AN ALGORITHM SUITE FOR MINIMIZING THE NUMBER OF HOPS BETWEEN COMMUNICATION PEERS IN A BLUETOOTH SCATTERNET

Csaba Kiss Kalló, Carla-Fabiana Chiasserini,
Roberto Battiti and Marco Ajmone Marsan

November 2004

Technical Report # DIT-04-101

An Algorithm Suite for Minimizing the Number of Hops between Communication Peers in a Bluetooth Scatternet

Csaba Kiss Kalló
Roberto Battiti
Università di Trento
Dipartimento di Informatica e Telecomunicazioni
{*kkcsaba,battiti*}*@dit.unitn.it*

Carla-Fabiana Chiasserini
Marco Ajmone Marsan
Politecnico di Torino
Dipartimento di Elettronica
{*chiasserini,ajmone*}*@polito.it*

Abstract

Mobility, and the fact that nodes may change their communication peers in time, generate permanently changing traffic flows in a Bluetooth scatternet. Thus, forming an optimal scatternet for a given traffic pattern may be not enough, rather a scatternet that best supports traffic flows as they vary in time is required.

In this article we propose an algorithm suite that enables us to modify the nodes' links and roles. Periodically executing these algorithms helps maintaining the distance (measured in hops weighted with the corresponding traffic intensity) between every source-destination pair at a minimum. This will allow for higher network throughput, lower packet delivery delay and nodes' energy consumption, and reduced communication overhead.

1 Introduction

Even though the Bluetooth wireless technology originally was aimed to replace cables, its suitability for a wide range of applications was rapidly revealed. A main feature that opened new horizons for this technology is its support for seamless networking with different types of devices.

The Bluetooth Specification 1.2 [1] defines how to organize Bluetooth-enabled devices in a network with at most 8 nodes, called *piconet*. Further, it introduces the *scatter mode* for supporting inter-piconet communication. In particular, it describes how time division should be implemented on nodes participating in multiple piconets, earlier discussed also in [12, 3]. However, the problem of efficient scatternet formation as well as many relevant optimization issues, discussed in numerous research papers [14, 11, 16, 10, 4, 6, 2, 5, 9, 15, 12, 3, 13], are still not addressed in this latest version of the specification.

Researchers revealed many factors influencing the operation of a scatternet. We believe that optimal values of the following parameters provide good performance [10]:

– *energy consumption* – lower is better
– *supported traffic* – higher is better
– *duration of scatternet formation* – shorter is better
– *link scheduling* – lower bridge degree is better.

Many proposals for scatternet formation are available in the literature by now, each making different trade-offs with the aforementioned parameters. One of the earliest works on Bluetooth scatternet structures [9] proposes three approaches: the first uses the *sniff* mode to enable up to 255 nodes connect to a single piconet while the other two build a tree-structured and a decentralized topology, respectively. Tree-shaped topologies were proposed also by others [16], however this structure has some important drawbacks [15]. In such a scatternet packets between different branches of the tree are routed up and down through the whole topology; if one node leaves the network, its whole subtree will loose connectivity; the root node is a bottleneck and all the other masters are also slaves in the same time, resulting in intra-piconet communication outage when the master is active in the other piconet.

In ring-shaped topologies, formed by the algorithms in [6], every node is a master, forming its own piconet. This is a major drawback because high inter-piconet interference can be generated by a small number of nodes. It is even worse that the algorithms in [6] require for all nodes to be in-range.

Bluetooth scatternets that provide good performance in most application environments are decentralized. Such scatternets are formed by the algorithms proposed in [14, 10], requiring for all nodes to be in communication range, but there also exist more general ones [15, 4] without this shortcoming. Decentralized scatternets impose less constraints on selecting the roles of the nodes and on how to set up links between them. In change, routing and link scheduling becomes more complicated than in the case of tree- and ring-structured topologies, with major influence on the overall performance. Next we outline several research efforts focused on compensating this shortcoming.

In the series of optimization works Miklós et.al. in [11] are the first to observe that setting up a high number of links in a scatternet can increase the overall throughput only to a certain limit. Above that limit the link scheduling overhead will lower the overall performance.

In [2] a technique for reducing the load of the most congested node is described and the approach is further improved in [5] with a technique that enables incremental formation of feasible scatternet topologies.

Raman et.al in [13] argue that power consumption and communication speed can be improved using application layer information at the lower layers. This is known as cross-layer optimization.

According to our best knowledge, neither the above works nor other optimization efforts study the possibilities of reducing the distance in hops between source and destination nodes in a scatternet. Thus, in our work we present a technique that fills in this gap.

Mobility and the fact that nodes may change their communication peers in time generate permanently changing traffic flows in a scatternet. Even if we had a scatternet formation algorithm that would produce optimal topologies, some time after the scatternet formation suboptimal traffic flows would show up, due to the changing source-destination associations between the nodes as well as because of mobility. Therefore, it is not enough to form an optimal scatternet, rather a scatternet that best supports traffic flows as they vary in time is required.

Our approach consists of an algorithm suite that enables the modification of nodes' links and roles and searches for such a scatternet configuration that minimizes the distance (in hops weighted with the traffic intensity) between all source-destination communication peers. This will allow for a higher network throughput, lower packet delivery delays and nodes energy consumption, and reduced communication overhead.

The remaining part of this work is organized as follows. In Section 2 the parameters of our problem are introduced and a formal description of the optimization problem is given. A centralized solution to the problem as well as a detailed presentation of the optimization algorithms are available in Section 3. We present our simulation results in Section 4. Finally, conclusions are drawn in Section 5.

2 Notations and Problem Formulation

Each piconet is composed of a master and up to 7 active slaves. A node participating in more than one piconet is called *bridge*. A node can be a master in only one piconet but it can be a slave in any number of piconets.

Let $\mathcal{N}$ be the *set of nodes* in the scatternet, $\mathcal{M}$ the *set of masters*, and $\mathcal{S}$ the *set of all slaves*. Notice that only pure masters are not elements of $\mathcal{S}$ and $\mathcal{S} \bigcap \mathcal{M} \neq \emptyset$ if there are master&bridge nodes in the scatternet. We denote with $\mathcal{C}$ the *set of traffic connections* in the scatternet.

$\mathcal{R} = \{r_{ij}^{sd}\}$, the *routing matrix*, stores the path between each source-destination pair $(s,d) \in \mathcal{C}$; we have

$$r_{ij}^{sd} = \begin{cases} 1 & \text{if connection } (s,d) \text{ is routed on arc } (i,j), \\ 0 & \text{otherwise.} \end{cases}$$

$\mathcal{T} = \{t_{sd}\}$ is the *traffic matrix* with $t_{sd} \in [0,1]$ indicating the intensity of the data flow on the connection (s,d). $t_{sd} = 0$ means that there is no traffic flow between the nodes s and d.

