Bluetooth Scatternet Tree Formation for Wireless Sensor Networks

Michael Methfessel, Steffen Peter and Stefan Lange
IHP, Im Technologiepark 25, D-15236 Frankfurt (Oder), Germany
{*methfessel, peter, lange*}*@ihp-microelectronics.com*

***Abstract*—Wireless sensor and actor networks based on Bluetooth technology require an algorithm to build up and maintain a Bluetooth scatternet. Although much research has been done, actual deployment is rare because a self-organizing and self-healing procedure suitable for actual Bluetooth nodes has not yet been developed. We present the SFX algorithm to build a scatternet tree with the required properties. The method was verified by simulations for up to 400 nodes and by field tests.**

***Keywords*-Bluetooth, scatternet, sensor networks**

I. Introduction

Since its introduction in the 1990's, Bluetooth has become a widespread technology for wireless personal area networks. Typical applications replace wired links over short distances, e.g., between a mobile phone and a headset, or a PC and keyboard. A completely different type of application connects an unlimited number of nodes into a "scatternet". This is built on the basic unit of Bluetooth communication, a "piconet" of maximally eight nodes. A scatternet is a collection of piconets, with some nodes member of two or more piconets, arranged in such a way that any two nodes can communicate via multiple hops over intermediate nodes.

Theoretically a simple concept, a scatternet is surprisingly tricky to build up. Numerous applications in industry, environmental monitoring, and sensor networks would benefit from a robust algorithm to set up and maintain a scatternet. The task is demanding: a large number of independent nodes must self-organize into a coherent network with 100% reliability, operating within the somewhat quirky features of the Bluetooth system. Specific issues are:

- Nodes must discover other nodes in the neighborhood, using the Bluetooth procedure "inquiry" which is is slow and interrupts normal data traffic.
- The Bluetooth piconet structure must be obeyed (a single master, communication only between master and slaves). Different solutions can be found, none of which are especially straightforward.
- A self-healing property is required. As communication links are temporarily blocked, nodes fail, or new nodes enter, the network must reconfigure to maintain scatternet integrity.
- Depending on the application, trade-offs exist between power consumption, latency, network robustness, and throughput.

In this paper, we consider large networks of up to several thousand nodes. This leads to additional requirement:

- It is not feasible to collect information about the nodes centrally and to construct the network based on this knowledge.
- In general nodes will not be within radio range of each other.

A final set of requirements arises when commercially available Bluetooth devices are used. These generally have additional restrictions compared to the standard. Specifically, we consider the BlueCore implementation from CSR [1].

- The standard allows a Bluetooth device to be a member of any number of piconets (albeit master in only one). In practice, a device can be in maximally two piconets.
- The standard allows each master to control up to seven slaves. In practice, performance in a sactternet degrades strongly for more than 3 to 4 slaves.

In this paper, we present the SFX algorithm for constructing and maintaining a scatternet which satisfies all the listed requirements. It was developed to network the collectors in large solar energy farms (leading to the name SFX). This application is characterized by an unlimited power supply, targets for the data traffic, and latency requirements. However, the final algorithm is of general applicability to large scatternets, possibly modified to some extent. Our approach is based on the published SHAPER algorithms, but adds crucial elements required for use in real-life environments. We discuss relevant aspects of Bluetooth and previous work in Section II. Section III presents the SFX algorithm and Section IV briefly discusses the implementation and evaluation. Section V gives the conclusions.

II. Related Work

A. Summary of relevant Bluetooth features

Bluetooth is a short-range wireless system based on frequency hopping, characterized by low cost and robustness. Any Bluetooth connection lies within a piconet of up to eight nodes. The piconet is coordinated by one node (the master) whereby data can only be transfered between it and the slaves. A scatternet is a collection of piconets, with "bridge" nodes lying in two piconets. The bridge node can be master in one piconet and slave in the other, or it can be slave in both. For a given link, either node can request a role switch, which may or may not be granted by the peer node.

Every Bluetooth device has a unique 6-byte device address, generally denoted by `BD_ADDR`. A Bluetooth application desiring to control connection setup, termination, and master/slave roles can be built on the event-driven host controller interface (HCI), which is described in detail in the 1700 page Bluetooth specification [2].

