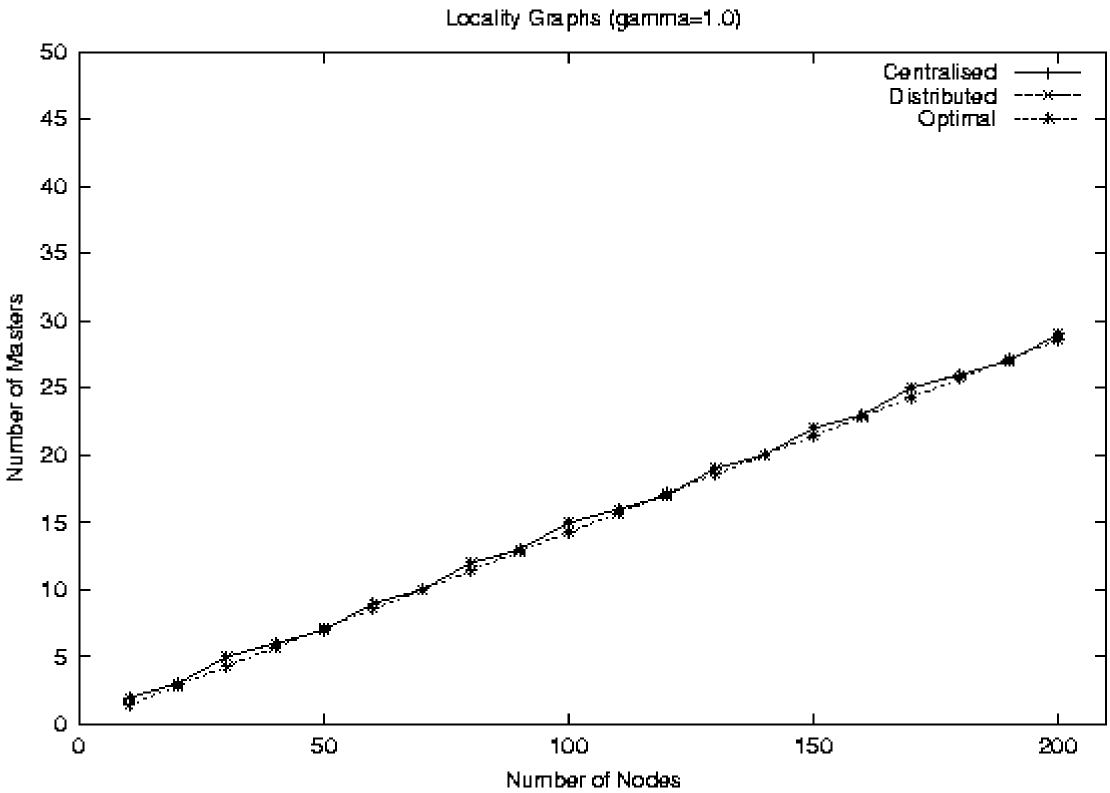

Random Graphs (prob=1.0)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200


Locality Graphs (gamma=0.2)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200


Locality Graphs (gamma=0.4)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200


Locality Graphs (gamma=0.6)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200


Locality Graphs (gamma=0.8)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200


Locality Graphs (gamma=1.0)
Centralised
Distributed
Optimal
Number of Masters
Number of Nodes
0
5
10
15
20
25
30
35
40
45
50
0
50
100
150
200

G. Conclusions

In this paper, we have given a graph theoretical formulation of the Bluetooth scatternet formation problem and proved the algorithmic hardness of the feasibility and the optimisation problems associated with it. We proposed a centralised greedy algorithm and derived an approximation bound of 1+ [k_1 min(s,n)/n] for it, where s is clique-cover size, k_1 is the maximum bound on the number of Slaves that any Master can have, and n the number of nodes in the network. A distributed algorithm assuming the 2-hop information is also proposed. The performance of these algorithms were analysed using simulation experiments. The experiments show the good performance of these algorithms in the networks that arise in practice.

ACKNOWLEDGEMENTS: The authors would like to thank Indraneel Chakraborty and Sundeep Kapila for the implementation of a preliminary version of the algorithms presented in this paper.