$\mathcal{H} = \{h_{sd}\}$, the *hop matrix*, contains the minimum number of hops between any connection $(s,d) \in \mathcal{C}$.

$\mathcal{P} = \{p_{ij}\}$ is the *radio proximity matrix* with

$$p_{ij} = \begin{cases} 1 & \text{if nodes } i \text{ and } j \text{ are in-range,} \\ 0 & \text{otherwise.} \end{cases}$$

The *link matrix* $\mathcal{L} = \{l_{ij}\}$ is defined as

$$l_{ij} = \begin{cases} 1 & \text{if } i \text{ is master of } j, \forall i,j \in \mathcal{N}; i \neq j \\ 0 & \text{otherwise.} \end{cases}$$

The link matrix indicates the master-slave connections in the scatternet. Link matrix properties are explained below and summarized in Table 1.

1) A *master* has on its row one entry equal to 1 for each of its slaves.

2) A *pure slave* has one entry equal to 1 on its column corresponding to its master.

3) A *slave&bridge* has on its column exactly one entry equal to 1 for each of its masters.

4) A *master&bridge* node has one entry equal to 1 for each of its slaves on its row and for each of its masters on its column.

5) A *free node* – node not belonging to any piconet – has all 0s on both its row and column.

Table 1: Link matrix properties based on nodes' role

Role of node k	**Property**
Master	$\sum_{j=1}^{N} l_{kj} \geq 1$
Pure slave	$\sum_{i=1}^{N} l_{ik} = 1$
Slave&bridge	$\sum_{i=1}^{N} l_{ik} \geq 2$
Master&bridge	$\sum_{j=1}^{N} l_{kj} \geq 1$ and $\sum_{i=1}^{N} l_{ik} \geq 1$
Free node	$\sum_{j=1}^{N} l_{kj} = 0$ and $\sum_{i=1}^{N} l_{ik} = 0$

We define function F as

$$F = \sum_{(s,d) \in \mathcal{C}} t_{sd} h_{sd}. \tag{1}$$

F is the sum of the number of hops weighted with the traffic intensity between all source-destination pairs in the scatternet.

Our objective is to solve the following optimization problem,

$$\mathcal{P}: \quad \min_{H} F \tag{2}$$

subject to the following constraints [2].

- A piconet must contain one master and up to 7 slaves.

$$\sum_{j=1}^{N} l_{kj} \leq 7, \forall k \in \mathcal{M} \tag{3}$$

- There can exist a master-slave relationship between two nodes if and only if they are in radio proximity of each other.

$$l_{ij} \leq p_{ij}, \forall i,j \in \mathcal{N} \tag{4}$$

- If i is master of j, then j cannot be master of i.

$$l_{ij} + l_{ji} \leq 1, \forall i, j \in \mathcal{N}; i \neq j \qquad (5)$$

- Traffic between source s and destination d can be routed through edge (i, j) only if i and j communicate, i.e. either i is assigned to j, or j is assigned to i.

$$r_{ij}^{sd} \leq l_{ij} + l_{ji} \; \forall \, (s, d) \in \mathcal{C}, \forall i, j \in \mathcal{N} \qquad (6)$$

- All traffic between two nodes is routed through a minimum length paths, with no loops. The selected path may not necessarily be the same in both directions, if more than one minimum length paths exist.

$$h_{sd} = h_{ds} \; \forall \, (s, d) \in \mathcal{C} \qquad (7)$$

$$\sum_{i,j \in \mathcal{N}} r_{ij}^{sd} = h_{sd} \; \forall \, (s, d) \in \mathcal{C} \qquad (8)$$

- Other constraints that are widely used for making routing possible should also be considered.

3 Problem Solution

To solve our problem we generate a connected and totally functional scatternet, as detailed in Section 3.2. Based on the processing power, we choose one of the masters, the so-called *optimizer*, to coordinate the optimization procedure. The optimizer collects relevant information about all nodes in the scatternet, such as the identity and role of the nodes and their neighbors, masters and communication peers, and feeds it into the optimization algorithm.

The optimizer uses a local search strategy based on a set of possible changes that can be made on the topology, the so-called *moves*. Moves may lead to piconet formation or merging, or just make slaves move from one piconet to another one. In particular, moves targeting slaves typically increase the number of piconets in the network, while moves targeting masters may merge piconets. Thus, in our optimization procedure we try to reduce the number of hops by first moving slaves and then moving masters. As an example, consider the scatternet shown in Figure 1. If there is a high traffic flow between slaves 8 and 12, then the scatternet can be optimized by removing node 8 from master 2 and assigning it to master 1.

For each move, the optimizer calculates the new value that function F would assume. If the move decreases F, it is stored, otherwise it is dropped. At the end of the optimization the most convenient scatternet configuration, stored during the search, is set.

The optimization algorithm can be executed periodically. We call the time between two executions *optimization period*. Implicit feedback from the scatternet, like the number of recently arrived/left nodes and the gain of previous executions, can be used for dynamically determining the optimization period. Thus, in a scatternet with dynamically changing traffic

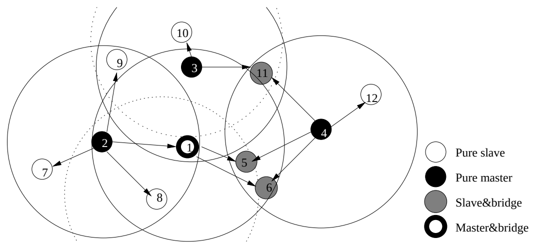

Figure 1: Scatternet snapshot

connections and high node mobility the optimization period will be short, while in quasi-static environments the optimization will be rarely executed. Note, however, that in this paper we only study single executions of the optimization procedure. The behavior of periodically optimized scatternets is subject for future study.

3.1 Move Types

A *move* is a set of modifications on the master-slave relationship between two nodes in the network. Such modifications are made by link creation, deletion and/or by master-slave role exchange. If, due to these modifications, some nodes get disconnected, the operations necessary to reconnect them to the scatternet are considered as parts of the same move.

We identify four kinds of possible moves:

Slave to Slave (SS) – a slave connects to a different master or establishes a new piconet with a node which then exchanges its role from slave to master. Since moving bridge nodes influences considerably the routing scheme of the scatternet we are not moving bridge nodes but only pure slaves.

