Improving Connectivity in a Scatternet Formation Algorithm

Cristina L. Abad
University of Illinois at Urbana–Champaign
cabad@uiuc.edu

Matthew J. Miller
University of Illinois at Urbana–Champaign
mjmille2@uiuc.edu

Abstract

Scatternets are ad hoc networks formed by Bluetooth devices. They are formed by connecting smaller networks known as piconets. In this paper, we present and evaluate a Tree Scatternet Formation with Tokens (TSF–T) protocol to form scatternets efficiently. This protocol, which extends existing algorithms, allows devices to be out–of–range and dynamically enter and leave the network. In addition, TSF–T provides quick scatternet formation and almost always merges all disjoint networks into one scatternet. This scheme increases the efficiency of the devices that bridge piconets by reducing the master/slave breeches and gives the routing protocol more freedom to choose efficient paths. The paper also describes a method to allow two nodes in a tree to enter a critical section which could be used to assure only two nodes are transmitting certain data at any given time in a communication network.

1 Introduction

Issues in ad hoc networks have been one of the richest areas of research recently. One area of interest is how to form topologies efficiently so that routes are established while battery power is conserved. Bluetooth technology [1] has generated interest in this area because of its potential in the commercial market. The Bluetooth interface, described in Section 1.1, makes topology formation even more difficult because of the extra constraints placed on how devices may form a network. These constraints are necessary due to the limited number of channels devices use for communication and scarce battery power characteristics of most Bluetooth devices.

1.1 The Bluetooth Interface

The Bluetooth Special Interest Group [1] was founded to develop a technology that could be used to form short–range, wireless, ad hoc networks. Bluetooth provides communication between a variety of electronic devices via low power, low cost radio chips.

Devices communicate by using spread spectrum frequency hopping. There are 79 channels available for communication within the 2.45 GHz band. As the number of Bluetooth devices in an area increases, there must be ways to efficiently share the limited resources. To deal with this issue, the group defined two logical arrangements to better utilize the communication channels.

A *piconet* consists of one master device and up to seven slave devices. All members of a piconet share the same channel and communication is regulated by the master. The maximum number of slave devices is bounded to limit device overutilization. There is no direct communication between slaves, so all data must first pass through the master. To form piconets, devices first enter an INQUIRY or INQUIRY SCAN state. These states correspond to a device sending packets in an attempt to discover other devices (INQUIRY) or listening for INQUIRY packets (INQUIRY SCAN). Since there is a wide range of possible frequencies for this state (32 for the US and most of Europe), the sender and listener hop frequencies at different rates. The sender then must determine the listener's identity by entering the PAGE state. Similarly, the listener will enter the PAGE SCAN state to respond and eventually form a connection. More detail about the time associated with these states can be found in [2, 3, 4]. It suffices to say the inquiry time is the dominating phase of the connection process.

The second logical arrangement is a *scatternet* which allows many devices to communicate. Scatternets are composed of piconets which share a device. This shared entity multiplexes its time between the piconets. It is obvious that the method by which scatternets are formed can have a large impact on the performance of the ad hoc network. However, the Bluetooth Specification does not provide any protocol for scatternet formation. In Section 1.2, we will discuss some attempted solutions to this problem.

1.2 Related Work

In [4], the Bluetooth Topology Construction Protocol (BTCP) is presented. In BTCP, a scatternet is formed by the devices first choosing one leader via an election algorithm. This leader then determines how many masters are necessary to minimize the number of piconets. Masters and bridges are then assigned by the leader subject to constraints specified in the election phase. This approach minimizes the number of piconets, allows roles to be chosen efficiently (since constraints are specified) and requires no a priori knowledge of other devices. Another contribution of the paper is the idea that devices randomly enter the INQUIRY or INQUIRY SCAN state during the election process. This is an important part of the algorithm discussed in Section 2.1. BTCP, however, suffers greatly in scalability as the scheme is only designed and analyzed for up to 36 devices. Another downside is the algorithm is highly dependent on the interarrival time of the devices in the election process. The greater the differences in device arrival times, the worse the algorithm will perform.

In [5], a protocol is presented which achieves $O(\log n)$ time for scatternet formation. The formation process resembles the election phase of BTCP, but connects the devices while finding leaders. The protocol is more dynamic with piconets merging when possible and devices migrating to different piconets under certain circumstances. This approach does not suffer from the scalability or interarrival time issues of BTCP.

Both of these approaches operate under unrealistic assumptions. First, it is assumed all devices are within range of each other. There could be scenarios where devices are out–of–range, but could still participate in the same scatternet by bridged piconets. Another shortcoming is the algorithms do not deal with devices leaving the piconets. In Section 2.1, a Tree Scatternet Formation (TSF) protocol is presented that addresses these issues.

The contributions of our paper are:

- Extend Raymond's Tree Algorithm [6] to allow two nodes in a tree to enter a critical section at one time. This could have applications in networks where it is desired that exactly two devices are communicating some specified data at any given time.

- Improve the connectivity of TSF to form larger scatternets in some situations where the original protocol would result in disjoint scatternets.