An important issue is device discovery, handled by the "inquiry" and "inquiry scan" modes. One device is in inquiry, repeatedly sending a special inquiry data packet and waiting for a response, while the second device is in inquiry scan, listening for the inquiry packet and responding. Inquiry typically is done for some seconds or until enough responses have been collected. Inquiry scan is done periodically and is invisible to the application; default is a scan of 11.25 ms every 2.56 s. The inquiry procedure is hopped over 32 dedicated frequencies, puzzlingly split into two trains of 16. Even in error-free transmission conditions, the time for discovery can be up to 10.24 seconds [2] for the default configuration. A number of papers have studied the discovery process (see for example [3]). In practice, a central ingredient is to use an "interleaved" inquiry scan. The scan then takes 20.5 ms and discovery is highly probable if the the inquiry and inquiry scan on the two devices overlap.

By default, the result of discovery is essentially the `BD_ADDR` of the remote node. The SFX algorithm relies heavily on the fact that, optionally, a 240-byte payload can be returned in an "extended inquiry response" (EIR). By this means nodes can inform others of their status in their tree before a connection is established.

A node in inquiry does not participate in normal data transfer; data packets are queued until the inquiry mode is ended. This can lead to large latencies in multi-hop networks if nodes go into inquiry frequently.

B. Previous work on scatternet formation

Scatternet formation has been intensely studied and good surveys of the results exist [4]. Overall, most algorithms can be classified as mesh-based or tree-based (although star and ring topologies have been considered). A mesh allows multiple routes between nodes, while a tree is loop-free with a only single route for each node pair. Advantages of a mesh are that the inherent redundancy handles disrupted links, while a tree must be repaired. Also, load balancing or latency reduction can be done by appropriate routing. On the other hand, a mesh topology requires effort and overhead for routing and inter-piconet scheduling. Examples for mesh-building algorithms can be found in [5] and [6].

Since SFX is a tree-building algorithm, we focus on related previous work. A variety of approaches have been proposed, such as TSF [7], Bluetrees [8], and SHAPER [9]. The topic here is to find an algorithm for large systems which can be implemented with existing Bluetooth hardware.

One issue is decentralization. For example, the BTCP (Bluetooth Topology Configuration Protocol) [10] is a centralized approach in which a coordinator determines optimal piconets and sends lists of children to the masters. Indeed, if the coordinator has all required information this leads to an optimal topology. However, this approach is not feasible for large systems due to bad scalability and overhead for collecting/distributing information. An algorithm is required which runs independently on all nodes in parallel.

A second issue is node-to-node visibility. A simple and effective way to build a tree is the TSF algorithm [7]. Trees merge via their roots only, with one becoming root of the merged tree. While this is definitely simpler than SFX, it assumes that all nodes are within communication range of each other. For a large systems, this cannot be assumed.

A more realistic tree-building approach is SHAPER [9]. This is a distributed self-healing algorithm producing multi-hop Bluetooth scatternet trees, compatible with a limited communication range. This algorithm is the basis for the work in this paper. However, to the best of our knowledge none of the approaches has been applied in practice for large networks. The evaluation typically focuses on simulations under ideal conditions. making impractical assumptions such as time synchronization, visibility of all nodes, or the neglect of the consideration of concurrency. In SFX, the inclusion of realistic conditions have lead to a coherent and verified solution.

III. Tree formation and repair

The purpose of the SFX algorithm is to build up and maintain a tree of Bluetooth nodes by defining a suitable piconet structure. The term ”tree” includes the master-slave relation along each link. A node may be member of up to two piconets. An ”intermediate” node participates in two piconets: it is slave in the piconet coordinated by the master above it in the hierarchy, and it is master of the piconet containing its own children. There are also ”leaf” nodes (without a master role) as well as a single ”root” node (master role only). The master/slave relation will be represented by an arrow from master to slave (see Fig. 1).