Example (based on Figure 1): we want to remove slave 8 from master 2 and assign it to master 1. To this end we set $l_{28} = 0$ and $l_{18} = 1$; i.e., first we cancel the link between master 2 and slave 8, then we create the link between master 1 and slave 8.

Slave to Master (SM) – a slave creates a new piconet by paging another node.

Example (based on Figure 1): we want to remove slave 8 from master 2, change its role into master and assign slave 5 to it. To this end we set $l_{28} = 0$ then $l_{85} = 1$. This means that we cancel the link between master 2 and slave 8, then we create the link between node 8 and slave 5.

Master to Slave (MS) – a master becomes a pure slave. Such a move is possible only if the slaves of the moving master (i.e. the node giving up its role of master) can be assigned to other nodes in the scatternet. The optimizer takes the abandoned slaves (i.e the slaves of the moving master) one-by-one and assigns them to an already existing master, using SS and SM moves.

Master to Master (MM) – merging two piconets: a master overtakes all the slaves of another master. Such a move can take place when any node in the two piconets is in the range of the persisting master (i.e. the node maintaining its role of master after the move) and the total number of nodes in the two piconets is not greater then 8. This move can be done by removing from $\mathcal{L}$ all the 1s from the row of the master that is about to become a slave and adding them to the corresponding positions in the row of the persisting master. The old master should be connected to the persisting one through an additional operation. For instance, if node i gives up its role of master and joins the piconet of master j, the additional operation would be $l_{ji} = 1$.

3.2 Scatternet Generation

For our simulations we generate an initial scatternet based on the algorithm proposed by Basagni and Petrioli in [4]. We generate N nodes randomly positioned in the network area. The size of the network area is chosen so that the radio topology results to be connected with high probability.

For each node we generate a random weight, that indicates the willingness of the device for assuming special roles (master or bridge) in the scatternet. After all nodes and weights have been generated, we select the *init masters* (the nodes with the biggest weight in their radio proximity [4]). Slaves get connected to one of the init masters in their radio proximity (at less then 10 meters). If a slave is located in the radio proximity of more then one init masters, a random choice is made. We believe that parking and unparking slaves (like in [9]) when scatternet formation is possible, is an inconvenient operation since it does not give the possibility to all the devices to be active at the same time and requires extra operations from the master. Therefore, in the case where more than 7 nodes are in the radio proximity of an init master, only 7 of them become members of its piconet while the others will organize themselves in one or more other piconets. This happens in a subsequent step of the scatternet formation when the masters and their salves are selected in the same manner as in the case of init masters but taking into account also the nodes that have gotten a role already.

Once all nodes have been assigned to a piconet, the algorithm proceeds with selecting firstly the one-hop bridge nodes between neighboring masters. Only one bridge is placed between two masters and, if possible, a bridge gets connected to two masters only. If there are more then one potential bridge nodes between two masters, we select the one that has the smaller number of masters (possibly 1) and whose physical distance from the two masters is smaller. By doing so we reduce the bridge scheduling overhead and select bridges that receive the strongest signal from the masters.

As the last step of the scatternet formation, we connect all of the two-hop pure masters if they were not already connected through the scatternet. In [4] all two-hop masters are connected. Although this is necessary for building a connected scatternet, it introduces redundant links between the nodes that consume device resources. Since our work concentrates mainly on scatternet optimization and not formation, we assume that some time after the network formation nodes abandon these redundant links and maintain the shortest path only to those neighboring nodes that are at least 6 hops away from them. (Notice that the minimum distance between the pure slaves of two two-hop pure masters is 5, hence the value of 5 above.) At this time, a master node may also become a

```
OPTIMIZER
$\mathcal{L}_{init} \leftarrow \mathcal{L}$
$F \leftarrow$ ActualNrHops()
slavelist ← list of slaves (for SX module)
masterlist ← list of masters (for MS and MM)
for k ← 1 to nr_diversifications do
    call SX or MS or MM optimization module
    if $F >$ ActualNrHops() then
        $F \leftarrow$ ActualNrHops()
        $\mathcal{L}_{opt} \leftarrow \mathcal{L}$
    $\mathcal{L} \leftarrow \mathcal{L}_{init}$
    HUpdate()
    shuffle slavelist (for SX module)
    shuffle masterlist (for MS and MM module)
$\mathcal{L} \leftarrow \mathcal{L}_{opt}$
HUpdate()
end OPTIMIZER

SX optimization module
for each slave i in slavelist do
    ssbestmove ← SS(slavelist[i])
    smbestmove ← SM(slavelist[i])
    if ssbestmove.localF < smbestmove.localF then
        SSmover(slavelist[i], ssbestmove.bestid)
    else SMmover(slavelist[i], smbestmove.bestid)

MS optimization module
for each master i in masterlist do
    moves ← MS(masterlist[i])
    if moves is not empty then
        MSmover(moves)

MM optimization module
for each master i in masterlist do
    mmbestmove ← MM(masterlist[i])
    if mmbestmove.localF < F then
        MMmover(masterlist[i], mmbestmove.bestid)
```

Figure 2: Pseudo code of the optimizer

bridge.

If physically possible, at the end of the algorithm we obtain a connected scatternet. The simulation environment generation ends with selecting a predefined number of traffic connections between random source and destination nodes.

3.3 The Optimization Procedure

The *optimization procedure* is the core of our work. It coordinates the various kind of modifications performed on the scatternet topology, aimed to reduce the number of hops between communicating nodes. The optimization procedure should run on a selected node, possibly with strong computational power, capable of collecting all the necessary data about participating nodes.

The optimization algorithm is presented in Figure 2. It consists of a main body from which three different optimization modules, namely SS, SM (either one denoted by SX in Figure

2), MS and MM, can be called. At the beginning of the main body several initializations are performed. First the initial state of the link matrix $\mathcal{L}$ is saved (line 2). In line 3 with the function *ActualNrHops()* we retrieve the number of weighted hops between all the source-destination pairs and assign it to F, our function to be minimized. In lines 4-5 all pure slaves and masters are selected and put in *slavelist* and *masterlist*, used later by the *slave optimization* (i.e. when SS or SM moves are performed) and *master optimization* (i.e. when MS or MM moves are performed) modules, respectively.

In line 6 we cycle through the optimization procedure *nr_diversifications* times. Inside the cycle one of our three optimization modules is executed. Each of these modules performs a local search [8] using the corresponding moves for finding a scatternet configuration that reduces the value of F. The operation of the optimization modules as well as the role of the aforementioned cycle is explained in details later in this section.