- Decrease the time it takes for the protocol to form one scatternet in which all nodes can communicate with each other.

- Allow some bridges in the scatternet to be configured as slave/slave in the piconets they connect. This reduces the master/slave breeches in the scatternet. This concept is further explained in Section 2.

- Consider device constraints, such as mobility and battery power, when forming scatternets.

The rest of this paper is organized as follows. In Section 2, some desirable properties of scatternet formation are enumerated and the TSF protocol is described. In Section 3, our extensions to TSF are described. In Section 4, the new protocol is simulated and its performance compared to the original. In Section 5, directions of future work are suggested. Section 6 concludes the paper.

2 Scatternet Properties

From the previous section, it is easy to see there are many issues that arise when attempting to form scatternets. Since Bluetooth was designed primarily for wireless settings, device resources may be scarce, and hence, overutilization can be a major problem. Therefore, scatternet formation protocols should adhere to certain design principles. Some of the properties an ideal scatternet formation should include are:

Minimize the Number of Piconets This allows easier network control and reduces the number of collisions due to shared channels. This also minimizes the number of bridge devices. Communication time will increase with the number of bridges because for every piconet a packet passes through, it must wait for the bridge to be multiplexed.

Bridges Connect Two Piconets Switching between piconets is a heavyweight procedure since devices must change the channel they are using. If a bridge was involved in a large number of piconets, the multiplexing overhead would dominate communication time and delay other packets waiting for the bridge. Additionally, requiring a bridge to relay messages for more than two piconets could overload the device.

Connectivity Every device should be able to communicate with other devices in the scatternet. If two scatternets are within range of each other by at least one device, they should merge to achieve maximum connectivity.

Dynamic Configuration Scatternets should adjust when nodes enter and leave at random times.

Network Diameter The maximum number of hops between any two nodes in the scatternet should be minimized. Increasing the number of hops will cause more delay and resource utilization for sending packets.

Reduce Contention Similar to the problems caused by the network diameter. If other piconets are frequently using a specific piconet for a relay, a significant portion of the piconet master's time will be devoted to bridging these requests rather than controlling communication within the piconet.

Avoid Master/Slave and Master/Master Bridges Since the master must control all communication in a piconet and slaves cannot exchange data without the master, it is desirable to form scatternets such that bridges are slaves in all their piconets. Otherwise, the master will have to switch between piconets which will degrade the amount of inter–piconet communication.

Consider Device Constraints Bluetooth spans a wide variety of computing devices from hand–held PDA's and cell phones to workstations and printers. The scatternet formation should attempt to avoid placing devices with scarce resources (e.g. battery power) in a location where utilization may be heavy. In Bluetooth, masters will have higher utilization than slaves. Also, the utilization will increase with the number of connections to a device. Another possible constraint in Bluetooth is mobility. Devices with greater mobility will have a higher probability of link failure and thus should not have as many connections as a stationary device.

The TSF protocol, discussed in more detail in Section 2.1, addresses the dynamic configuration property and improves the connectivity of previous algorithms by allowing out–of–range devices to exist in the same scatternet. Table 1 offers a comparison of some existing scatternet formation algorithms with respect to these properties.

2.1 Tree Scatternet Formation

The original TSF algorithm for determining a device's state is shown in Figure 1. A device can be one of three types: ROOT, NON–ROOT, or FREE. Every device begins as a FREE node and alternates states accordingly.

Algorithm	Strengths	Weaknesses
BTCP	Minimizes the number of piconets. Allows formation according to device constraints.	Not scalable. Does not handle devices entering and leaving after initial setup. All devices must be in–range.
Law *et al.*	Forms diameter of $O(\log n)$. Allows devices to enter dynamically.	Does not handle devices leaving. All devices must be in–range. Does not consider device constraints.
TSF	Devices can be out–of–range. Allows dynamic configuration.	Two scatternets may not connect if ROOT's are out–of–range. All bridges are master/slave. Does not consider device constraints.

Table 1: Scatternet Formation Algorithm Properties

```
TSF()
 1  while true
 2  do if $last =$ FORM:INQUIRY
 3        then $state \leftarrow$ FORM:INQUIRY-SCAN
 4             $last \leftarrow$ FORM:INQUIRY-SCAN
 5        else $state \leftarrow$ FORM:INQUIRY
 6             $last \leftarrow$ FORM:INQUIRY
 7     $t \leftarrow \textsc{Random}(E[T_{inq}], D)$
 8     remain in $state$ for $t$ time
 9     if $type =$ ROOT
10        then switch to other FORM state
11             $t \leftarrow \textsc{Random}(E[T_{inq}], D)$
12             remain in $state$ for $t$ time
13     $t \leftarrow f_{comm} \times \textsc{Random}(E[T_{inq}], D)$
14     if $t > 0$
15        then $state \leftarrow$ COMM
16             remain in $state$ for $t$ time
```

Figure 1: Original TSF Algorithm

The f_{comm} function on line 13 of TSF is defined as:

$$f_{comm} = \begin{cases} 0 & \text{if } type = \text{FREE} \\ d & \text{if } type \neq \text{FREE and } A < \text{threshold} \\ A \times d & \text{otherwise} \end{cases}$$

where A is how long the device has been in the scatternet and d is the degree, or number of connections of a device. This is designed such that FREE devices will devote all their time to the FORM stages. The function also allows devices with more connections to spend a larger portion of time in the COMM stage. The $E[T_{inq}]$ value in the algorithm is the expected time of the INQUIRY phase in Bluetooth. The D value is an upper bound on how long a device can spend in a state. These values are not provided in the TSF description.