We seek a decentralized algorithm which can be iterated until all nodes have merged. In the general case, two intermediate nodes have decided to merge their trees by creating a connection. This cannot be done directly: whichever node is slave in the new connection would become member of three piconets. However, we cannot avoid this case: due to the limited radio range, this could be the only way in which the trees can merge. Following SHAPER, we reconfigure one of the trees by means of master-slave role switches until the root has moved to the node intending to create the new connection. This node now lies in only one piconet (which it manages) and can create the new connection, as long as becomes slave. The whole procedure can be continued indefinitely, since each merge results in a single tree with an identical type of structure.

Although we adopt the central idea of SHAPER, a number of other features of that algorithm are not practical and lead to unnecessary complexity. Furthermore, procedures must be added for effective locking, to ensure that a loop-free geometry results, to optimize the topology, and to obtain rapid tree self-healing for disrupted links. We discuss these questions below.

A. Basic tree-merging procedure

On each node, we characterize its role by the ”tree identifier” (the Bluetooth device address of the root node as in SHAPER) and its depth in the tree. In addition, each node maintains a list of local connections and its master/slave role in each.

The algorithm starts with node discovery, by means of the inquiry state entered at random intervals. The node accumulated state information from neighbor nodes by collecting extended inquiry results (EIRs). A node returns its current tree identifier, the depth in its tree, the number of connections, and the number of descendants. The active node then analyzes the responses and decides whether to merge trees (described next) or to reduce depth (described later). The exact

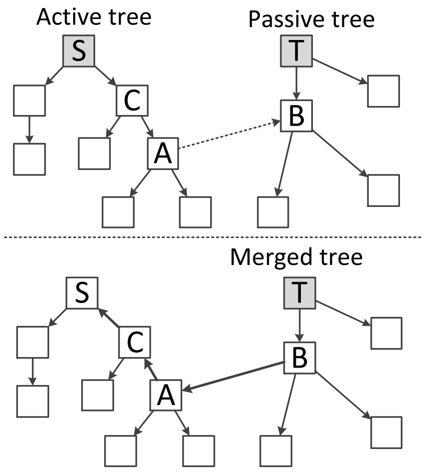

Figure 1. Top: two trees before node A initiates merging with B. After the active tree reconfigures to make A root, node A sends a connection request to B. Bottom: the merged tree where A is slave in the new connection and T is root of the merged tree.

preconditions for a merge are considered later; at the minimum, the two nodes must lie in different trees.

An important distinction to SHAPER is that the SFX merging procedure is highly asymmetrical. Considering Fig. 1, node A has decided to merge with node B, indicated by the dashed line. At this stage, B is completely passive at the application level (the EIR was sent automatically by the Bluetooth system) and has no knowledge of A's intentions. The ”active” left-hand tree must now reconfigure to make A the root. Node A sends a ”reconfigure request” message upstream to its master C. This node passes the message on until the root S is reached. The root node (even when moving) continually maintains a lock to ensure that only one merge operation is active at a time. If the reconfiguration is refused, a ”reconfigure reject” message is returned. Node A then cancels the merge attempt.

Alternatively, if node S grants the reconfiguration, it sends a ”reconfigure OK” message to C. Node C initiates a role switch with S. When the switch is complete, node S has changed to intermediate node. Node C is temporarily root, sends a reconfigure OK message downstream to A, waits for the role switch to be initiated by A, and is then again an intermediate node. Node A has now become root and initiates the connection to B. The connection will be set up with an immediate role switch: node B will becomes master, A will be slave.

At this point, node B for the first time notices the merge attempt as the connection request arrives. It rejects it if it cannot accept more children (say, some other new child arrived while tree A was reconfiguring).

Otherwise, it accepts and sends its current tree id and depth over the new connection. Node A increments the depth, sets it own tree id/depth, sends a tree id message to all children, and unlocks. The merge procedure is now complete. However, the tree id message might still propagate some time down the subtree. Each node increments the depth, sets its own tree id/depth, and sends the message on to its children.