After the optimization module has been selected and executed, we obtain a scatternet configuration with an F value that we confront with the initial (or previously stored) value of F. If this new value is smaller than the initial one, we have a more optimal scatternet configuration. Therefore, we save this configuration in $\mathcal{L}_{opt}$ and update the value of F (lines 8-10). Before the next iteration of the **for** loop we set $\mathcal{L}$ to its initial value (line 11). We also update the hop matrix $\mathcal{H}$ since the moves made during the optimization modified it. This is performed by the *HUpdate()* function based on $\mathcal{L}$. Finally, the nodes in *slavelist* or *masterlist* are reordered (i.e. a *diversification* is done) and the algorithm proceeds with the next iteration of the **for** loop.

After the **for** loop $\mathcal{L}$ is set to the best configuration found, stored in $\mathcal{L}_{opt}$, and the hop matrix is updated.

Our optimization modules operate as explained next.

The *SX optimization module* evaluates one-by-one each node from *slavelist* (line 19). The SS and SM algorithms are called one after the other in this module, since they both provide an alternative for moving the same slave. As we will see, this is not the case when moving masters.

Although not explicitly mentioned in the pseudo code, during the search slaves can change their role to master. Therefore, as the for loop in line 19 cycles through the slaves in *slavelist*, each node should be checked whether it is still a slave. All the pure slaves are then evaluated using a series of SS and SM moves (lines 20-21) for possible reductions of F. The functions SS() and SM(), implementing this series of moves, get as input the identifier of a slave from *slavelist*. They both return a pair (*bestid*, *localF*), containing the identifier (*bestid*) of the node to which the slave should be moved for obtaining the biggest reduction of F, held by *localF*. These pairs are stored in the variables *ssbestmove* and *smbestmove* for SS() and SM(), respectively. The return value of these two functions can be interpreted as "*the best* F *value of* sxbestmove.nrhops *for this slave can be obtained by connecting it to node* sxbestmove.id". If no move of the slave reduces the value of F, the identifier of its current master is returned accompanied by F's initial value. A detailed description of the SS and SM algorithms can be found in Section 3.4.

The optimization algorithm continues by evaluating the outcome of the SS() and SM() algorithms (lines 22-24). Depending on the hop reduction provided by SS() and SM() the *SSmover()* or the *SMmover()* function is executed. *SSmover()* and *SMmover()* act on the link matrix $\mathcal{L}$, therefore after their execution the hop matrix should be updated. Indeed, this is what the *HUpdate()* function is used for in line 12.

A secondary task of *SSmover()* and *SMmover()* is to update the variable (not shown in the pseudo code) storing the role of each node. We prefer to use such a variable instead of recalculating every time the roles from the link matrix, in order to reduce the execution time.

The two master optimization modules operate in a similar manner, but they are somewhat simpler since the MS and MM moves have separate, dedicated modules. This is required since they do not provide alternative moves for the same master.

The *MS optimization module*, however, differs on a further point from all other modules since the MS() algorithm returns a list of already executed moves instead of one single move that is to be executed later. This is because for transforming a master into slave we need to move all its slaves to some other master (as detailed in Section 3.5). After this series of moves has been executed inside MS() it would be a waste of CPU time resetting them at the end of MS() then setting them again in the frame of the *MSmover()*. Thus, the only task of the *MSmover()* function is to update the variable (not shown) storing the roles of the nodes in the *moves* list. The impact of this choice on the system's modularity is minor.

Thus, we call the MS() algorithm for each master in the master list. If it returns a non-empty list of moves (meaning that a move has been performed) that reduces F, the *MMmover()* function is called for updating the roles of the moved nodes.

In the *MM optimization module* (line 30) the MM() algorithm shows similar behavior to that of the SS() and SM() algorithms. It returns a node identifier (*mmbestmove.bestid*) and the number of weighted hops (*mmbestmove.localF*) that the MM move with that identifier would produce. If *mmbestmove.localF* is less then the value of F, it means that a more optimal scatternet configuration has been found.

Returning to the the for cycle in line 6, let us take a closer look to why is that necessary to repeat the optimization with different ordering of the slaves and masters when we already found a configuration with a lower F. The reason is that even if this configuration has lower number of hops between all its source-destination pairs than the original one, it is high the probability that other, better solutions exist. For example, in case of slave optimizations better solutions can exist due to the fact that during the execution of the algorithm slaves can change their masters. Since pure slaves do not forward data, SS and SM moves directly influence only the number of hops between the moving slave and its destinations. Pure slaves take part only as target nodes in any communication. However, indirectly they can affect also communication links where they do not play neither the role of source nor that of destination. In particular, this is possible when a slave moves to another master and another slave could have used it in a latter iteration to shorten the path to its own destinations.

On the other hand, if slaves A and B communicate, they will mutually try to move closer to each other. The number of hops that can be cut off from the route between A and B is not always symmetric. In many cases if we make the hop reduction from the point of view of A, it is not possible to carry out also the reductions from B, because of the route modifications.

We can conclude that the order in which slaves are analyzed by the optimizer is important. Therefore, it is reasonable to suppose that repeating the same procedure (lines 6-14) but with a different ordering of the slaves in the *slavelist* could produce a better solution. This is what we do in our algorithm. We reset $\mathcal{L}$ to its saved initial value (line 11), update the hop matrix $\mathcal{H}$, generate a random ordering of the slaves (line 13) and re-execute the search.

The reason why we generate randomly the order of slaves is that we want to examine only a few possibilities from the huge search space. Recalculating the minimum paths between all nodes of the scatternet would be extremely time-consuming. For example, in a scatternet with only 30 nodes typically the number of pure slaves will be 12. All the possible permutations of 12 slaves are almost 480 million. Trying all these permutations requires an unacceptably big amount of time. Therefore, we prefer to repeat the search for only several times and choose the best solution found. The importance of the cycle from line 6 is to specify the number of, so-called, *diversifications* of the search trajectory [8]. This means that we will randomly reorder the slaves in $slavelist$ $nr\_diversifications$ of times. The local minima found in a diversification is compared against the former optima stored in F. If the new local minima is smaller, F will be assigned this new value and the link matrix will also be saved in $\mathcal{L}_{opt}$ (line 8-10).

The situation is similar also with masters. The central idea is that at every iteration of the optimization algorithm we have to chose only one move from a set of mutually excluding moves. Therefore choosing one move from the set we may eliminate the possibility of performing moves that could produce a lower F value. Thus, repeating the same search trying different moves from the same set of mutually excluding moves raises the probability of finding a more optimal configuration. This is the reason why we repeat the search $nr\_diversifications$ times.