The FORM:INQUIRY and FORM:INQUIRY–SCAN states correspond to a device entering the Bluetooth state of INQUIRY or INQUIRY SCAN, respectively. When two devices discover each other they enter PAGE and PAGE SCAN to form a new connection. However, not all nodes are allowed to join with each other. TSF allows the following types to join:

FREE and FREE One device becomes the master and the other a slave in a newly formed piconet. The master becomes the ROOT and the other device NON–ROOT.

ROOT and ROOT One ROOT becomes a NON–ROOT and joins the other device's piconet as a slave.

NON–ROOT and FREE The FREE node becomes a slave and the NON–ROOT a master in a newly formed piconet. The FREE node updates its type to NON–ROOT.

It is proven that this scheme will produce loop–free topologies that consist of a forest of trees. Therefore, if one scatternet is formed, the protocol will produce the minimum number of links necessary to guarantee a path between any two nodes. This is desirable in the Bluetooth environment because each link represents a connection and increased connections will result in more power consumption.

The protocol places stringent rules on what type of devices can join. This can limit connectivity in the situation where nodes are out–of–range of each other. Consider the situation where two separate scatternets, S_1 and S_2, have been formed. Let u and v be NON–ROOT devices in S_1 and S_2, respectively, that are in–range and involved in only one piconet apiece. If the roots of S_1 and S_2 are out–of–range, then the scatternet will never become fully connected even though this could be achieved by forming a link between u and v.

Another area where TSF may perform poorly is contention. Assuming the tree formed by TSF is almost balanced and communication between any two devices is equally likely, the piconet of the root will devote a majority of the time to forwarding packets between its subtrees. Since the root spends approximately twice as much time in the FORM stages as other devices, the communication time will be further limited.

This protocol also makes each bridge a master/slave in its piconets. This means the number of master/slave devices is proportional to the number of piconets. It would be more desirable to have the potential for slave/slave bridges since there is no fundamental reason why master/slave bridges are necessary.

Finally, TSF assumes a tree topology will be the most efficient formation. However, if a small number of cycle–forming links are introduced, there could be potential benefits. First, the network diameter could be reduced by linking devices that are several hops from each other. Also, this would give the routing protocol more freedom in choosing the path for a packet. If path becomes congested or resources are overutilized, the routing protocol could discover alternate paths.

3 TSF with Tokens

In this section we present our protocol which improves the existing TSF version. The design starts by recognizing that by restricting TSF to only one node per scatternet that has the ability to merge trees, the connectivity and scatternet formation time of the protocol is restricted. Therefore, we propose allowing all eligible devices the ability to potentially merge trees.

In Bluetooth, 64 Dedicated Inquiry Access Codes (IAC) are defined for the INQUIRY process. Devices will only react to others using the same IAC during the INQUIRY and INQUIRY SCAN processes. In TSF, the ROOT uses the Limited Inquiry Access Code (LIAC), while other devices use the Generic Inquiry Access Code (GIAC). This is the method by which a ROOT will only attempt to discover another ROOT. First, consider the scenario where the protocol maintains one ROOT per scatternet, but the ROOT dynamically changes. This is the classic problem of achieving mutual exclusion in a tree structure. A well–known solution to this problem is Raymond's Tree Algorithm [6]. A token is introduced to the scatternet and the token–holding device becomes the ROOT. The protocol developed is called *Tree Scatternet Formation with Tokens*, or *TSF–T*.

By introducing the token, connectivity is improved because tree mergers are no longer subject to the range of only one device in each scatternet. This design also reduces contention because there is not one device that must continually spend twice as much time in the FORM stage. However, by introducing tokens, message overhead is introduced and the protocol must assume some routing protocol exists.

3.1 Raymond's Tree Algorithm

The token passing algorithm is further described in this section since it is important to the TSF–T protocol. Raymond's algorithm is chosen because it provides mutual exclusion, requires small amount of state that is $O(k)$ for a k-ary tree and averages only $O(\log n)$ messages for a device to enter the critical section (where n is the number of devices in the tree). The algorithm is also fair in the sense that the token does not remain within one subtree when other devices are requesting it. Each device maintains a *holder* pointer, which simply points in the direction of the token in the tree. Each device also maintains

```
RAYMOND($e$)
switch
  case $e =$ request CS :
      if $request\_q$ is empty
        then send request to $holder$
      ENQUEUE($request\_q, myself$)
      return
  case $e =$ receive request message :
      if $request\_q$ is empty
        then send request to $holder$
      ENQUEUE($request\_q, requester$)
      return
  case $e =$ receive token :
      $next \leftarrow$ DEQUEUE($request\_q$)
      $holder \leftarrow next$
      if $next \neq myself$
        then send token to $next$
              if $request\_q$ not empty
                then send request to $holder$
        else $canAccessCS \leftarrow$ true
      return
  case $e =$ leave CS :
      if $request\_q$ not empty
        then $next \leftarrow$ DEQUEUE($request\_q$)
              $holder \leftarrow next$
              send token to $next$
              $canAccessCS \leftarrow$ false
      if $request\_q$ not empty
        then send request to $holder$
      return
```

