# An On-Demand Bluetooth Scatternet Formation Algorithm*

Elena Pagani, Gian Paolo Rossi, and Stefano Tebaldi

Computer Science Dept., Università degli Studi di Milano
{pagani,rossi}@dico.unimi.it, Stefano.Tebaldi@unimi.it

**Abstract.** In this paper, we propose the *On-Demand Bluetooth scatternet formation algorithm* (ODBT). ODBT characterizes an ad hoc infrastructure with a tree topology. It is able to cope with topology changes due to either leaving or moving Bluetooth devices, as well as with devices that dynamically join the scatternet. It can support out-of-range devices. We describe in detail how ODBT can be implemented in the Bluetooth protocol stack, and we analyze its performance in comparison with other proposals existing in the literature, also by means of simulation techniques.

## 1 Introduction

The Bluetooth technology has been designed with the purpose of replacing cabling amongst neighbor devices, for instance to connect computers and mobile phones to external devices and accessories via wireless links. Bluetooth is an interesting solution to rapidly deploy wireless infrastractures using low cost, easily available devices such as PDAs, notebooks and cellular phones.

The base Bluetooth network infrastructure is represented by a *piconet*, that is formed by up to 8 Bluetooth devices (BDs) actively participating in the communications, one of which has the role of *master* while the others act as *slaves*. The communication range of a BD is around 10-30 mt., arriving for some devices to up to 100 mt.; this range is the maximum piconet radius. The Bluetooth specification includes the possibility of building *scatternets*, obtained by connecting several piconets into an ad hoc infrastructure. The scatternets allow to increase the communication range and the number of BDs involved in a system. Yet, the scatternet formation mechanism is not provided by the specifications, and is currently matter of research.

In this paper, we present the *On-Demand Bluetooth Scatternet Formation* algorithm (ODBT) to characterize a communication infrastructure connecting a set of BDs in a tree topology. The scatternet formation is started when needed by an initiator node, that becomes the tree root; the other BDs are progressively grafted to the structure, so that the overall topology is optimized with respect to the latency in the data forwarding. Tipically, the tree root can be the group

* This work has been partially supported by the Italian Ministry of Education, University and Research in the framework of the FIRB "Web-Minds" project.

coordinator or the data source. The tree structure guarantees the absence of loops and minimizes the number of roles each BD can have in the scatternet, thus reducing the probability that a BD can represent a bottleneck.

## 2 Bluetooth Architecture

In this section we provide a brief overview of the Bluetooth protocol stack and operations. The complete Bluetooth specification is provided in [2]. In figure 1($a$), we show the low layers of the Bluetooth protocol stack. As the wireless media, Bluetooth exploits the internationally unlicensed ISM band ranging from 2.4 GHz to 2.48 GHz. It adopts Frequency Hopping Spread Spectrum as the radio transmission technique, to achieve robustness to the interferences. The hop rate is 1600 hops per second; hence, each hop slot length is 625 $\mu$sec. The channel is partitioned in 79 subchannels having 1 MHz bandwidth each. The baseband services are exploited by the Bluetooth devices (BDs) to agree on the frequency hop sequence and to establish the links. The functionalities for the piconet set-up are involved in the baseband and the Link Manager Protocol (LMP) layers. LMP also allows to perform the BDs, links and packets configuration. The baseband and LMP services are accessed through the Host Controller Interface (HCI). The

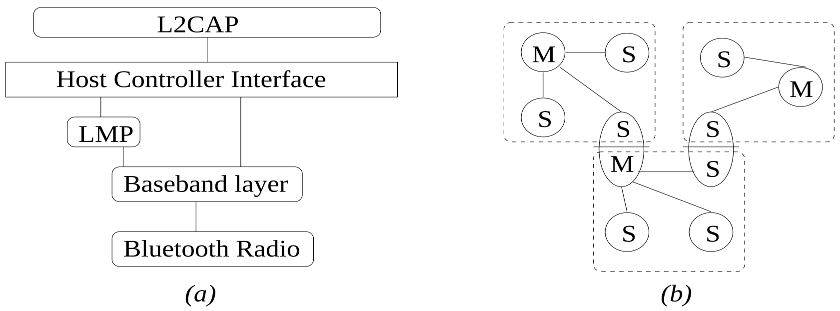



**Fig. 1.** ($a$) Bluetooth architecture. ($b$) Example of a scatternet infrastructure.

network layer services are provided by the Logical Link Control and Adaptation Protocol (L2CAP).