The optimization algorithm can execute the optimization modules sequentially, combining them in different ways. For instance, if we perform the SX, MS and MM optimization modules, we obtain an optimization algorithm that we refer to as SX_MS_MM. The SX module can also be replaced by an SS or SM module giving the so-called SS_MS_MM and SM_MS_MM optimizations, respectively.

```
1.  SS ($id$)
2.  $localF \leftarrow ActualNrHops()$
3.  $bestid, m \leftarrow MyMaster(id)$
4.  for each neighbor $i$ of slave $id$ do
5.      if $h[id][i] > 2$ and
6.          ($i$ is not master or (for SM: NA)
7.          ($i$ is master and (for SM: NA)
8.          $NrSlaves(i) < 7$ )) then (for SM: NA)
9.          $l[m][id] \leftarrow 0$
10.         $l[i][id] \leftarrow 1$ (for SM: $l[id][i] \leftarrow 1$)
11.         if $localF > ActualNrHops()$ then
12.             $localF \leftarrow ActualNrHops()$
13.             $bestid \leftarrow i$
14.         $l[m][id] \leftarrow 1$
15.         $l[i][id] \leftarrow 0$ (for SM: $l[id][i] \leftarrow 0$)
16. return ($bestid$, $localF$)
17. end SS
```

Figure 3: Pseudo code of the SS and SM algorithms

Regardless of the optimization modules used, after executing the optimization algorithm a certain number of times, we find a scatternet configuration with fewer hops connecting traffic sources to destinations. Our algorithm can guarantee a global optimal configuration only if each optimization module is called for all possible permutations of nodes in the corresponding *slavelist* and *masterlist*. This would take an unacceptable long period of time. Therefore, a good trade-off between the number of diversifications and execution time should be found for achieving acceptable performance in real environments.

3.4 Reduction of hops by moving slaves

As already mentioned in the previous sections, slave optimization aims at finding the best possible SS or SM move for reducing the number of weighted hops between a slave and all its communication peers. During the optimization slaves are assigned one-by-one to each node in their radio proximity and the produced reduction of hops is evaluated. After all the neighbors of a slave were checked, the move that produced the biggest hop reduction will be selected. Next the slave optimization algorithms, SS() and SM(), are detailed. Since these two algorithms are similar we discuss them together, outlining the differences where it is the case.

Slave optimization starts by several initializations (Figure 3). *localF* stores the best solution found until the current instant for the current ordering of the slaves (line 2). With function *MyMaster()* the master of the evaluated slave is retrieved and stored in *m* (line 3). The same initial value is assigned also to the variable *bestid*, which keeps the identifier of the best potential target node.

The cycle starting at line 4 evaluates one-by-one all neighbors *i* of slave *id*. Not all neighbors will be used for the optimization. In the lines 5-8 we impose a series of requirements for the neighbors that will be evaluated. Thus, the condition in line 5 filters out all neighbors that are in the same piconet in which the evaluated slave, *id*, is. This way we avoid creating piconets inside another piconet. The conditions in lines 6-8 are used only in the SS algorithm to avoid moving new slaves to a master that already has 7 of them. Therefore, only those neighbors are evaluated, which are either pure slaves, bridges or masters having less then 7 slaves. These constraints are not applicable (NA) for SM moves where the number of slaves that a master has is not important, since after the move the slave *id* will create its own piconet.

Before the execution of an SS move the slave *id* is connected to its master *m* while the target neighbor, *i*, is not connected to that same master (recall condition in line 5). Therefore, the corresponding positions in the link matrix $\mathcal{L}$ are $l[m][id] = 1$ and $l[i][id] = 0$. The SS and SM moves alter these settings as shown in lines 9-10. In case of an SS move the neighbor *i* pages slave *id*, while for SM moves slave *id* becomes a master and pages neighbor *i*. Once the move has been executed, its effect on the number of hops is evaluated and, if there is any improvement, the newly calculated number of hops and the identifier of the target node, will be stored in *localF* and *bestid*, respectively (lines 11–13). After the evaluation the original links are restored, differentiating again between SS and SM moves (lines 14-15).

Once the evaluation of the whole neighborhood was terminated, the pair (*bestid, localF*), containing the identifier of the target node that produced the biggest hop reduction (*localF*), is returned (line 16).

3.5 Reduction of hops by moving masters

The MS Algorithm

After moving slaves, we try to reduce the number of hops between source and destination nodes by moving also masters.

The MS optimization algorithm (Figure 4) gets as input the identifier of a master, *id*, and returns the list of moves that the algorithm was able to identify for reducing the value of F (see (1)), after a predefined number of iterations.

The MS algorithm has three main parts. The first part (line 3-16) concerns the reassignment of *id*'s slaves to new masters while the second part (line 17-26) finds the new master of *id*, based on the number of hops between all source-destination pairs in the scatternet. Finally, the last 9 lines re-execute the series of moves that produce the highest hop reduction and return the *movelist* to the optimizer. Next we present the algorithm in details.

In line 2 two variables are initialized: F contains the lowest number of weighted hops found up to the current moment while *initnrhops* is the value of F at the beginning of the algorithm. After the initializations a cycle is started (line 3) for reassigning to new masters all the slaves of *id*, i.e. pure slaves, bridges and master&bridges. In this cycle the slaves of *id* are analyzed one-by-one. The algorithm handles differently the pure slaves on one hand and bridges and master&bridges on the other hand.

```
1.  MS(id)
2.  F, initnrhops ← ActualNrHops()
3.  for each slave j of master id do
4.      if j is a pure slave then
5.          sugmast ← SuggestMaster(j,id)
6.          if sugmast ≠ id then
7.              l[sugmast][j] ← 1
8.              l[id][j] ← 0
9.              append move to movelist
10.         else
11.             Restore(movelist,id)
12.             clear movelist
13.             return movelist
14.     if j is a bridge or master&bridge then
15.         l[id][j] ← 0
16.         append move to movelist
17. for each potential target master i do
18.     if CommonMaster(i,id) = 0 then l[i][id] ← 1
19.     append move to movelist
20.     HUpdate()
21.     if CheckConnectivity(movelist) ≠ 0 and
22.         F > ActualNrHops() then
23.         F ← ActualNrHops()
24.         bestmaster ← i
25.     if CommonMaster(i,id) = 0 then l[i][id] ← 0
26.     remove last move from movelist
27. if initnrhops > F then
28.     if CommonMaster(bestmaster,id) = 0 then
29.         l[bestmaster][id] ← 1
30.     append move to movelist
31. else
32.     Restore(movelist, id)
33.     clear movelist
34. HUpdate()
35. return movelist
```

Figure 4: Pseudo code of the MS algorithm

The *SuggestMaster()* function (line 5) finds for each pure slave a master in the slave's radio proximity, that can overtake it from *id*. Further, it finds the master that reduces the most the number of weighted hops between the slave and all its destinations. If no such a master is found, the function returns the identifier of the slave's old master. However, if the return value, stored in the variable *sugmast*, is a master's identifier, other than *id*, then the slave is moved to this master (lines 7-8) and the move is stored in a *movelist*. The elements of *movelist* are made of node pairs indicating which node has been moved to which master.