Noteworthy aspects are:

- The merge procedure is asymmetrical. The active tree containing A has locked, done a lengthy reconfigure operation, actively initiated the connection, and distributed the new tree identifier to its sub-tree. Node B has merely responded to the connection request and sent its tree id over the new link. No locks were set and no information was distributed. This is a major advantage for parallel tree-building in all parts of the system. Any number of other trees can merge with tree B at the same time. Also, we avoid the badly-scaling distribution of information in the ever larger target tree.
- There is only a single case to be handled, as opposed to four separate merge procedures in SHAPER. This makes an implementation of our algorithm easier.

In case node A finds its connection request refused by B, tree A has reconfigured in vain and the tree id/depth information is not consistent. Node A performs a "grab" to inform all descendants that is is now root. It sends a tree id message to its children which then propagates as described. No attempt is made to determine the time when all descendants have received the message and set their tree id/depth correctly. This is discussed further in the context of tree self-healing.

B. *Locking and loop avoidance*

Additional measures are still needed to prevent loop formation. Locking ensures that maximally one merge attempt is done per tree, but parallel merges from tree A to tree B and from tree B to tree A can occur. A conceivable solution is to implement some analogous locking procedure on tree B. This would lead to a severely increased complexity with badly-scaling delays and race conditions. Worst of all, it sacrifices the property that many trees can merge into tree B simultaneously.

We have therefore adopted a known device for such issues: a tree only merges if its own tree id can be reduced. This leads to a robust procedure for large systems, allowing distributed merges in parallel. For example, it can happen that some other node in tree B has decided to merge with a third tree X while tree A was reconfiguring. This only means that the tree id finally presented to A is even lower than expected. Depending on the timing, tree A might receive several tree id messages, each lowering the tree id further. In every case, a valid loop-free tree will result.

Clearly, the final root node is preordained, namely the one with the lowest Bluetooth device address. This node will never actively merge, and all other nodes will join its tree eventually. This feature is used to choose a specific node as root of the final tree. This node (and no other) sets a flag which is part of the tree identifier (together with the aforementioned Bluetooth device address). The comparison between two tree identifiers is defined so that a flagged tree id always comes out lower. In our application, this node is the gateway by which collected data is transferred to the Internet. In the next subsection, the tree is optimized to reduce the depth relative to this specific node.

Whatever node happens to be root, reconfigure requests walk upstream until they hit this node, which then returns the accept or reject decision. There is an obvious consistency problem here. After every merge, new tree id/depth information is flooded downward, but a node can never be sure that its current tree id/depth information is up to date. The issue is important: a node which bases its decision to merge on stale data might set up a parallel connection to a remote tree and create a loop. Some kind of acknowledgment scheme to verify consistency over the whole tree is not feasible in practice. The best solution is to include the current tree id of the originating node in each reconfigure request. The node which detects a discrepancy between its own tree id and that in the request immediately sends back a reject message to cancel the merge attempt. This device ensures that nodes only perform a merge when their tree id is correct.

C. *Tree optimization*

Depending on the application, the tree could be constructed to balance the load on the nodes, to conserve energy, or to maximize the throughput etc. Here it is of interest to minimize the number of hops between the root (gateway) node and any other node. A simple optimization strategy was used as follows. When a node has joined the target tree (i.e., with flagged tree id) it begins to optimize to reduce its depth. The list of EIRs obtained during inquiry is inspected to find a peer node which is at least two steps higher in the tree and can still accommodate another child. In that case, the node disconnects from its current master and builds a new connection to the peer node. Simulations have shown that this simple technique effectively reorganizes the tree to a suitable topology. At that point, no node can further reduce its depth by local optimization.

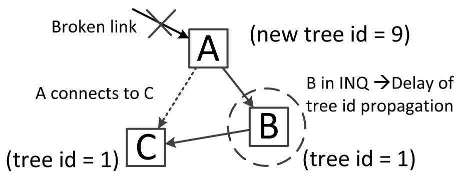

Figure 2. Example for loop formation after grab.

The peer node might reject the incoming connection request, e.g., because it has reached the maximal number of children. In that case, the already-merged tree has been split and must be re-merged. The node attempting to reduce its depth is now root of the split-off tree and performs a grab as described above. After that, the procedure is handled by the self-healing behavior, described next.