Figure 2: Raymond's Tree Algorithm for Event e

a FIFO queue, $request\_q$, which keeps track of when the device or one of its neighbors requests the token. The outline of the algorithm is shown in Figure 2.

Introducing the token allows more devices the opportunity to merge trees. Next, consider the problem of forming some cyclic links in the tree. By generating a second token in the tree, such that a device can only hold one token at a time, the token holders can not only merge trees, but also bear the responsibility of creating these cyclic links. Later, the design of these cyclic links is further described. In this section, an extension to Raymond's Tree Algorithm is presented to allow for two tokens to exist in a tree.

Let tok_1 be the first token and tok_2 be the second token in the tree. First, the restriction is made that a device may be the token holder for at most one of the two tokens. The motivation behind introducing the second token is to allow exactly two devices access to the critical section at a time. Therefore, the design would not benefit by having one device hold both tokens. If such a scenario is desired, the restriction could be relaxed without affecting the algorithm.

The basic idea is to overlay two Raymond's Tree Algorithms running independently on the same tree. Each device maintains pointers to two token holders, $holder_1$ and $holder_2$. Additionally, two separate request queues are maintained for neighboring devices, $request\_q_1$ and $request\_q_2$. When a device wants to request a token, it first looks in both request queues to make sure it has not already requested one. If it has not requested a token, then with probability p, a message is generated requesting tok_1 according to Raymond's Tree Algorithm. With probability $1 - p$, tok_2 is requested. For our scenario, $p = \frac{1}{2}$ so each token is requested with equal probability.

There is another subtle adjustment to the algorithm. Notice that a device could be holding tok_1 which implies some state is set to indicate the device can enter the critical section. While in this state, tok_2 could pass through the device in transit. The algorithm must not allow such an event to cause the device to leave the critical section because this would cause an inconsistency. The device would be holding tok_1 but not be allowed access to the critical section. This change is shown with respect to tok_2 in Figure 3 and a corresponding change would be necessary for the event when tok_1 is received.

3.2 Extending TSF

Now a mechanism exists for passing two tokens in a tree, TSF can be modified to take advantage of this algorithm. Additionally, TSF–T will consider a device's *score* when forming a new connection. The *score* variable is an indication of how many connections a device should have. The exact definition of a device's *score* is an area of further research. We envision it could be a function of a device's battery

```
RECEIVE–TOKEN2()
1  $next \leftarrow$ DEQUEUE($request\_q_2$)
2  $holder_2 \leftarrow next$
3  if $next \neq myself$
4     then if $holder_1 \neq myself$
5              then $canAccessCS \leftarrow$ false
6           send token2 to $next$
7           if $request\_q_2$ not empty
8              then send request to $holder_2$
9     else $canAccessCS \leftarrow$ true
```

Figure 3: Modification to Raymond's Algorithm for Two Tokens

```
JOIN–FREE($n_1, n_2$)
1  if $n_1.score < n_2.score$
2     then make $n_1$ master
3     else make $n_2$ master
4  $n_1.holder_1 \leftarrow myself$
5  $n_1.holder_2 \leftarrow n_2$
6  $n_1.type \leftarrow$ ROOT
7  $n_2.holder_1 \leftarrow n_1$
8  $n_2.holder_2 \leftarrow myself$
9  $n_2.type \leftarrow$ ROOT
```

Figure 4: Join Procedure for Two FREE Nodes

power and mobility among other factors. Without loss of generality, let a low *score* indicate a device with plentiful power supply and low rate of mobility. As a device's *score* increases, its power supply decreases and rate of mobility increases. Each device must have at least one connection to join the scatternet. When forming additional connections, it is desirable to increase the degree of devices with a low *score* as opposed to those with a high *score*. Considering these factors, TSF–T must modify TSF's device joining procedure to handle tokens, consider the *score* variable and allow cyclic links.

Initially devices not involved in a scatternet are of type FREE, have a predetermined *score* value, both *request_q*'s empty and both *holder* variables set to *nobody*. When a device becomes a token holder, it is of type ROOT. Otherwise, it is FREE or NON–ROOT depending on whether or not it has any connections. When a device becomes ROOT, it will stay in this state for a random amount of time chosen uniformly from the range $(E[T_{inq}], D)$. This helps avoid possible synchronization effects. When a device is of type NON–ROOT, it will request one of the tokens with a certain probability after line 16 in the TSF pseudocode if it is eligible. A device is eligible to request the token if:

1. It is not holding a token
2. It has not requested a token
3. It is not slave/slave. In this situation the device cannot form a new piconet or join an existing one without becoming involved in more than two piconets.
4. It is not master/slave and the piconet for which it is master has the maximum number of slaves. It cannot connect and still be in at most two piconets.

