Reducing the Number of Hops between Communication Peers in a Bluetooth Scatternet

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

Carla-Fabiana Chiasserini
Marco Ajmone Marsan
Dipartimento di Elettronica
Politecnico di Torino, Torino, Italy
Email: {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.

I. Introduction

Even though Bluetooth was originally aimed to replace cables, its suitability for a wide range of applications became rapidly evident. One of the most appealing features of the Bluetooth 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 *scatternets* (i.e inter-connected piconets) with optimized link scheduling rules. However, the problem of efficient scatternet formation as well as many relevant optimization issues, discussed in numerous research papers [2], [3], [4], [5], [6], [7], [8], [9], are still not addressed in this latest version of the specification. The solutions for scatternet formation and optimization proposed so far in the literature typically aim at establishing the optimal tradeoff among the following performance metrics:

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

To the best of our knowledge, none of the previous works addresses the issue of scatternet optimization with respect to changing traffic flows. 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 creates optimal topologies, the scatternet would become suboptimal as new traffic flows show up because of changing source-destination associations and nodes' mobility. Therefore, forming an optimal scatternet may be not enough, since maintaining the scatternet in the state that optimally supports the actual traffic flows is also very important. We address this issue in this work.

Our approach consists of modifying the nodes' links and roles, and in searching for such a scatternet topology that minimizes the distance (measured in hops weighted with the corresponding traffic intensity) between every source-destination pair. This increases the network throughput, and reduces the packet delivery delay, nodes' energy consumption and communication overhead.

II. Notation 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 all masters*, and $\mathcal{S}$ the *set of all slaves*. Notice that 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

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

$\mathcal{T} = \{t_{sd}\}$ is the *traffic matrix* with

$$t_{sd} = \begin{cases} f & f \in (0,1], \text{ if data flow } f \text{ exists between } (s,d) \\ 0 & \text{otherwise.} \end{cases}$$

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

$\mathcal{P} = \{p_{ij}\}$ 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.

1) A *pure master* has on its row one entry equal to 1 for each of its slaves and has all 0 entries on its column (e.g., node 2 in Fig. 1).

2) A *pure slave* has one entry equal to 1 on its column corresponding to its master (e.g., node 7 in Fig. 1).

3) A *slave&bridge* has on its column exactly one entry equal to 1 for each of its masters (e.g., node 5 in Fig. 1).

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 (e.g., node 1 in Fig. 1).

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

We define function F, the weighted sum of the number of hops between each source-destination pair in the scatternet, as

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

Our objective is to solve the optimization problem

$$\mathcal{P}: \min_{\mathcal{H}} F,$$

subject to the constraints (2)–(8) [6].

A piconet must contain one master and up to 7 slaves,

$$\sum_{j=1}^{N} l_{kj} \leq 7, \forall\, k \in \mathcal{M}. \tag{2}$$

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

$$l_{ij} \leq p_{ij}, \forall\, i,j \in \mathcal{N}. \tag{3}$$

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

$$l_{ij} + l_{ji} \leq 1, \forall\, i,j \in \mathcal{N}; i \neq j. \tag{4}$$

All traffic between two nodes is routed through one of the available minimum length paths, not necessarily the same in both directions,

$$h_{sd} = h_{ds} \quad \forall\,(s,d) \in \mathcal{C} \tag{5}$$

$$\sum_{j\in\mathcal{N}} r_{ij}^{sd} + \sum_{j\in\mathcal{N}} r_{ji}^{sd} = 2h_{sd} \quad \forall\,(s,d) \in \mathcal{C}, \forall\, i,j \in \mathcal{N}. \tag{6}$$

Traffic on connection $(s,d) \in \mathcal{C}$ can be routed through edge (i,j) if and only if i and j are directly connected,

$$r_{ij}^{sd} \leq l_{ij} + l_{ji} \quad \forall\,(s,d) \in \mathcal{C}, \forall\, i,j \in \mathcal{N}. \tag{7}$$

In order to minimize F, loop-free routes are established,

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

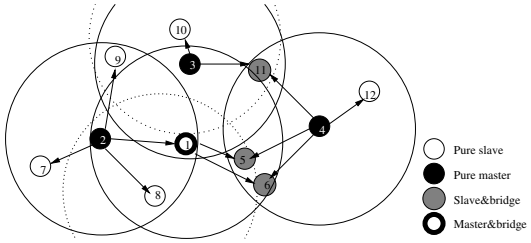

Fig. 1. Scatternet snapshot

III. Overview

To solve our problem we generate a connected and totally functional scatternet, as detailed in Section IV-A. 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 Fig. 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 optimizer is 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.