The piconet is built in two steps: the `inquiry` phase and the `page` phase. In the former step, a BD may discover its neighbors by sending broadcast messages (`inquiry` procedure). A BD that wants to be discovered performs the `inquiry_scan` procedure: upon receiving an `inquiry` message, it replies with its own 48-bits MAC layer address (BD_ADDR) and an estimate of its clock value. In this phase, the BDs are not yet synchronized: in order to increase the probability that two BDs contemporarily use the same frequency and can then successfully communicate, an *inquiry hopping sequence* is used involving 32 frequencies. Those frequencies are split into two groups of 16 frequencies each (a *train*); each train is repeatedly tested before switching to the other train. The BD_ADDRs and clock estimates of the discovered BDs are used in the successive `page` phase to establish a connection. The paged device (the slave) performs the `page_scan` procedure by listening for a hop pattern computed basing on its own BD_ADDR, that was previously sent to the master. The paging device (the master) enters the `page` state, in which it sends to the slave information concerning its own BD_ADDR and clock value, which is used to compute the frequency hopping pattern for the subsequent communications. Since the two BDs clocks are not yet synchronized, the `page` procedure is carried out similarly to the `inquiry` procedure: the paging pattern involves two trains of 16 frequencies each, that are repeatedly tested. The `page` procedure is carried out separately for each master's neighbor.

Once the channel has been successfully established at the end of the `page` procedure, the master and its slaves communicate exploiting a time-division duplex policy. The even-numbered slots are for master-to-slave communications; the odd-numbered slots are for slave-to-master communications. A slave can use a slot only if it is allowed to by the master in the immediately preceding slot.

As we said before, a master can manage up to 7 *active* slaves, that can participate in the communication; in the `page` phase, the master assigns to each active slave a 3-bits active member address (AM_ADDR). Indeed, the number of slaves grafted to a given master can be higher than 7, but the remaining slaves must be in *park* mode. A BD that does not need to participate in the communication can enter the park mode: the master substitutes the slave AM_ADDR with a PM_ADDR, that can be used to un-park the BD. A BD can enter the sniff mode to save energy: it listens to the channel only every $T_{sniff}$ slots and it maintains its AM_ADDR. A slave can be put in hold mode by the master for a hold time $holdTO$; in the meanwhile, it does not listen to the channel, but, it can be active in other piconets and it maintains its AM_ADDR. When $holdTO$ expires, the slave must wait for a re-synchronization message from the master.

When two piconets are connected into a scatternet structure, the BDs belonging to both piconets assume the role of *bridges*. In fig.1($b$), three piconets are shown (dashed squares): two piconets are connected through a BD behaving as slave in both of them (S/S bridge); two piconets are connected through a BD behaving as slave in one piconet and as master in the other piconet (M/S bridge). The bridges take in charge the traffic forwarding amongst piconets. Since a BD can be active only in one piconet at a time, the bridges must become active alternatively in each piconet they are connected to. As a consequence, the bridges represent the potential bottlenecks.

### 2.1 Related Works

A few other works exist in the literature, that propose mechanisms to form scatternet infrastructures. In Bluetree [8], the scatternet formation is initiated by the Blueroot, whose identity is predetermined. The Blueroot starts connecting all its neighbors as slaves. Then, the Blueroot slaves act as masters for their neighbors and so on recursively. The maximum allowed number of slaves for each master is

5. To fulfil this constraint, in the second phase of the algorithm the scatternet is reconfigured by splitting the piconets that are too dense. Bluenet [7] builds scatternets by initially aggregating the BDs into piconets involving at most $N_{max}$ slaves, and then by interconnecting the piconets. In the simulations presented in [7], $N_{max}$ was equal to 5. In Bluenet, a bridge node must avoid to form multiple links with the same piconet. This is achieved by having the BDs that provide information about the piconet they currently belong to, when they are paged. BTCP [5] is a 3-phases algorithm: in the first phase an election protocol is carried out, at the end of which the chosen coordinator knows the identities of all the participants. In the second phase the coordinator chooses other $P-1$ masters to form $P$ piconets, and $P \cdot (P-1)/2$ bridges, thus characterizing a completely connected topology among the piconets in the third phase. The probability that the scatternet is connected depends on the time spent in electing the coordinator. In [5] some considerations are presented, concerning the timers sizing. LMS [3] is a distributed randomized algorithm. The BDs try to aggregate into piconets, which are then merged to form a scatternet. Piconets are reconfigured to make them dense (by merging low populated piconets), thus minimizing the number of piconets composing the scatternet. It has been proved that the formed scatternets have a $O(\log n)$ diameter, with $n$ the number of involved BDs, and the message complexity equals $O(n)$. Some extensions are outlined to support moving, joining and leaving nodes, as well as BDs that are not all in range with each other. TSF [6] characterizes a tree scatternet by building a forest of piconets and scatternets that are then merged. The merging procedure is such that it guarantees loop freeness. TSF is the unique solution that explicitly allows nodes to start communicating while the scatternet is under construction. Among the considered algorithms, only TSF explicitly addresses the problems related with the management of the node mobility, and BDs dynamically joining and leaving the scatternet. Anyway, TSF may fail in characterizing a connected scatternet if the nodes are not all in range of each other; in particular, it is not able to merge two scatternets if their roots are out of range.