Next, consider what changes must be made to the joining procedures in Section 2.1.

FREE and FREE If one of the devices has a lower *score*, allow it to become the master of the newly formed piconet. The *score* values are considered similar to the way TSF considers the types of a NON–ROOT device and FREE device when such a connection occurs. When the connection occurs, each device becomes a token holder (one for tok_1 and one for tok_2) and updates the opposite *holder* pointer to the other device. The pseudocode is shown in Figure 4.

NON–ROOT and FREE If the NON–ROOT is slave/slave, no connection can be established without violating the two piconet constraint. If the NON–ROOT is already the master of a piconet and does not have the maximum number of slaves, the FREE device should join this piconet. This helps minimize the number of piconets by utilizing existing ones when possible.

If the NON–ROOT is only in one piconet and is a slave, create a new piconet and the device with the smaller *score* is master. The FREE device should update both *holder* variables to point to the erstwhile NON–ROOT device. The pseudocode for the procedure is shown in Figure 5.

ROOT and ROOT The most extensive changes must be made in this case. First, if either device is slave/slave a connection cannot be established. Next, if both devices are already in two piconets, a connection is not possible. With these possibilities removed, there are four scenarios that need consideration:

1. If one device is master/slave and the other is master/–, join the first device's piconet if it is not full. Otherwise, a connection cannot be established.
2. If both devices are master/–, join the piconet of the device with the lower *score*. If this piconet is full, try to join the other device's piconet.
3. If both devices are slave/–, create a new piconet and make the device with the lower score master.
4. If one device is master/– and the other is slave/–, try to join the existing piconet. If it is full, start a new piconet where the erstwhile slave becomes the new master.

The algorithm for updating the tokens if the two ROOT's are from different scatternets is shown in Figure 6.

There are some other issues that must be considered for the case of two ROOT devices joining. The main consideration is how to handle the tokens. If the two devices joining are from separate scatternets (which can be determined if the distance between the devices is ∞), both of the tokens should be removed during the connection process. This maintains the invariant that any scatternet will have exactly two tokens. The proof of this is inductive. If both scatternets had two tokens before the merge, then there exist four tokens at the instant the merge begins. Two tokens must be present at the merging devices, so if both tokens are removed only two tokens must exist when the merge completes. If the devices are from the same scatternet, neither token should be removed. In either case, if the connection does not occur, both tokens should remain in circulation. A token is removed by making a device NON–ROOT, making the respective *holder* variable point to the other device and, if the respective *request_q* is non–empty, making a token request to the other device.

```
Join–Non–Root($n_1, n_2$)
 1   /* $n_1.type$ = NON–ROOT */
 2   /* $n_2.type$ = FREE */
 3
 4  if $n_1$ is slave/slave
 5     then return
 6  if $n_1$ is a master and its piconet is not full
 7     then  join $n_1$'s piconet
 8     else  if $n_1.score < n_2.score$
 9              then  make $n_1$ master
10              else  make $n_2$ master
11  $n_2.holder_1 \leftarrow n_1$
12  $n_2.holder_2 \leftarrow n_1$
13  $n_2.type \leftarrow$ NON–ROOT
```

Figure 5: Join Procedure for NON–ROOT and FREE Nodes

When tokens are removed from the tree there is another complication to consider. For two separate scatternets, S_1 and S_2, let device u_1 be $holder_1$ in S_1 and v_1 be $holder_1$ in S_2. This means the $holder_1$ variable in every device in S_1 and S_2 points to u_1 and v_1, respectively. Now, define u_2 and v_2 to be $holder_2$ in S_1 and S_2, respectively. If u_1 and v_1 merge the scatternets, both tok_1's are removed. This creates an inconsistency because all device's $holder_1$ is pointing toward u_1 and v_1 where no token exists. Additionally, some of the device's $holder_2$ will point to u_2 while others will point to v_2. The solution to this problem is devices which are both ROOT's can only join if one is $holder_1$ and the other is $holder_2$. The algorithm

```
UPDATE–TOKENS($n_1, n_2$)
1    /* $n_1.holder_1 = myself$ */
2    /* $n_2.holder_2 = myself$ */
3
4   $n_1.type \leftarrow$ NON–ROOT
5   $n_2.type \leftarrow$ NON–ROOT
6   $n_1.holder_1 \leftarrow n_2$
7   $n_2.holder_2 \leftarrow n_1$
```

Figure 6: Updating Tokens for Two ROOT's from Disjoint Scatternets

for this situation is shown in Figure 6.

3.2.1 Forming Cycles

As mentioned previously, if some cycles are allowed in the scatternet, connectivity can increase and contention can decrease. Since TSF–T has two tokens in any scatternet, cyclic links can be formed between the token holders. Assuming a token holder has passed the eligibility tests described in Section 3.2, we must consider when token holders should form a cycle and how it will affect the token passing algorithm.