A. Move Types

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

We identify four kinds of possible moves.

Slave to Slave (SS) – a slave connects to a different master. Since moving bridge nodes influences considerably the routing scheme of the scatternet, we are not moving bridge nodes but only pure slaves.

Slave to Master (SM) – a slave creates a new piconet by paging another node. For example, for reducing the hop count between slaves 8 and 12 (Fig. 1), we can remove 8 from master 2, make 8 become a master and assign node 5 to it.

Master to Slave (MS) – a master becomes a pure slave. This 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 one-by-one and assigns them to an already existing master.

Master to Master (MM) – merging two piconets: a master overtakes all slaves of another master. This move can take place when all nodes in the two piconets are in the range of the persisting master (i.e the master maintaining its role after the move) and their number is not greater than 8. Additionally, the old master should also be connected to the persisting one.

IV. Our Solution

In this section, we describe in some detail our solution to scatternet formation and adaptation to traffic conditions.

A. Scatternet Generation

We generate an initial scatternet based on the algorithm proposed in [5]. We generate N randomly positioned nodes in the network area and assign to each of them a random weight indicating the node's willingness to assume special roles (master or bridge) in the scatternet.

There are two differences between the scatternet formation algorithm in [5] and the one we implemented. First, in the case of multiple choices when selecting slave&bridges, we always choose the node with the lowest degree and whose distance from the involved masters is minimal, in order to reduce the bridge scheduling overhead and select bridges that receive the strongest signal from the masters.

Second, we connect only those two-hop pure masters that are either not already connected or the shortest path between their pure slaves is at least 6 hops long. This way we avoid creating redundant paths and master&bridges connecting two piconets. However, if the traffic between the nodes of these two piconets is intense, new links reducing the number of hops will be created during the optimization. (Notice that the minimum distance between the pure slaves of two two-hop pure masters is 5, hence the value 6 above.)

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

B. The Optimization Procedure

The *optimization procedure* is the core of our work. It determines the modifications to be performed on the scatternet topology in order to reduce the number of hops between communicating nodes.

The optimization algorithm (Fig. 2) consists of a main body (lines 1-17) from which our optimization modules, namely SS, SM (either one denoted by SX in Fig. 2), MS and MM,

```
1.  OPTIMIZER
2.  L_init ← L
3.  F ← ActualNrHops()
4.  slavelist ← list of slaves (for SX module)
5.  masterlist ← list of masters (for MS and MM)
6.  for k ← 1 to nr_div do
7.      call one of SX, MS or MM optimization modules
8.      if F > ActualNrHops() then
9.          F ← ActualNrHops()
10.         L_opt ← L
11.     L ← L_init
12.     HUpdate()
13.     shuffle slavelist (for SX module)
14.     shuffle masterlist (for MS and MM module)
15. L ← L_opt
16. HUpdate()
17. end OPTIMIZER

18. SX optimization module
19. for each slave i in slavelist do
20.     execute SS and SM optimization
21.     perform the best move and update roles

22. MS optimization module
23. for each master i in masterlist do
24.     execute MS optimization
25.     update roles

26. MM optimization module
27. for each master i in masterlist do
28.     execute MM optimization
29.     perform the best move and update roles
```

Fig. 2. Pseudocode of the optimizer

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). Using the function *ActualNrHops()* we retrieve the number of weighted hops between each source-destination pair 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 SX, MS and MM optimization modules, respectively.

In line 6 we cycle trough the optimization procedure *nr_div* times. At every iteration we must choose only one move, the most favorable one, from a set of mutually excluding moves. Then, we randomly reorder the nodes in *slavelist* and *masterlist* at the end of each iteration, and repeat the search. By doing so, we change the order in which nodes are evaluated, and obtain a new series of moves to evaluate.