The ODBT algorithm we propose in this work represents an improvement on the Bluetree algorithm. We study in depth the issues involved with the practical implementation of the scatternet formation algorithm in the Bluetooth architecture, and we describe mechanisms to support mobility and BDs dynamically joining and leaving the scatternet. ODBT can, to some extent, support out-of-range BDs; moreover, it allows the data forwarding during the tree construction. ODBT achieves a trade-off between the data transmission latency and the control overhead, trying to maintain a low tree depth. In sec. 4, we compare the above algorithms with ODBT.

## 3 Scatternet Formation Algorithm

In this section, we describe ODBT. For the sake of simplicity, in section 3.1 we firstly assume that the BDs are all in range, and that the BDs do not move, neither they dynamically join or leave the system; crash failures are not considered. In section 3.2, we discuss in detail the synchronization issues that allow the link maintenance and the message forwarding, by appropriately controlling the behavior of the bridge BDs switching between two piconets. In section 3.3, we finally relax the assumptions above by considering the mechanisms adopted to deal with dynamically changing membership and topology.

### 3.1 Algorithm Overview

The scatternet formation is initiated by the *scatternet root* (SR). We assume that each BD knows whether it has to assume the role of SR or not. This assumption is appropriate, as for the most part of the customized applications requiring a scatternet infrastructure, the BD coordinating the group activities is known at configuration time.

The scatternet is built trying to minimize the number of piconets by maximizing the number of BDs involved in each piconet, with a twofold purpose: the lower is the number of piconets, the lower are both the path lengths and the number of bridges, which are bottlenecks in the data forwarding because they have to switch between two piconets. To build the first scatternet layer, the SR performs the `inquiry` procedure, while all the BDs different from SR start performing the `inquiry_scan` procedure. The `inquiry` and `page` procedures are performed as usual to form a piconet, but they are repeated until the SR connects exactly 7 slaves[1]. At that point, the SR notifies its slaves that they may in turn continue the scatternet formation, by becoming masters and connecting other BDs. This procedure is recursively repeated, with each master that, after having connected 7 slaves, authorizes them to become masters and to form new piconets. Notice that, continuing the scatternet formation only when a piconet is full does *not* guarantee that the obtained tree is balanced. Indeed, one branch may grow faster than another, because it is not required that a whole tree level is completed before starting a new one. Yet, this policy aims at filling up each piconet as much as possible before creating new piconets, while a good balancing is guaranteed by the contemporary start of all the bridges belonging to the same piconet. A perfectly balanced tree could be obtained only by having the SR that coordinates the tree growth level by level. This would be expensive in terms of the number of control messages that should be exchanged.

Once a BD is connected to a master, it does not reply to the `inquiries` of other BDs; as a consequence, all the bridges in the scatternet are of type master/slave and each BD belongs to at most two piconets. This also guarantees that the tree is loop free: indeed, a loop could form if a master connects as slave a BD already connected to another piconet.

The bridge BDs must alternate between the slave role in the upstream piconet and the master role in the downstream piconet. According to the Bluetooth specification, a link is considered broken if it is not used for a *supervisionTO* timer, whose default value is 20 sec. To guarantee that links are not disrupted in a piconet while a BD is active in the other, we choose to exploit the *Hold*

[1] Remind that we are assuming that all the BDs are in range.

*Mode* for two reasons: the hold mode does not require to reassign the BD active member address, and a BD in hold mode does not listen to the channel until the *holdTO* expires, thus allowing energy saving. As we discuss in sec.3.2, the *holdTO* value must be negotiated depending on the state of the downstream piconets. According to the Bluetooth specification, the hold mode negotiation requires one slot for the master to broadcast the `LMP_hold_request` and one slot for each slave to reply with a `LMP_hold_accepted`. In figure 2, we show the