First, the number of cyclic links per device should be relatively small. Consider the case when a scatternet, S_1, is relatively large and no devices have attempted to join in a long time (joining could be via FREE devices or merging scatternets). If no limit is placed on the number of cyclic links a device can have, it is expected that after some time all the devices will maximize the number of connections they can handle. This is bad for two reasons. First, battery power is wasted since each device has more connections than it probably needs (i.e. it only *needs* one route to any given device). Secondly, if new devices do attempt to join, S_1 will not be able to merge because no devices can be added to a piconet or accept any new slaves. Therefore, TSF–T limits the number of cyclic links a device can have to two. This is the value that was most efficient in the simulations.

Next, there should be a minimum hop count before devices are allowed to form a cyclic link. Let d be the hop count between two devices. Obviously, if two devices have $d = 1$, they already have a connection and another would be redundant and useless in most circumstances. If $d = 2$, the devices are relatively close and may in fact be slaves in the same piconet. Allowing devices with $d = 2$ to join does not gain much because two devices should not both be in the same two piconets. For the situation where $d = 3$, there are still scenarios where the extra link does not justify the cost. Consider the situation where there are two piconets in existence. Let a piconet with device 1 as master and devices 2 and 3 as slave be denoted $P1_{2,3}$. Assume the piconets are $P1_{2,3}$ and $P4_2$. If cyclic links are allowed with $d = 3$, devices 3 and 4 could form $P3_4$. The only gain is the distance between 3 and 4 decreases from 3 to 1. The distance to every other device in the scatternet remains unchanged. This situation is shown in Figure 7. In this figure, the master points to its slaves and the red arrow is the potential cyclic link being considered. This gain is not worth the price (i.e. adding a connection to each device, increasing the number of cyclic links for these devices closer to the maximum and forming a new piconet). Therefore, in TSF–T, cyclic links can only be formed when $d \geq 4$. Increasing this constraint presents a trade–off. If d_{min} is high, then adding a cyclic link will reduce the hop counts more significantly, but there is a lower probability such a link will ever be formed.

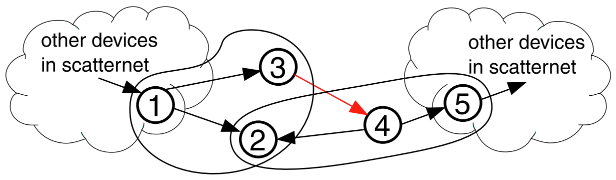

Figure 7: Forming a Cyclic Link between Devices Three Hops Apart

The final issue considered in this section is how the cyclic links affect the token passing algorithms in the scatternet. Recall that Raymond's Tree Algorithm is designed to allow mutual exclusion in a tree. TSF–T works by using two instances Raymond's Tree Algorithm operating concurrently. Therefore, the token passing does not behave correctly in a graph, which is produced by adding cyclic links. To maintain correct token passing, when a cyclic link is formed, no update is made to either of the device's *holder* pointers. This means the token passing algorithm has no knowledge of this link that is formed and hence will never use it. The routing protocol must have some knowledge of the token passing so any time a token or request message is sent, it will be sent on the direct link between the two devices rather than routed some other way. If shortest path routing is used with respect to hop count, this is not an issue. However, if some other metric is used, the token and request messages could be send along a different path. This notification could be accomplished by some means similar to the *router alert* option in IP.

4 Simulations

This section presents the results of simulations done to compare the performance of TSF and TSF–T. The results show that TSF–T consistently generates fewer scatternets than TSF. The number of piconets is also reduced. Furthermore, reduced connection times are obtained when using TSF–T. All this comes at the cost of higher network diameter.

4.1 Simulation Setup

We designed software to perform the simulations using aspects of the Bluetooth protocol. The *ns*-2 simulator [7] with Bluetooth extensions [8] was not used because we were unable to obtain a copy of the original TSF protocols from the authors [9]. Our simulator modeled the frequency hopping, INQUIRY and PAGE states. We did not include collisions and backoff. In comparing the timing of our simulator with the results presented in [3], our numbers were comparable but not close enough to be measured against *ns*-2. Therefore, all the timing comparisons are shown on a relative, not absolute scale. We feel the inaccuracy of timing in our simulator should affect TSF and TSF–T equally.

Simulations were performed using 10, 20, 30, 40, 50 and 60 Bluetooth devices. For each number of nodes, 10 different settings were tested with 10 runs performed for each setting. All nodes arrive at the same time. Simulations are run until a steady state is reached for TSF, and for a short period of time after that for TSF–T, to give time for extra links to form. The upper bound for the running time of the simulations was 200s, which should be sufficient for the amount of time a device is willing to wait to form a scatternet connection.