Within the cycle, one of the SX, MS or MM optimization modules is executed. Each of these modules performs a local search [10] executing one sequence of moves of the appropriate type. SX will evaluate the possibility of moving each slave using both, SS and SM moves (lines 18-21). The

move producing the greatest hop reduction will be accepted, and the node roles modified by this move will be updated accordingly. Similar operations are performed in the MS and MM modules, except that they act on the *masterlist* performing MS and MM moves, respectively.

A further difference is that for MS moves it is preferable to leave the reassigned slaves at their new master, instead of resetting them at the end of the move and moving them again in a later step of the MS module. In this way we can save CPU processing time. Thus, in the MS module the best move should not be re-executed, only the node roles must be updated (line 25).

If the optimization module gives a smaller value of F, we update the value of F and save the obtained configuration in $\mathcal{L}_{opt}$ (lines 8-10). Before moving to the next iteration of the *for* loop, we set $\mathcal{L}$ to its initial value (line 11). This requires $\mathcal{H}$ to be updated. *HUpdate()* takes the necessary input data from $\mathcal{L}$, uses Floyd's algorithm for solving the *all-shortest path problem* [11], and stores the result in $\mathcal{H}$. Finally, the nodes in *slavelist* or *masterlist* are reordered (i.e a *diversification* is done) for 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.

The optimization algorithm can execute the optimization modules in sequence, 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.

Regardless of the optimization modules used, after executing the optimizer a certain number of times, the so-called diversifications (nr_div), 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 unacceptably long period of time. Therefore, a good trade-off between the number of diversifications and execution time should be found to achieve acceptable performance in real environments.

C. Reduction of Hops by Using SS and SM Moves

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 *id* and all its communication peers. During the optimization, *id* is sequentially moved to each node in its radio proximity, except those that are in the same piconet as *id*. Additionally, in the SS algorithm, neighboring masters with already 7 slaves are also excluded from the search.

Consider that slave *id* is connected to master *m* while the target neighbor, *i*, is connected to a different master (i.e $l[m][id] = 1$ and $l[m][i] = 0$). The SS and SM moves alter these settings. In the case of an SS move, *i* pages slave *id*, while for SM moves slave *id* becomes a master and pages neighbor *i*. If the value of F after the move is decreased, the move is stored. After the current evaluation, the original links are restored, so that the subsequent move can be executed under the same conditions.

At the end of the procedure, after all neighbors of *id* have been checked, the move minimizing F is returned to the optimizer.

D. Reduction of Hops by Using MS Moves

The MS optimization algorithm changes the role of a master, *id*, into slave, connects each of *id*'s slaves to a new master and returns a list of performed moves.

In the first phase of the MS algorithm, all slaves of *id* are assigned to a new master that minimizes the number of hops between each slave and its communication peers. If any of the slaves cannot be assigned to some other master, the algorithm terminates, indicating that no MS optimization is possible with master *id*.

In case all slaves were successfully reassigned to masters, *id* itself is also moved to one of its neighboring masters, *i*. To become the new master of *id*, *i* must have less than 7 slaves and minimize the value of F by accommodating *id* in its piconet. Moreover, it should be assured that after assigning *id* to *i* a path exists between *id* and all its earlier slaves, in order to keep the scatternet connectivity unaltered. Finally, it should be verified whether *id* is the master of *i* and whether they have a third master in common. Creation of triangles or making two nodes masters of each other has to be avoided.

If all of the above conditions are met, master *i* is stored and the search continues with the next neighboring master. After all masters have been checked, the one giving the greatest hop reduction is chosen as the new master of *id*, and the corresponding sequence of moves is returned to the optimizer. If no master exists that meets all of the above conditions, no move with *id*'s slaves is performed, and the algorithm terminates without any hop reduction.

E. Reduction of Hops by Using MM Moves

Although MM moves target masters, the structure of the MM algorithm is similar to the SS and SM procedures. It consists of checking every neighboring master *i* of a master *id* (with the goal of merging their piconets), saving their links to their slaves, checking the hop reduction, and resetting the saved links.