If there is one single slave that cannot be moved to another master, then it is not possible to perform any optimization with the master *id*, thus the algorithm returns the control to the optimizer (line 13). It would be possible to transform a slave into master, however we do not want to increase the number of masters in the scatternet while performing the MS algorithm. Moreover, it is among the tasks of this algorithm to compensate the increase in the number of masters produced by SS and SM moves. Therefore, the MS algorithm restores the link matrix $\mathcal{L}$ (line 11) based on the *movelist* and returns an empty *movelist* to the optimizer, signifying that no MS move can be performed with the master *id*.

If the role of *id*'s slave is bridge or master&bridge (line 14), the situation is easier because we do not have to look for a master that can overtake it, since these kind of nodes already have at least one other master. Therefore, all the algorithm does is to remove the link of the slave from *id* and store this move in *movelist*. Since bridge and bridge&master nodes can always be moved, we do not have to perform any related verifications.

If the algorithm did not return the control to the optimizer in line 13, in line 17 we proceed with its second phase. Hence, we can assume that every slave of *id* has successfully been moved to some other master. Therefore, we start searching for a master that could accommodate also *id* in its piconet. Usually in big and dense scatternets there will be more then one such masters, however we want to find the one reducing the most the value of F. The second phase (line 17-26) concentrates on this issue.

We take one-by-one each potential target master that could accommodate *id*. In line 18, using the *CommonMaster()* function, we check whether *id* and the evaluated master i have a master in common. We do so in order to avoid creating "triangles", i.e. piconets in another piconet. A new link between i and *id* is created only if they do not already have a master in common. Anyway it be, in *movelist* will be recorded (line 19) that node *id* has been moved to node i, even if no modification on the link matrix has been performed. This operation is necessary for the function *CheckConnectivity()* that verifies whether the connectivity of the scatternet was damaged by the moves. If it finds a path between each pair of nodes in *movelist*, it can be stated that the moves have not damaged the connectivity, given that the scatternet was connected at the beginning of the algorithm. Notice that the function *HUpdate()* is called before *CheckConnectivity()*, to recalculate the shortest paths between every node of the scatternet. *HUpdate()* takes the necessary input data from the link matrix $\mathcal{L}$, uses Floyd's algorithm for solving the *all-shortest path problem* [7], and stores the result in $\mathcal{H}$ (recall from Section 2).

Beside the connectivity check the hop reduction obtained by moving *id* to i is also evaluated (line 22). The *ActualNrHops()* function returns the current number of weighted hops in the scatternet, which is compared to F. If the current number of weighted hops is less then the one stored in F, the value of F is updated and so is *besttagetmaster*, the variable containing the identity of the master that produced the lowest number of weighted hops so far.

For any outcome of the connectivity check in line 21, the move of *id* to master i is undone (line 25-26) and the algorithm continues with a subsequent iteration (line 17), evaluating the following potential new master of *id*.

The second phase of the algorithms terminates when the evaluation of all the masters is finished. At this point, in F we have the lowest number of weighted hops found during the search, while *bestmaster* stores the identity of the master that produced this F. Note that all moves performed with the slaves of *id* are stored in *movelist*, but there is no move for *id* itself yet (see lines 25-26). In the third phase, if F is smaller than the initial number of weighted hops (line 27), the best move found is performed again and this move is appended to *movelist* as well (lines 28-30).

If no better solution was found by moving master *id*, all moves performed with *id*'s slaves are undone (lines 32-33).

A final update of $\mathcal{H}$ is done in line 34, then the algorithm terminates by returning *movelist* to the optimizer. If a more optimal position for *id* has been found, *movelist* will contain the corresponding sequence of moves that lead to that configuration. Otherwise an empty list is returned.

The MM Algorithm

Although the MM moves target masters, the structure of the MM algorithm is similar to the SS and SM ones and not MS as one would expect. It basically consists of checking every neighboring master of a master *id*, saving their links to their slaves, checking the hop reduction and resetting the original links. The MM algorithm is presented in details next.

At the beginning of the algorithm we initialize several variables. Thus, *localF* is set to the current number of weighted hops between the communication peers; *tm* will hold the total number of masters in the scatternet using the *TotNrMasters()* function while *nrs* is set to the number of *id*'s slaves. *bestid*, the variable storing the identifier of the master that could overtake all the nodes in the piconet of *id*, is initialized to *id*. Finally, in line 6 all the links that connect *id* to its slaves are stored.

The main part of the algorithm starts in line 7 with a loop that takes one-by-one every neighboring master i (i.e. masters in radio proximity) of *id* with the goal of piconet fusion. These masters are further filtered in line 8. First, it is verified whether the total number of slaves in the piconets of *id* and i is less then 6. This condition is necessary to observe the specification requirement of 8 nodes in the new piconet. Second, it is also checked (using the *Collocated()* function) whether all the slaves of *id* are in the range of master i. If such a master is identified, the MM move described in Section 3.1 is executed, as follows. The loop in lines 10-12 moves the slaves of *id* to i then *id* itself is connected to i (lines 13–14), too. (The assignment in line 13 is useful only when a link exists between *id* and i.) After the move the hop matrix $\mathcal{H}$ is updated to enable hop counting (line 15), and the move is checked (lines 16-19) in the following manner. If the number of weighted hops in the scatternet is less then it was before the move or it is the same but the number of masters (i.e. piconets) decreased then the identity, i, of the target master is stored, as well as the number of weighted hops and piconets. In any case, after this verification the original state of the link matrix $\mathcal{L}$ is restored (line 22) and the search for another potential target master is continued.

```
MM(id)
$localF \leftarrow ActualNrHops()$
$tm \leftarrow TotNrMasters()$
$nrs \leftarrow NrSlaves(id)$
$bestid \leftarrow id$
store all slave links of id
for each neighboring master i of id do
    if $NrSlaves(i) + nrs \leq 6$ and $Collocated(i, id)$
        store all slave links of i
        for each slave j of id do
            $l[id][j] \leftarrow 0$
            $l[i][j] \leftarrow 1$
        $l[id][i] \leftarrow 0$
        $l[i][id] \leftarrow 1$
        HUpdate()
        if $localF > ActualNrHops()$ or
            ($localF = ActualNrHops()$ and
            $tm > TotNrMasters()$)
            $bestid \leftarrow i$
            $localF \leftarrow ActualNrHops()$
            $tm \leftarrow TotNrMasters()$
        restore all links of id and i
return (bestid, localF)
```

Figure 5: Pseudo code of the MM algorithm

After checking all neighboring masters the identifier of the target master that could overtake all nodes in *id*'s piconet, reducing the most the number of weighted hops (*localF*), is returned.