Parameter	**Value**
$E[T_{inq}]$	5.12 s
D	10.0 s
$\Pr[token_{request}]$	0.25
$age_{threshold}$	1000 s
d_{min}	4
$cyclic\_links_{max}$	2

Table 2: Parameter Values in Simulations

Unless stated otherwise, the simulations were done with a density of 0.14 nodes per 100m^2 where not all nodes are in range of every other node, but every node is in range with at least one other node. Some simulations were performed in a less dense setting (0.10 nodes per 100m^2) to show in those cases TSF–T improvements over TSF are even greater. It should be noted that the nodes in the simulation are considered to be "in range" when they are separated by no more than 80m.

Table 2 summarizes the heuristic values used for the simulation (unless stated otherwise). $E[T_{inq}]$ and D are parameters from the TSF algorithm (we tuned them through simulations because [3] does not mention the values that should be used). $\Pr[token_{request}]$ is the probability a token is requested. The $age_{threshold}$ parameter is how long the device has been in the current scatternet. It is intentionally set to a value that has no effect since we are more concerned with initial scatternet formation and no value was provided in the original TSF specification. The d_{min} and $cyclic\_links_{max}$ parameters are described in Section 3.2.1.

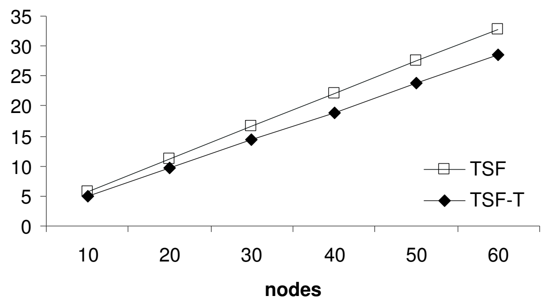

Figure 8: Number of Piconets Formed

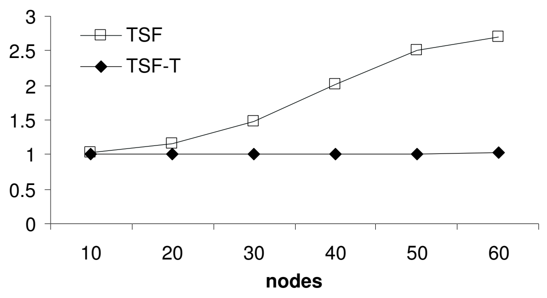

Figure 9: Number of Scatternets Formed in a Dense Setting

4.2 Number of Piconets

As stated earlier, it is convenient to have less piconets to reduce the number of bridge devices and the number of collisions due to shared channels. Figure 8 shows that TSF–T forms fewer piconets in every case.

4.3 Connectivity

In practice, connectivity may be the most important metric. Many real–life applications may turn out to not be usable if multiple scatternets are formed instead of one (we want to be able to send messages to any node, not to just a limited set of nodes).

We present various results that show that TSF–T provides improved connectivity compared to TSF. This is the expected result because TSF, unlike TSF–T, limits the bridge connections to master/slave types.

The number of scatternets formed on each run is an indicator of connectivity. If only one scatternet is formed, any two nodes can communicate with each other. If more than one scatternet is formed, many pairs of nodes are prevented from communicating. Figure 9 shows the significant improvement in this metric when using TSF–T. The use of tokens enables more nodes to connect as bridges, thus, improving connectivity and allowing the formation of only one scatternet most of the time.

TSF–T's improvement over TSF with respect to the number of scatternets formed is even greater in cases with less node density (shown in Figure 10 with

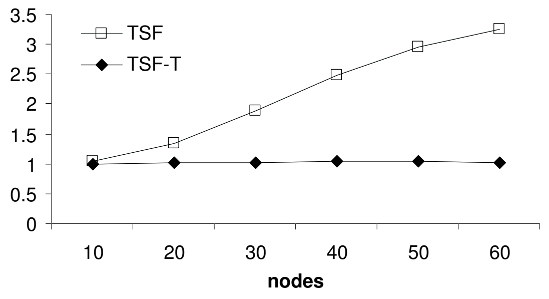

Figure 10: Number of Scatternets Formed in a Sparse Setting

0.10 nodes per 100m^2).

Another interesting value is the percentage of times only one scatternet is formed using each algorithm. Figure 11 shows that while TSF–T is able to form one scatternet almost all the time, TSF performance decreases if more nodes are present.

4.4 Scatternet Formation Time

The use of tokens does not only help in improving connectivity, but it also helps in reducing the scatternet formation time. Figure 12 shows the improvement in scatternet formation time obtained with using TSF–T compared to TSF. By analyzing the graph we can observe that in most of the cases we can obtain more than 50% reduction of the scatternet formation time by using TSF–T.