REFERENCES:

[1] H.H. Abu Amara, "Fault-tolerant distributed algorithm for election in complete networks," IEEE Trans. Computers, vol. 37, no. 4, pp.449-453, Apr. 1988.
[2] H.H. Abu Amara and J. Lokre, "Election in asynchronous complete networks with intermittent link failures," IEEE Trans. Computers, vol. 43, no. 7, pp.778-788, July 1994.
[3] M. Gerla and J.T.C. Tsai, "Multicluster, mobile, multimedia radio network," ACM Baltzer Journal of Wireless Networks, vol. 1, no. 3, pp.255-265, Oct. 1995.
[4] H.H. Abu Amara and A. Kanevsky, "On the complexities of leader election algorithms," 5th Int. Conf. on Computing and Information, 1993.
[5] G. Singh, "Efficient distributed algorithms for leader election in complete networks," 11th Int. Conf. on Distributed Computing Systems, 1991.
[6] R. Dechter and L. Kleinrock, "Broadcast communication and distributed algorithms," IEEE Trans. Computers, C-35, pp.210-219, 1986.
[7] A.D. Amis, R. Prakash, T.H.P. Vuong, and D.T. Huynh, "Max-min D-cluster formation in wireless ad hoc networks," IEEE INFOCOM 2000.
[8] R. Krishnan, R. Ramanathan, M. Streenstrup, "Optimization algorithms for large self-structuring networks". IEEE INFOCOM 1999.
[9] C.V. Ramamoorthy, J. Srivatsava and W.T. Tsai, "Clustering techniques for large distributed systems," IEEE INFOCOM 1986.
[10] J. Wu and H. Li, "On calculating connected dominating set for efficient routing in ad hoc wireless networks," DIAL M'99.
[11] L. Ramachandran, M. Kapoor, A. Sarkar, A. Aggarwal, "Clustering algorithms for wireless ad hoc networks," DIAL M 2000.
[12] http://www.bluetooth.com
[13] http://www.bluetooth.net
[14] B. Das, E. Sivkumar and V. Bhargavan, "Routing in ad hoc networks using a spine," IEEE Int. Conf. On Computers, Communication, and Networks, pp.1-20 (IC3N), Sept. 1997.
[15] J.A.Bondy and U.S.R.Murty, "Graph theory with applications", American Elsevier, NewYork, 1976
[16] S. Guha and S. Khuller, "Approximation algorithms for connected dominating sets," Algorithmica, 20(4):374-387, April 1998.
[17] http://www.cc.gatech.edu/projects/gtitm