To this end, first we verify whether the total number of slaves in the piconets of *id* and of a particular *i* is less than 6. This condition is necessary to respect the maximum number of 8 nodes in the new piconet. Second, we check whether all slaves of master *i* are in range of *id*. If such a master is identified, the piconets can be merged using an MM move.

After all masters have been checked, the master whose piconet merging would reduce the most the number of weighted hops is returned to the optimizer.

V. Numerical Results

To evaluate the performance of our algorithms, we implemented a Bluetooth scatternet simulator in C++. Since the

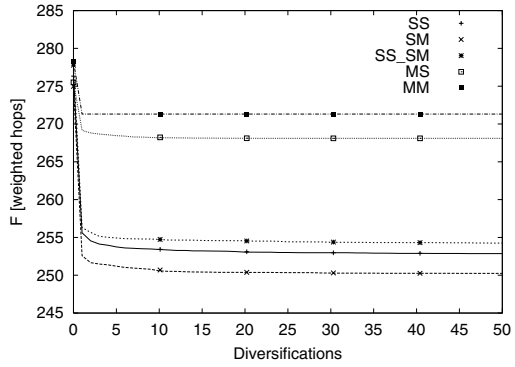

Fig. 3. Optimization with simple moves

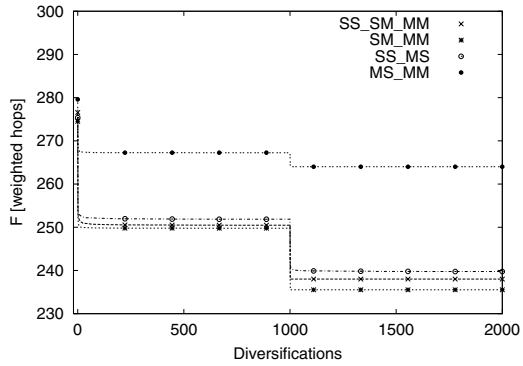

Fig. 4. Optimization in two phases

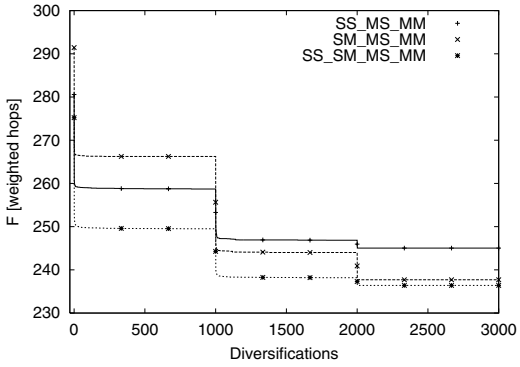

Fig. 5. Optimization in three phases

TABLE I
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

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 $nr\_div = 1000$. We randomly generated 100 source-destination bidirectional (200 unidirectional) communication pairs (elements of $\mathcal{C}$). The traffic intensity (f) 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 experimented combining the SS, SM, MS and MM optimizations in many different ways. For the sake of brevity, here we present only the results derived from some 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. Fig.s 3-5 show the evolution of function F during these experiments, against the number of diversifications.

Then, for each group of experiments, the one giving the best performance is evaluated in greater details in Tables I-III. 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 (f) is expressed in *weighted hops* ([*wh*]), and the distance is measured in *hops* ([*h*]). The rows referring to the SM, MS and/or MM 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 Fig. 3 and Table I. In Fig.3, 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 I 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

TABLE II

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 III

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

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 Fig. 4. It can be seen that the largest hop reduction (14.64%) is obtained through the SM_MM optimization, whose performance is reported in Table II. Table II 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.

Fig. 5 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 III). 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 Fig.s 3-5 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 $nr\_div = 10$, 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.

VI. 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 showed 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 Specification 1.2*, May 2003. [Online]. Available: http://www.bluetooth.com

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

[3] 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)*, vol. 99, 2001, pp. 273–7.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] I. Foster, *Designing and Building Parallel Programs*. Online edition, 1995, ch. 3.9. [Online]. Available: http://www-unix.mcs.anl.gov/dbpp/text/node35.html