4 Numerical Results

To evaluate the performance of our algorithms, we implemented a Bluetooth scatternet simulator in C++. Since the algorithms operate on the scatternet topology, in our simulator we mainly considered topology-related aspects like physical links, nodes' roles, radio proximity and multihop connections, omitting protocol stack details and the problem of connection establishment delays.

We tested the optimizer by generating 50 scatternets, of 100 nodes each, over an area of 66x66 m^2. We set the nodes' radio range to 10 m and $nr\_diversifications = 1000$. We randomly generated 100 source-destination bidirectional (200 unidirectional) communication pairs (elements of $\mathcal{C}$). The traffic intensity (t_{sd}) on these connections is 0.1, 0.25 and 0.5 for 50%, 30% and 20% of (s, d) connections, respectively. All simulations were run on a Linux PC with a 1.7 Ghz CPU and 256 Mb RAM.

We performed experiments combining the SS, SM, MS and MM optimizations in many different ways. In this section we present results derived from the most illustrative sample optimizations. In particular, we consider three groups of experiments referring to the case where the optimizer is called one, two and three times, respectively, during the same simulation. Each call to the optimizer corresponds to a different optimization module. Figures 6-8 show the evolution of function F during these experiments, against the number of diversifications.

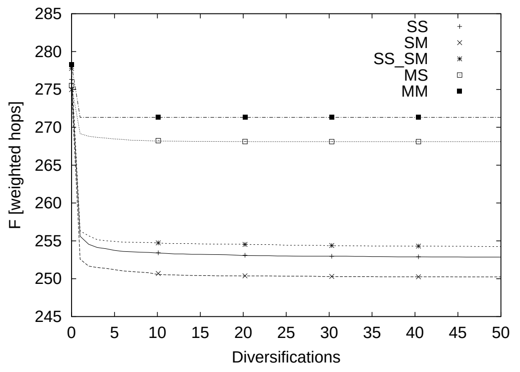

Figure 6: Optimization with simple moves

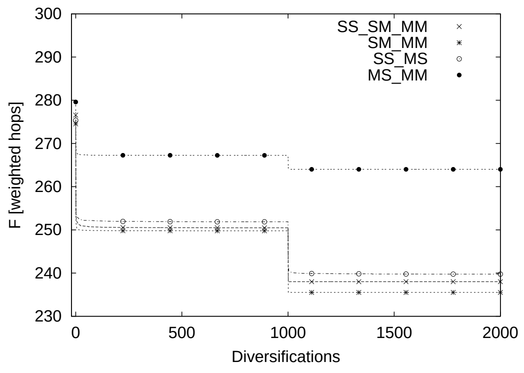

Figure 7: Optimization in two phases

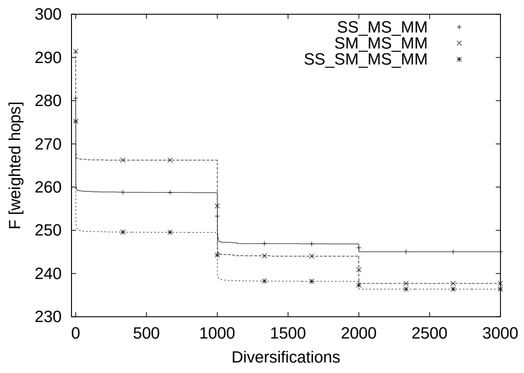

Figure 8: Optimization in three phases

Then, for each group of experiments, the one giving the best performance is evaluated in greater details in Tables 2-4. In each table, the following metrics are presented. The *Slaves*, *Slave&bridges* and *Total master* parameters report the number of pure slaves, slave&bridges and total number of pure masters and master&bridges, respectively. The parameter *Links* represents the number of links in the scatternet. The *Weighted hops* row shows the overall optimization achieved after each module of the optimization is terminated, while the *Hops* row presents the corresponding hop count. The distance weighted (i.e multiplied) by the traffic intensity (t_{sd}) is expressed in *weighted hops* ($[wh]$), and the distance is measured in *hops* ($[h]$). The rows referring to the SM, MS and/or MM

Table 2: Optimization with SM moves

EXPERIMENT #1	**Before**	**After**	**Diff.**	**%**
Pure slaves	32.92	14.32	-18.60	-56.50
Slave&bridges	32.48	33.42	0.94	2.89
Total Masters	34.60	52.26	17.66	51.04
Links	121.32	121.32	0.00	0.00
SM optimization [wh]	274.97	250.22	-24.75	-9.08
Hops [h]	1222.08	1121.68	-100.40	-8.26
Weighted hops [wh]	274.97	250.22	-24.75	-9.08

Table 3: Optimization with SM_MM moves

EXPERIMENT #2	**Before**	**After**	**Diff.**	**%**
Slaves	33.34	21.20	-12.14	-36.41
Slave&bridges	32.34	38.02	5.68	17.56
Total Masters	34.32	40.78	6.46	18.82
Links	121.04	131.12	10.08	8.33
SM optimization [wh]	274.48	249.81	-24.67	-9.09
MM optimization [wh]	249.81	234.75	-15.06	-6.14
Hops [h]	1221.88	1054.92	-166.96	-13.76
Weighted hops [wh]	274.48	234.75	-39.73	-14.64

Table 4: Optimization with SM_MS_MM moves

EXPERIMENT #3	**Before**	**After**	**Diff.**	**%**
Pure slaves	33.02	24.82	-8.20	-24.83
Slave&bridges	32.84	35.68	2.84	8.65
Total masters	34.14	39.50	5.36	15.70
Links	121.42	125.26	3.84	3.16
SM optimization [wh]	278.74	254.47	-24.27	-8.86
MS optimization [wh]	254.47	241.36	-13.11	-4.48
MM optimization [wh]	241.36	235.06	-6.30	-2.67
Hops [h]	1228.24	1047.88	-180.36	-14.34
Weighted hops [wh]	278.74	235.06	-43.68	-15.26

optimizations in the *Before* and *After* columns contain values of the weighted distance in the scatternet configuration exactly before and after that specific phase of the optimization. All other values in these two columns refer to the beginning and the end of the entire optimization procedure. The last two columns indicate the differences between the values in the *Before* and *After* columns, expressed in the appropriate unit (*Diff.*) as well as in percents (%).