APPENDIX A

1) Detailed Pseudo code for Centralised Algorithm

```
centralised_algorithm( )
  {
     label_setting( ) ;
     merge_trees( );
    }//end of centralised algorithm

label-setting( )
input : a graph
output : a set of trees (which is obtained by connecting the masters to its slaves)
//This phase (first phase) sets the labels of the nodes.
   {
       cluster_size := k1 ;
       for (all nodes u) initialise(u);
       while (there are "unmarked" nodes)
         {
          pick the node v with highest effective_degree;
          label(v)= "Master";
          eligible_bridges := { u :  u is adjacent to v and u is marked as
"Slave"};
          if (eligible_bridges is not empty)
             {
                pick the node u with smallest device id from eligible_bridges;
                slaves(v)  :=  {u} U slaves(v);
                masters(u) :=  {v} U masters(u);
                capacity(v) := capacity(v) - 1 ;
              }

         if (effective_degree(v) <= capacity(v)) then
            {
              for (every neighbor u of v) mark_slave(u,v);
            }
        else
             {
               list := sort_non-decreasing(neighbor(v));
               eligible_slaves(v) := { ui  of list :   i <=   capacity(v) };
//first k elements are picked
                                          // from the set "eligible_slaves"
               for (all u which belongs to eligible_slaves(v))  mark_slave(u,v);
              }
        for (all nodes u) update_effective_degree(u);
          }// end while loop
 }//end of label_setting phase


sort_non-decreasing( )
//sorts in non-decreasing order of effective degrees and if there are more than one
node with //the same effective degree sorst them by increasing order of their
device id's
   {
      }

merge_trees( )
input: a set of trees {T1, T2,...Ts}
output: a single tree {T}
//phase 2 of the algorithm it merges the trees obtained in the first phase
{
    S := {T1, T2,...Ts};
    (a) while (| S | > 1 )
       {
           i := 1 and j = 2;
          (1) try_Master-Slave-Master_1(Ti , Tj );//case where the Master is
unsaturated
            if (number_of_trees_reduce =  "yes")  update(S) and goto (a);
            else j:=j+1 and goto (1);

          i := 1 and j = 2;
```

```
         (2) try_Master-Master(Ti , Tj );
            if (number_of_trees_reduce = "yes") update(S) and goto (a);
            else j:=j+1 and goto (2);

          i := 1 and j = 2;
         (3) try_Master-Slave-Master_2(Ti , Tj ); //case where the Master is
saturated
           if (number_of_trees_reduce = "yes") goto update(S) and (a);
            else j:=j+1 and goto (3);

          i := 1 and j = 2;
         (4) try_Master-Slave-Slave-Master(Ti , Tj );
            if (number_of_trees_reduce = "yes") update(S) and goto (a);
            else j:=j+1 and goto (4);

            } // end while
   } //end of merge_trees


intialise(u)
//initialises u
 {
    label(u) := "unmarked";
    slaves(u) := empty, masters(u) := empty;
    effective_degree(u) := degree(u);
    capacity(u) := cluster_size;
  } //end of initialisation


mark_slave(u,v)
//marks u as slave of v
 {
     slaves(v) := {u} U slaves(v);
     capacity(v) := capacity(v) -1;
     label(u) := "Slave";
     masters(u) := {v} U masters(u);
   }//end

update_effective_degree(u)
//updates effective degree of u
{
      labelled_neighbours(u) := {w : w is a neighbor of u which is marked as
"Master" or                                                           "Slave"};
      effective_degree(u) := degree(u) - number of nodes in labelled_neighbours ;
   } //end

update(S)
//update the tree collection after merging
 {
     Merge the tree Tj and Ti ;
      S := S - {Tj};
  }//end

try_Master-Slave-Master_1(Ti , Tj)
  {
     for (all nodes u and v labelled as "Master" in Ti and Tj respectively)
        {
            if (it is possible to make a slave of u also a slave of v)
                {
                    make the slave of u also a slave of v ;
                    capacity(v) := capacity(v)-1 ;
                    return number_of_trees_reduce := "yes";
                  }
          }
     return number_of_trees_reduce := "no";
  } //end

try_Master-Master(Ti , Tj)
  {
      for (all nodes u and v labelled as "Master" in Ti and Tj respectively)
        {
            if (u is a neighbor of v)
```

```
                  {
                      merge the trees by adding the edge (u,v);
                      return number_of_trees_reduce := "yes";
                  }
          }
     return number_of_trees_reduce := "no";
  } //end

try_Master-Slave-Master_2(Ti , Tj)
  {
     for (all nodes u and v labelled as "Master" in Ti and Tj respectively)
        {
            if (it is possible to make, a slave of u also a slave of v but
capacity(v) = 0)
                  {
                    merge the trees by adding the edges (u,slave) and (slave,v)
where slave is the                           common slave between u and v;

                    relabel the slave as Master;
                    return number_of_trees_reduce := "yes";
                  }
          }
     return number_of_trees_reduce := "no";
  } //end

try_Master-Slave-Slave-Master(Ti , Tj)
  {
     for (all nodes u and v labelled as "Master" in Ti and Tj respectively)
        {
            if (a slave of u is adjacent to a slave of v, namely, slave_u and slave_v,
respectively)
                  {
                    merge the trees by adding the edges (u,slave_u), (slave_u,slave_v)
and  (slave_v,v);
                    relabel slave_u as Master;
                    return number_of_trees_reduce := "yes";
                  }
          }
     return number_of_trees_reduce := "no";
  } //end
```