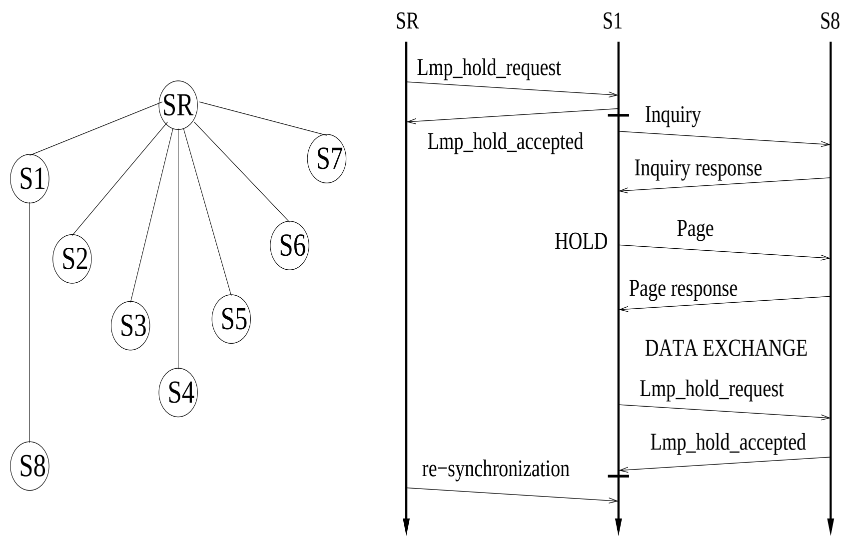



**Fig. 2.** Communication pattern for the first tree levels.

communication pattern for the SR and the first 2 levels of the tree. When SR has grafted the slaves $S_1$ through $S_7$, it authorizes them to form new piconets. Then, SR puts its slaves into hold mode so that they can become active as masters. Before the *holdTO* imposed by SR to its slaves expires, $S_1$ must put its own slaves into hold mode. This way $S_1$ can return active in the SR piconet without its inactivity being interpreted as a disconnection by $S_8$. While a piconet is under construction, the slaves it involves simply alternate between the active and the hold state.

### 3.2 Device Synchronization

To guarantee the tree connectivity we must carefully synchronize the BDs; the synchronization must be compliant with the Bluetooth specification. In figure 3, we show the lifecycle of a bridge, a connected slave of its still incomplete piconet and a free BD[2] throughout the tree construction. Those lifecycles have

[2] That is, a BD that does not belong to any piconet.

to fit together: a master that is forming its own piconet must periodically return active in the upstream piconet as a slave, according to the *holdTO* imposed by its own master. By contrast, when the SR's slaves are active in their own

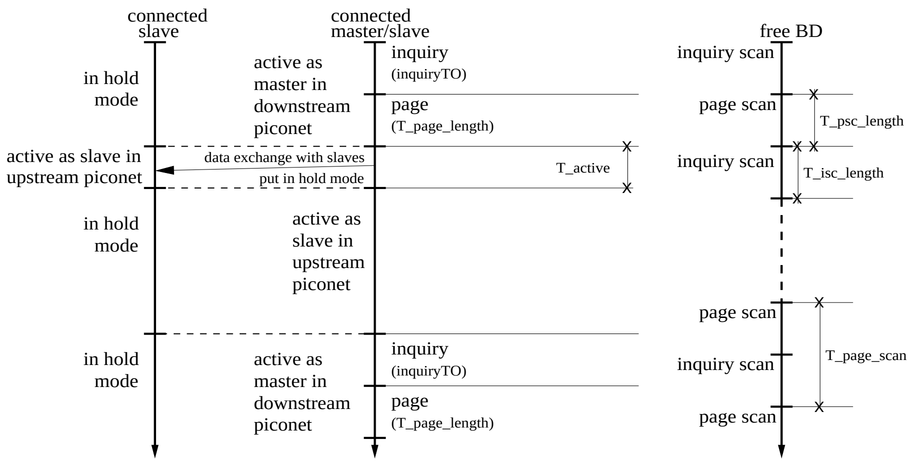



**Fig. 3.** Timers sizing.

piconets, SR remains inactive (fig.2).

Let us define the following time intervals:

*inquiryTO* is the interval in which a bridge performs the `inquiry` procedure;

$T_{isc\_length}$ is the interval during which a free BD performs the `inquiry_scan` procedure. Since a free BD that receives an `inquiry` waits for a random time $T_R$ ($0 < T_R < 1023$ slots, that is, $T_R \leq 0.64$ sec) before replying, it must be $T_{isc\_length} > T_R$;

$T_{psc\_length}$ is the interval during which a free BD either performs the `page_scan` procedure, if it has replied to an `inquiry` message in a previous `inquiry_scan` phase, or it stays inactive waiting to start a new `inquiry_scan` phase;

$T_{page\_scan}$ is the interval between the beginnings of two consecutive `page_scan` phases for a free BD. It equals $T_{isc\_length} + T_{psc\_length}$;

$T_{page\_length}$ is the interval during which a bridge performs the `page` procedure;

$T_{active}$ is either the interval in which a bridge is active as a master in its own piconet or the interval in which a BD is active as a slave in the upstream piconet.