First, let us consider Figure 6 and Table 2. In Figure6, for the sake of readability we present only the first 50 diversifications out of 1000 (the curves remain flat from that point onward). The plot shows that the SM optimization produces the greatest weighted hop reduction among the simple moves. Master moves (MS and MM), instead, produce small hop reduction. This confirms that in our initial scatternet masters were selected with care. Table 2 presents the results of the SM optimization, i.e the most performing among the simple moves. Observe that the SM optimization gives a weighted hop reduction of 9.08%, at the expense of 51% increase in the number of masters (i.e piconets). In fact, according to their definition, SM moves transform the moving slave into a master creating a new piconet. The number of links instead is unchanged, since SM moves always tear off a link for another.

To improve performance in terms of weighted distance and keep the number of piconets small, we perform also master moves after having moved the slaves (i.e after the 1000th iteration). The results are presented in Figure 7. It can be seen that the largest hop reduction (14.64%) is obtained through the SM_MM optimization, whose performance is reported in Table 3. Table 3 shows that the 14.64% gain in hop reduction corresponds to 18.82% and 8.33% increase in the number of masters and links, respectively. We would like to mention that the SS_MS optimization produces a lower hop reduction (12.8%) but it increases the number of masters and links by only 1.49% and 0.54%, respectively. This highlights that master moves counterweight the increase in number of masters produced by slave moves.

Figure 8 presents the results of our third experiment, composed of slave optimizations followed by both MS and MM moves (diversifications $1000 \div 1999$ and $2000 \div 3000$, respectively). We obtained the best results with the SM_MS_MM optimization (see Table 4). This optimization gives also the best overall performance. However, if we take into account the average optimization execution times, we have: 26.94, 42.95 and 82.43 minutes for SM, SM_MM and SM_MS_MM, respectively. Thus, we can conclude that it is not worth performing both, MS and MM moves for additional 1-2% of hop reduction.

Finally, we highlight that the step-like behavior of function F in all of the three plots in Figures 6-8 suggests that most of the hop reductions happen at the beginning of each call to the optimizer, namely within the first 10-50 diversifications. Therefore, we can drastically reduce the number of diversifications and, thus the execution times without any significant impact on the overall performance. For example, repeating the SM, SM_MM and SM_MS_MM optimizations with $nr\_diversifications = 10$, the (execution time, reduction) pairs, expressed in [s,%], will be of (15.33, 8.67), (21.71, 13.82) and (42.8, 14.29), respectively.

5 Conclusion

In this work we presented a centralized method to dynamically adapt a Bluetooth scatternet topology to changing traffic conditions. We defined an algorithm suite that reconfigures the nodes role and links so as to minimize the number of hops between communicating nodes in the scatternet. Our simulations have shown that slave reconfigurations (i.e *moves*) increase the number of piconets, but this can be compensated by master moves, thus obtaining an overall hop reduction between all communication peers of about 15%.

The weaknesses of our approach are its centralized nature and its long execution time. Finding a decentralized solution as well as reducing execution times will be subject for future work.

Acknowledgment

This work was partially supported by the Italian Ministry of University and Research through the VICOM project and by the Autonomous Province of Trento through the WILMA project.

We thank Mauro Brunato for his feedback and for his help in the graphical visualization of the simulation environment.

References

[1] *Bluetooth Specifi cation 1.2*. May 2003.

[2] M. Ajmone Marsan, C. F. Chiasserini, A. Nucci, G. Carello, and L. De Giovanni. Optimizing the topology of Bluetooth wireless personal area networks. In *IEEE INFOCOM 2002*, New York, NY, USA, June 2002.

[3] S. Baatz, M. Frank, C. Khl, P. Martini, and C. Scholz. Adaptive scatternet support for Bluetooth using sniff mode. In *26th Annual Conference on Local Computer Networks, LNC 2001*, Tampa, Florida, USA, 2001.

[4] S. Basagni and C. Petrioli. A scatternet formation protocol for ad hoc networks of Bluetooth devices. In *IEEE Vehicular Technology Conference (VTC)*, pages 424–8, 2002.

[5] C. F. Chiasserini, M. Ajmone Marsan, E. Baralis, and P. Garza. Towards feasible distributed topology formation algorithms for Bluetooth-based WPANs. In *36th Annual Hawaii International Conference on System Sciences*, Big Island, Hawai, January 2003.

[6] C. C. Foo and K. C. Chua. Bluerings - Bluetooth scatternets with ring structures. In *International Conference on Wireless and Optical Communications*, Banff, Canada, July 2002.

[7] I. Foster. *Designing and Building Parallel Programs*, chapter 3.9. On-line edition, 1995.

[8] J. Hurink. *Introduction to Local Search*. Lecture notes, 2001.

[9] M. Kalia, S. Garg, and R. Shorey. Scatternet structure and inter-piconet communication in the Bluetooth system. In *IEEE National Conference on Communications*, New Delhi, India, 2000.

[10] C. Law and K. Y. Siu. A Bluetooth scatternet formation algorithm. In *IEEE Symposium on Ad Hoc Wireless Networks 2001*, San Antonio, TX, USA, November 2001.

[11] G. Miklós, A. Rácz, Z. Turányi, A. Valkó, and P. Johansson. Performance aspects of Bluetooth scatternet formation. In *First Annual Workshop on Mobile and Ad Hoc Networking and Computing (MobiHoc)*, pages 147–48, August 2000.

[12] A. Rácz, G. Miklós, F. Kubinszky, and A. Valkó. A pseudo random coordinated scheduling algorithm for Bluetooth scatternets. In *ACM Symposium on Mobile AD Hoc Networking and Computing*, volume 99, pages 1–100, Long-Beach, CA, USA, 2001.

[13] B. Raman, P. Bhagwat, and S. Seshan. Arguments for cross-layer optimizations in Bluetooth scatternets. In *Symposium on Applications and the Internet*, San Diego, CA, 2001.

[14] T. Salonidis, P. Bhagwat, L. Tassiulas, and R. LaMaire. Distributed topology construction of Bluetooth personal area networks. In *IEEE INFOCOM*, Anchorage, April 2001.

[15] Z. Wang, R. Thomas, and Z. Haas. Bluenet – a new scatternet formation scheme. In *35th Annual Hawaii International Conference on System Sciences*, Big Island, Hawaii, 2002.

[16] G. V. Zaruba, S. Basagni, and I. Chlamtac. Bluetrees - scatternet formation to enable Bluetooth-based ad hoc networks. In *IEEE International Conference on Communications (ICC 2001)*, volume 99, pages 273–7, 2001.