2) Detailed Psuedo code for the distributed algorithm

```
initial_code(node v)
// initially every node v runs this code
{
    effective_degree(v):= degree(v);
    2-hop-neighborhood := get_2-hop_information( ); //including the degree of its

//neighbors-neigbors
    2-hop-neighbors := all the nodes in the 2-hop neighborhood ;
    for (every node u in the 2-hop-neighborhood of v) effective_degree(u) :=
degree(u);
    label(v) := "unmarked";
    message(v) := none ;
    masters(v) := empty;
    slaves(v) := empty;


    treeid(v) := device_id(v); //treeid is set to its device_id initially
    distributed_algorithm(v);
  }

distributed_algorithm(node v)
//This algorithm is executed at every node v of the graph
  {
     if (label(v)= "unmarked") label_setting(v) ; //where v is the node in which
the algo is run         if (label(u) = "Master" or "Slave" for  every node u in the
2-hop neighborhood of v                                 inclusive of itself)
```

```
        {
            connect_local( );
          }
 }//end of distributed algorithm


label-setting(v)
//In this phase (first phase) a vertex v sets itself a "Master" or "Slave"
   {
          if (message(v) = "become slave of u ")
               {
                  label(v) := "Slave";
                  masters(v) := masters(v) U {u};
                  treeid(v) := treeid(u);
               }//end if

         else
              {
              if (effective_degree(v) >= effective_degree(u) and device_id(v) >
device_id(u)) for                              every u in  2-hop-neighobors(v) such
that label(u)= "unmarked")
                    {
                     label(v) := "Master";
                     set_slaves(v);
                     for (all the nodes u in the 2-hop neighborhood of v)
                          {
                            update_effective_degree(u);
                            distributed_algorithm(u);
                           }//end for
                     } //end if
               }//end else
   }//end of label_setting phase

set_slaves(v )
  {
      eligible_bridges(v) := {u : u is a neighbor of v and label(u)= "Slave"}
      if (eligible_bridges(v) is not empty)
          {
             pick the node u with smallest device_id from eligible_bridges;
             slaves(v):={u} U slaves(v);
             treeid(v) := treeid(u) ;
             capacity(v):=capacity(v)-1;
             message(u):= "become slave of v";
            }
      unmarked_neighbours := neighbours of v which are marked "unmarked";
      sort_non-decreasing( unmarked_neighbours);

      eligible_slaves(v) := { ui  of list :   i <=   capacity(v) }; //the
capacity(v) nodes are picked
                                            //from the set "eligible_slaves(v)"
      for (all u which belongs to eligible_slaves(v))
          {
                slaves(v):={u} U slaves(v);
                message(v,u):= "become slave of v"; //message from v to u

           }
      for (all nodes w in the 2-hop neighborhood of v such that label(w)=
"unmarked" and w                      not in the set "eligible_slaves(v)")
         {
                message(v,w) := "slaves of v are slaves(v)";    //message from v to
w
           }
}


update_effective_degree(u)
  {
       current_unmarked_2-hop-neighbors := 2-hop-neighbors(u);
       for (all nodes v in the 2-hop neighborhood)
          {
              if (message(u,v)= "slaves of v are slaves(v)")
                 {
```

```
                  current_unmarked_2-hop-neighbors :=
current_unmarked_2-hop-neighbors -
                         ({v}U slaves(v)) ;  //update by removing the marked nodes
                  }
           }
        effective_degree(u) := number of nodes in current_unmarked_2-hop-neighbors;
     }

sort_non-decreasing( )
//sorts in non-decreasing order of effective degrees and if there are more than one
node with //the same effective degree sorst them by increasing order of their
device id's
   {
      }

connect_local(v)
//phase 2 connects the trees obtained in the first phase locally
{
     treeid_set(v):= {u : u in 2-hop_neighbours and treeid(u)=treeid(v)}; //keeps
track of                                                                //the nodes
v is connected to in it 2-hop neighborhood
       for (all u in 2-hop_neighbours such that label(u)= "Master" and treeid(u) is
not in                                      treeid_set(v))
              {
                   try_Master-Slave-Master_1(u , v);//case where Masters are
unsaturated
               }
       for (all u in 2-hop_neighbours such that label(u)= "Master" and treeid(u) is
not in                                      treeid_set(v))
                {
                      try_Master-Master(u , v );
                  }
        for (all u in 2-hop_neighbours such that label(u)= "Master" and treeid(u)
is not in                                      treeid_set(v))
                {
                        try_Master-Slave-Master_2(u , v); //case where Master is
saturated                                 }
         for (all u in 2-hop_neighbours such that label(u)= "Master" and treeid(u)
is not in                                      treeid_set(v))
                {
                          try_Master-Slave-Slave-Master(u, v);
                 }

     }//end connect_local

try_Master-Slave-Master_1(u, v)
//tries Master-Slave-Master connection
   {
     }//end

try_Master-Master(u, v)
//tries Master-Master connection
  {
    } //end

try_Master-Slave-Master_2(u, v)// saturated case
//tries Master-Slave-Master where the Slave can't be shared because of saturation
of the
// the other Master
 {
   } //end

try_Master-Slave-Slave-Master(Ti , Tj)
//tries Master-Slave-Slave-Master connectivity breaks the for relabelling a slave
to master by //using lowest device_id
 {
     } //end
```