The *inquiryTO* and $T_{page\_scan}$ timers are defined as in the specifications. We can derive the above timer values from the Bluetooth specification, as follows. The minimum value for *inquiryTO* is 1.28 sec., which allows to perform half of a train test. We adopt $inquiryTO = 1.28$ sec. To maximize the probability that a master and a free BD meet, thus minimizing the scatternet formation latency, we choose the time spent in the `inquiry_scan` state ($T_{isc\_length}$) equal to $inquiryTO$, that is, 1.28 sec. Moreover, we set $T_{psc\_length} = 1.28$ sec.; this way, $T_{page\_scan} = 2.56$ sec. If the interval between two consecutive `page_scan` phases is between 1.28 sec. and 2.56 sec., then the master shall use mode R2 to perform the `page` procedure, that is, it must repeat a train scan at least 256 times, thus spending 2.56 sec. which we adopt as the value of $T_{page\_length}$. $T_{active}$ must be long enough to allow the master to turn into hold mode its slaves, and to forward them some data. We choose to have the BDs active for 28 slots in each piconet: 14 slots are for the data exchange and 14 slots are for the hold mode negotiation. Hence, $T_{active} = (28 \cdot 0.625 msec.) = 17.5$ msec. A bridge stays active 17.5 msec. in the upstream piconet and 17.5 msec. in the downstream piconet. We also need a mechanism to de-synchronize the masters and the free BDs: if a master constantly performs the `inquiry` phase while some of the free BDs surrounding it are in the `page_scan` phase, they never connect. We introduce some randomization by having each BD that waits for a random time after the bootstrap before starting the first `inquiry_scan` procedure.

When a piconet is completed, the execution of the `inquiry` and `page` procedures is useless, and the activity time of a master can be completely exploited for the data exchange. In fig.4, we show the timers settings after the algorithm

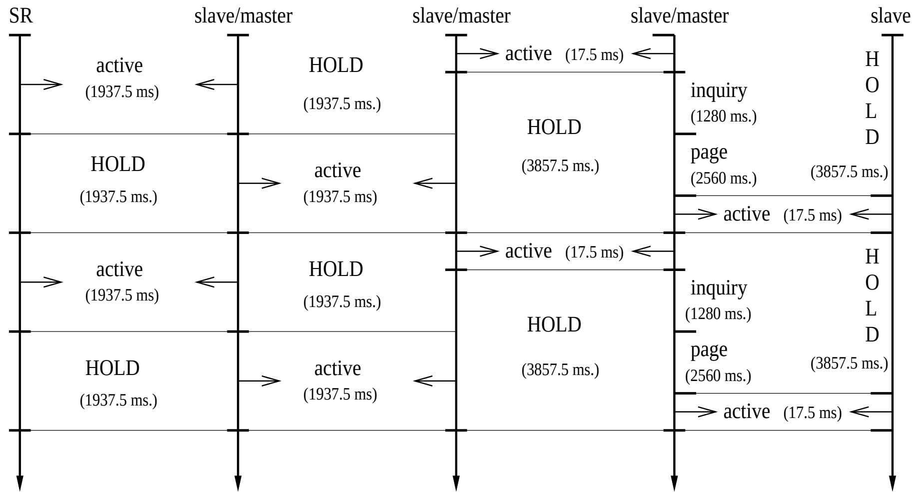



**Fig. 4.** Timers sizing after the scatternet has been built.

termination in the case of a 5-layers tree. The masters of the leaf piconets continue performing the `inquiry` and `page` procedures to support joining BDs, as we discuss in the next section. This requires an appropriate sizing of $holdTO$ at the above layer, that is achieved through the hold mode negotiation. The `inquiry` and `page` phases in the leaf piconets could be avoided only under the assumption that the number of BDs that must be connected to the scatternet is a priori known, or that an agreement is maintained about the membership cardinality in spite of dynamic join and leave events. With this assumption, the whole bandwidth could be used for the data exchange once it is known that all the BDs have been grafted. Yet, we do not consider this assumption because in real environments the membership knowledge is in general not available.

### 3.3 Dynamic Membership and Topology

In this section, we describe the mechanisms to support dynamic changes in the scatternet topology and membership.

A bridge that leaves the scatternet partitions the ad hoc infrastructure leaving disconnected all the downstream piconets. The same occurs if a bridge moves out of the range of its upstream or downstream neighbors. To reconnect the scatternet, the disconnected subtree is *flushed*: a bridge noticing the disconnection from its parent[3] notifies the failure to its own children and so on recursively, so that all the BDs in the disconnected subtree become free and they can rejoin. A free BD can graft the scatternet by becoming a slave in a not yet completed piconet.

It is more difficult to release the assumption concerning out-of-range BDs. In [8], the definition of *geographically connected* system is provided. A set of BDs is geographically connected if a (multi-hop) path exists between every two of them. This property is necessary to build a connected scatternet; anyway, it is not sufficient. If a master $M$ exists having $N$ free BDs in range, with $N$ greater than the number of BDs $M$ can connect, then $M$ must select a subset of those BDs as slaves. The other free BDs could be out of the range of every other master, thus failing in connecting the scatternet.