D. *Tree self-healing*

When communication over a Bluetooth link breaks down, both nodes are informed by the HCI. The node which has lost its master has become root of the split-off subtree. It performs a grab to inform its descendants of the new tree id. From that moment, all nodes in the subtree again initiate tree merging when a suitable peer is found in inquiry. This is the essence of the self-healing behavior: all nodes periodically probe their environment by means of inquiry and the collection of EIRs. In a stable system, each inquiry phase leads to the result that no action must be taken. As soon as the EIRs show the necessity to merge or reduce depth, these procedures are triggered.

Although tree-healing is in principle built into the algorithm, a specific problem can arise. By construction, whenever the tree splits and a grab is done, the tree id in the split-off subtree increases. Therefore, the automatic robustness which was present during tree building is broken. Consequently loops can arise in some cases and must be handled.

For example, in Fig. 2 node A has lost its master by a broken link and has grabbed the split-off tree by sending a tree id message to its child B. Because B is in inquiry, the message is delayed for some seconds. During this period, A goes into inquiry itself and receives stale information from C, since this node still believes itself to be in the tree with lower id. Node A consequently merges with node C, thereby creating a loop. These cases occur in practice but no really clean solution could be found. As simple but effective fix, node A detects the loop when the tree id message it has sent to B loops back. The node then disconnects all links to its neighbors and proceeds with tree merging after a pause of a few seconds.

IV. Implementation and Evaluation

Figure 3 illustrates the event-driven state machine for the active node A of Fig. 1. As mentioned, peer node B is mainly passive, responding to the connection request and returning its tree id/depth over the new connection.

In **NORMAL**, the node handles connection requests and data traffic, sends inquiry responses, and takes part in reconfiguration operations by other nodes in the tree.

After a random period, the node goes into **INQUIRY** and builds up a list of neighbors. This list is analyzed and can initiate one of three procedures:

- If A wants to merge trees and is root, it does a connection request and goes to **CONNECTION**. When the connection is established and the new tree id was received, A distributes the tree id to its descendants and goes to **NORMAL**.
- If A wants to merge trees and is non-root, it sends a reconfigure request upstream and goes to **REQUEST CONFIGURE**. If the reconfiguration is granted, A does a connection request and goes to **CONNECT** as in the previous case. Otherwise the merge attempt is canceled.
- If A lies in the target tree and can reduce its depth by becoming child of the peer node, it disconnects from its master in **DISCONNECT**, then does a connection request and goes to **CONNECT** as above.

If none of these procedures is appropriate, A returns from **INQUIRY** to **NORMAL**: the tree is stable.

The algorithm was implemented over the HCI interface. A surprise during simulation was the unexpected complexity of scatternet-building when all nodes play an independent and active role. The system is highly dynamic; when a node merges with another, it is probable this node is in some other state when the procedure is completed. Repeated cycles of implementation, simulation, and analysis led to the set of rules presented in the previous sections. Verification and development

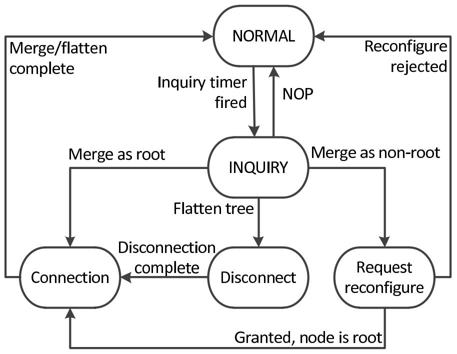

Figure 3. Implemented state machine for the merging node.

was done for several hundred different tree-building simulations for various geometries and system sizes up to 400 nodes. The final algorithm delivers a valid and optimized tree in every case, requiring approximately 180 second for 400 nodes. The simulation environment is an in-house development which allows the software to be run directly over the Linux BlueZ stack. Several hundred test runs in a system with 30 Bluetooth nodes delivered a correct tree each time in the expected timer frame. Currently, the algorithm is performing equally in a solar power station with 30 nodes, with a new test done each morning as the sun rises. At the time of writing, it is under deployment in a newly-built larger plant. A thorough presentation of the simulations and real-life measurements is deferred to a future publication.