As we will discuss in sec.4, the ability of exploiting the system geographical connectivity varies in the different algorithms. ODBT may suffer of the above syndrome. Anyway, the scatternet formation procedure may be modified to deal with out-of-range BDs while doing a best effort to form a connected infrastructure. According to the Bluetooth specification, in an error free environment the `inquiry` procedure must span at least 3 train switches (i.e., 10.24 sec.) to collect all the responses. We modify ODBT so that each master performs the `inquiry` procedure 8 times, before allowing its slaves to become bridges and to take forward the scatternet formation. This way, we still try to maximize the number of slaves per piconet, while at the same time preventing the algorithm from starving on a scatternet branch because of the lack of enough BDs in the range of a master. To dynamically graft moving and joining BDs, moreover, each master having less than 7 slaves periodically performs the `inquiry` and `page` procedures.

## 4 Performance Evaluation

In Table 1, we summarize the characteristics of ODBT and the other scatternet formation algorithms described in 2.1. The characterization of a tree topology,

[3] Because of the $supervisionTO$ expiration.

as done by ODBT, guarantees that no loops exist. The tree can be exploited as is for broadcast communications; otherwise, routing is easy to perform by appropriately forwarding the data along a subset of the tree branches. By contrast, when redundant paths exist between two BDs, a routing service is needed to avoid bandwidth waste in the message forwarding, but, the scatternet has a greater resilience to node mobility and failures. ODBT exploits M/S bridges only: according to [4], they allow to achieve lower inter-piconet delay with respect to S/S bridges. Differently from ODBT, the most part of the solutions proposed so far does not support BDs mobility or nodes leaving the scatternet, nor BDs joining after the procedure completes. Joining nodes can be supported provided that the masters of the incomplete piconets (i.e. those with $< 7$ slaves) periodically perform inquiries in order to discover them. The greater the number of incomplete piconets, the greater the probability that a free BD moves in a piconet that can graft it. On the other hand, the more populated are the piconets, the lower is the number of piconets, and the shorter are the paths within the scatternet. A short path requires a low number of bridge's switchings between adjacent piconets, thus providing a low latency. Bluetree and Bluenet explicitly bound the maximum number of slaves per piconet, but they are not able to exploit this characteristic to support dynamically joining devices. The goal of minimizing

**Table 1.** Comparison among scatternet formation algorithms.

| | ODBT | Bluetree | Bluenet | BTCP | LMS | TSF |
|---|---|---|---|---|---|---|
| topology | tree | tree | graph | fully connected | graph | tree |
| bridge type | M/S | M/S, S/S | M/S, S/S | M/S, S/S | M/S, S/S | M/S |
| mobility support | yes | no | no | no | with extensions | yes |
| Join/Leave support | yes | no | no | no | with extensions | yes |
| dense piconet | $\leq 7$ | 5 | $N_{max}$ | <7 | 6 | $\perp$ |
| # phases | 1 | 2 | 3 | 3 | 1 | 1 |
| out-of-range BDs | yes | yes | yes | no | with extensions | no |

latency also motivates the choice of all the algorithms of having $\leq 7$ slaves for each piconet: the management of the non-active slaves would prohibitively increase the re-synchronization complexity. On the other hand, apart from ODBT, all the considered works do not discuss in detail the problems involved by the BDs timer configuration for the switching between adjacent piconets. A performance index of a scatternet formation algorithm is the time spent to terminate before starting the data forwarding. Some algorithms require multiple phases, involving the discovery of the existing BDs and possible reconfigurations of the infrastructure characterized in the previous phases. Those reconfigurations may hinder the fast scatternet formation, thus negatively affecting the algorithm performance particularly in the case of mobility. By contrast, ODBT, LMS and TSF build the scatternet in one phase. The algorithms differ in the ability of dealing with out-of-range BDs. Those BDs cannot be supported by both BTCP and TSF. In the former algorithm, all BDs have to be in range to carry out the coordinator election. In the latter algorithm, two neighbor components could be prevented from merging because the respective roots are out of range. Bluenet has problems similar to TSF, but its ability of merging two piconets exploiting any BD makes it more tolerant to out-of-range devices. The ODBT behaviour should be similar to that of Bluenet with respect to supporting out-of-range BDs. Bluetree seems to be the more effective solution to support out-of-range BDs, thanks to the capability of reorganizing the piconets. Notice that dealing with out-of-range BDs allows to build scatternets spanning a larger area, thus overcoming the limitations due to the short communication range provided by the Bluetooth technology.

### 4.1 Simulation Results

In this section we provide a quantitative analysis of ODBT by means of simulation techniques. ODBT has been implemented in the framework of the NS-2 network simulator, extended with Blueware [1] that simulates the Bluetooth protocol stack. Blueware includes the TSF implementation; this allows us to compare the behaviors of ODBT and TSF. So far, the simulation environment does not allow to reproduce BDs movements, nor it involves the notion of distance amongst BDs. As a consequence, we performed experiments only with in-range[4] BDs that do not move.

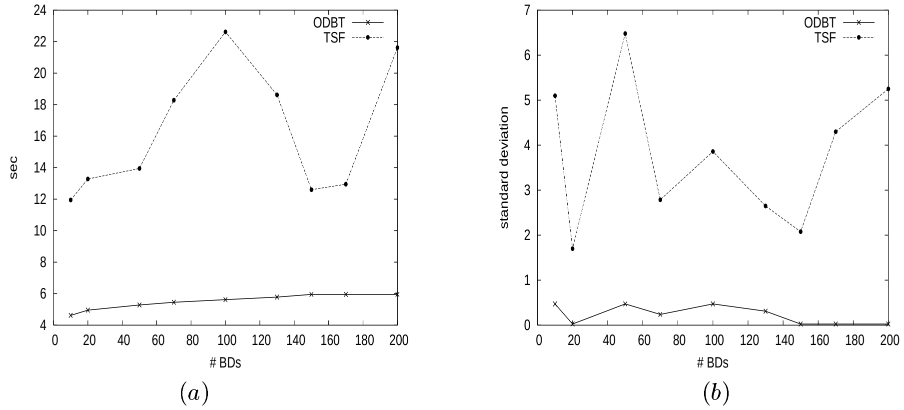
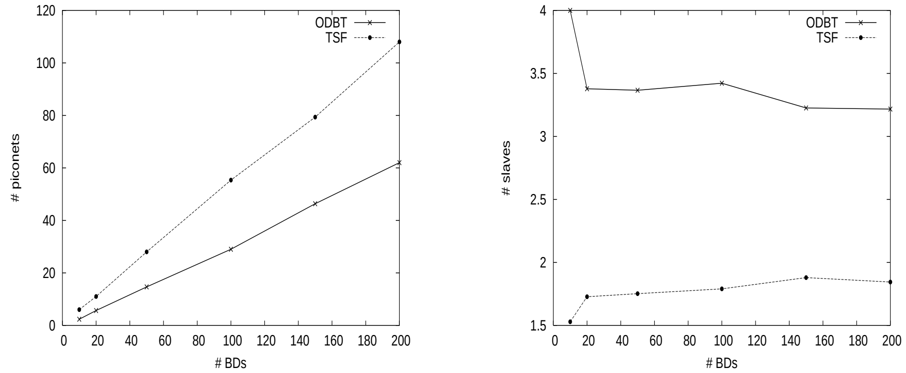



**Fig. 5.** $(a)$ Termination time of the scatternet formation algorithms. $(b)$ Standard deviation of the termination time.

The number of BDs varies between 10 and 200; the device with identifier 0 assumes the role of SR. We performed experiments with both BDs starting all at the same time and BDs starting sequentially every 2 sec., obtaining comparable results. The results reported in this section refer to the latter case; every experiment has been repeated 50 times and we report the average results.

[4] This is a required TSF condition.

In fig.5($a$), we show the average termination time. In TSF, when two components (either piconets or scatternets) meet, they can be merged into one. In our experiments, we observed that TSF tends to form pairs of BDs (a master connected to only one slave), that are then combined. This requires checking that the newly created links do not form loops, and possibly performing a master/slave switch to merge the two components. These operations are time consuming and they negatively affect the algorithm performance. Moreover, the algorithm behaviour is highly unstable: in fig.5($b$), we show the standard deviation of the termination time. That behaviour depends on how the components form and



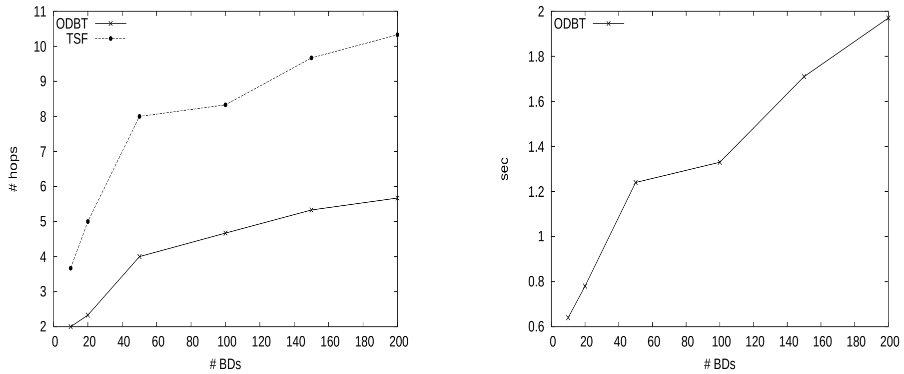

($a$) ($b$)

**Fig. 6.** ($a$) Average number of piconets forming the scatternet. ($b$) Average number of slaves per piconet.

merge throughout the simulation, depending on random events that make the algorithm performance not predictable. This characteristic may be a flaw with respect to the service provided to the users, in that it delays the data forwarding particularly to the last connected BDs, that can observe data loss if the transmission starts while the scatternet formation is still ongoing.