V. Conclusion

This work was driven by a real-life industry application, which dictated the use of Bluetooth technology as well as the ultimate scaling to thousands of nodes. A survey of the state-of-the-art for scatternets seemed to indicate the SHAPER algorithm as the most promising approach. Unfortunately, it emerged that SHAPER cannot be implemented, even though the basic idea is sound. SHAPER merges trees by forming a link, exchanging data, then deciding how to reconfigure. This contradicts the rules for e.g. the widespread CSI Bluetooth devices [1] restricting nodes to membership in two piconets. Our innovative solution is to use the extended inquiry response (EIR) to pass critical information between nodes in a broadcast fashion, bypassing the communication links. This allows decisions about reconfigure/merge operations to be made before the procedure starts. SHAPER postulates four different merge procedures. This complexity is difficult to realize in the event-driven Bluetooth HCI and is unnecessary: our analysis identified a unified asymmetrical procedure with no need to distinguish between ”free”, ”root”, and ”non-root” nodes. SHAPER correctly identifies the need for a locking mechanism to avoid loop formation. Their suggested approach (a lock maintained by the root) was also the basis for our algorithm. However, research showed numerous problems with race conditions, in particular during the distribution of data in a tree. Our solutions were presented in Section IIIB. Finally, SHAPER requires the maintenance of the tree cardinality over all nodes. Thus, whenever two trees merge, this number must be calculated and distributed; when a node leaves a tree, the same must be done. No solution was given for the associated race conditions and scale issues. To defuse this and related issues, our algorithm introduces a central requirement: the active role in the merge procedure is only taken if the node can lower its tree id by merging. This yields a highly robust mechanism, by which independent merges can take place at different positions with no risk of loop formation. Finally, we present a straightforward but successful method for reducing the tree depth. In parallel to tree merges, nodes disconnect and reconnect within their tree to reduce their depth. After a few steps, each node finds a stable position and remains there with no sign of oscillatory behavior.

We believe our work presents the first successful implementation of Bluetooth scatternets in realistic industrial environments. It opens the door to a multitude of similar applications.

References

[1] Cambridge Silicon Radio, “Bluecore scatternet support,” available from http://www.csr.com/.

[2] Bluetooth SIG, “Bluetooth specification v3.0,” 2009.

[3] J. Jiang, B. Lin, and Y. Tseng, “Analysis of bluetooth device discovery and some speedup mechanisms,” *Journal of the Institute of Electrical Engineering*, vol. 11, no. 4, pp. 301–310, 2004.

[4] R. Whitaker, L. Hodge, and I. Chlamtac, “Bluetooth scatternet formation: a survey,” *Ad Hoc Networks*, vol. 3, no. 4, pp. 403–450, 2005.

[5] C. Petrioli, I. Chlamtac, and S. . Basagni, “Bluemesh: degree-constrained multi-hop scatternet formation for bluetooth networks,” *Mobile Networks and Applications*, vol. 9, no. 1, pp. 33–47, 2004.

[6] C. Lim, M. Huh, C. Choi, and G. Jeong, “Gbsfp: General bluetooth scatternet formation protocol for ad hoc networking,” *IEICE transactions on communications*, vol. 93, no. 1, pp. 40–46, 2010.

[7] G. Tan, A. Miu, J. Guttag, and H. Balakrishnan, “Forming scatternets from bluetooth personal area networks,” *Massachusetts Institute of Techonology, http://lcs. mit. edu/, Tech. Rep. MIT-LCS-TR-826*, 2001.

[8] G. Zaruba, S. Basagni, and I. Chlamtac, “Bluetrees-scatternet formation to enable bluetooth-based ad hoc networks,” in *Communications, 2001. ICC 2001. IEEE International Conference on*, vol. 1, 2001, pp. 273–277.

[9] F. Cuomo, G. Di Bacco, and T. Melodia, “Shaper: a self-healing algorithm producing multi-hop bluetooth scatternets,” in *IEEE GLOBECOM’03*, 2003.

[10] T. Salonidis, P. Bhagwat, L. Tassiulas, and R. LaMaire, “Distributed topology construction of bluetooth personal area networks,” in *IEEE INFOCOM 2001*, 2001.