ODBT builds more dense trees than TSF: in fig.6($a$), we show the average number of piconets composing the scatternet, while in fig.6($b$) we report the average number of slaves per piconet. In the ODBT case, indeed, all the high level piconets have 7 slaves in the adopted experimental setting; the average is decreased by the leaf piconets. Having the slaves in a piconet that contemporarily start behaving as bridges allows to obtain balanced trees: in our simulations we observed a maximum difference of one level among the leaf piconets. By contrast, the random component merging in TSF tends to form low populated piconets that concatenate to each other yielding deep trees (fig. 7($a$)).

Minimizing both the tree depth and the number of piconets allows to minimize the number of bridges as well. Bridges represent the bottleneck in the data forwarding, as a bridge receiving data from its upstream master has to wait that the master puts it into hold mode, before acting as a master on its own and forwarding the data in its downstream piconet. We evaluated the la-




$(a)$ $(b)$

**Fig. 7.** $(a)$ Average tree depth. $(b)$ Maximum latency in the data forwarding with ODBT.

tency perceived at the farthest BD from the SR, after the scatternet has been formed, averaged on 1000 data packets (fig.7$(b)$); the reported results refer to the complete infrastructure where the leaf masters do not execute the `inquiry` and `page` procedures. The latency increases with the number of involved BDs proportionally to the increase of the tree depth.

## 5 Concluding Remarks

In this work we present the On-Demand Bluetooth scatternet formation protocol (ODBT) for the formation of scatternets with a tree topology. We show how the Bluetooth stack should be configured to implement ODBT. We compare ODBT with other algorithms existing in the literature, also by means of simulation techniques.

ODBT builds efficient tree structures, minimizing the number of bridges connecting the piconets and relying only on M/S bridges. ODBT shows a stable behaviour for increasing number of involved BDs. It is able to dynamically reconfigure the scatternet to deal with joining, leaving and moving BDs, and it can support out-of-range devices.

We are currently extending the simulator with the notion of distance amongst BDs and the simulation of movements, to perform further experiments with ODBT in order to evaluate its behavior in the presence of out-of-range moving devices. In particular, we are interested in observing how the data transmission latency varies as a function of the number of slaves that are grafted in the piconets.

As a future work, we plan to extend the protocol so that it can operate in the presence of multiple SRs. In the case different devices contemporarily start the formation of a scatternet involving the same BDs, we must be able to merge the different scatternets into one to guarantee the system connectivity. This can be achieved by including a SR election algorithm in ODBT. Further developments concern the implementation of unicast and multicast routing services on top of ODBT, and the deployment of ODBT on a testbed platform.

## References


1. Balakrishnan H., Guttag J., Miu A., Tan G.: *"Blueware: Support for Self-organizing Scatternets"*. `http://nms.lcs.mit.edu/projects/Blueware/`.
2. Bluetooth Special Interest Group: *"Bluetooth V1.1 Core Specifications"*. (May 2001). `http://www.bluetooth.org/`.
3. Law C., Mehta A.K., Siu K.-Y.: *"A New Bluetooth Scatternet Formation Protocol"*. Proc. IEEE Global Telecommunications Conference (GLOBECOM'01), 2001, 2864-2869. `http://perth.mit.edu/ ching/pubs/ScatternetProtocol.pdf`.
4. Misic V.B., Misic J.: *"Performance of Bluetooth Bridges in Scatternets With Exhaustive Service Scheduling"*. Proc. 36th Hawaii International Conference on System Sciences (HICSS'03).
5. Salonidis T., Bhagwat P., Tassiulas L., LaMaire R.: *"Distributed topology construction of Bluetooth personal area networks"*. Proc. IEEE INFOCOM 2001, Vol. 3 (2001) 1577-1586.
6. Tan G., Miu G., Guttag J., Balakrishnan H.: *"An Efficient Scatternet Formation Algorithm for Dynamic Environments"*. IASTED International Conference on Communications and Computer Networks (Nov. 2002)
7. Wang Z., Thomas R.J., Haas Z.: *"Bluenet - a new scatternet formation scheme"*. Proceedings of the 35th Annual Hawaii International Conference on System Sciences (Jan. 2002). `http://wnl.ece.cornell.edu/Publications/hicss02.ps`.
8. Zaruba G.V., Basagni S., Chlamtac I.: *"Bluetrees-scatternet formation to enable Bluetooth-based ad hoc networks"*. Proc. IEEE International Conference on Communications (ICC 2001), 2001, 273-277.