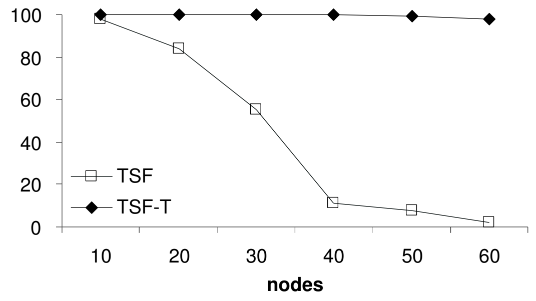

Figure 11: Percentage of Times One Scatternet is Formed

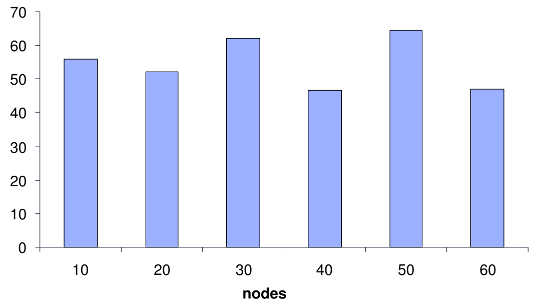

Figure 12: Improvement in Scatternet Formation Time

4.5 Degree

TSF–T was designed not only to improve connectivity, but also to keep into consideration certain characteristics such as mobility, processing capabilities or power consumption when forming the scatternet. Table 3 shows the average degree for each kind of device (averaged over 600 runs with varying number of nodes). The results show that TSF–T is able to form scatternets where the most powerful devices (score = 0) handle more work than the least powerful and more mobile devices (score = 2).

This property makes TSF–T more fault tolerant than TSF since the least powerful devices (or the more mobile ones) handle less work than the more powerful ones.

Score	**Degree**	
	TSF	TSF–T
0	1.88	2.11
1	1.86	2.01
2	1.92	1.93

Table 3: Average Degree for Device Types

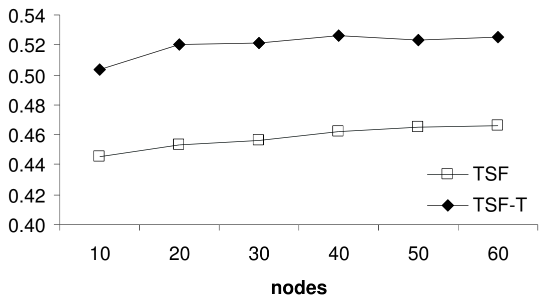

Figure 13: Percentage of Devices which are not Masters

4.6 Slave Percentage

It is important to increase the number of devices that are just slaves or that form slave/slave bridges. Figure 13 presents a comparison of the percentage of devices that are not masters in any piconet. Note that higher percentage of slaves is better. As a simple example, consider the case where eight devices are going to form a scatternet. The ideal scenario is when they form one piconet with a master and seven slaves. This gives $\frac{7}{8}$ slave percentage.

4.7 Network Diameter

To compare the network diameter of the scatternets formed using each algorithm, we ran tests in a more dense setting than before (0.31 nodes per 100m^2) and we only considered those cases when only one scatternet was formed. The runs where more than one scatternet was formed were discarded since in those cases the network diameter is smaller because the scatternets themselves are smaller. Figure 14 presents the results of the experiments. It can be observed that

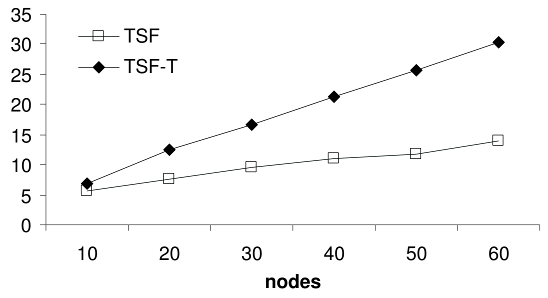

Figure 14: Network Diameter of Scatternets

TSF–T produces scatternets with a higher network diameter than those formed using TSF. This may be due to the fact TSF–T's tree is not as balanced.

5 Future Work

The simulation results provided have shown that TSF–T performs better than TSF. Its characteristics and few drawbacks make it appropriate to use in Bluetooth scatternet formation. Issues that need deeper analysis include:

- *Fault-tolerance*: Studies need to be done regarding fault-tolerance in TSF–T. What happens if a token-holder node stops working? What are the implications and possible solutions if a master or bridge node stops working?
- *Routing*: TSF–T depends on an underlying Bluetooth routing mechanism. It is worthwhile to study different routing mechanisms and their implications when combined with TSF–T.
- *Real-life testing*: Simulations have shown that TSF–T is a viable alternative for scatternet formation in Bluetooth. It would be interesting to see it in action in different Bluetooth scenarios.
- *Mobility, Processing Capabilities and Power Consumption*: It was mentioned earlier that TSF–T takes into consideration the *score* of a device when establishing connections between nodes. The most powerful, less mobile devices are preferred masters, since they are less likely to fail. Another approach to the usage of the score of a device is to take it into consideration when requesting the token. It seems that making the devices with lower score request the token more frequently than the devices with a higher score may be a good idea. In the simulation, the probability a device requests a token is fixed. This probability should be tested as a function of variables such as degree, time of last successful connection and *score*.
- *Score*: TSF–T assumes that the lower the score, the less mobile and more powerful a device is. But what are the exact criteria for assigning a particular score to a device? What should be the range of values for the score? Security issues should also be considered; for example, malicious devices may assign themselves the highest score possible even if they are able to manage a greater workload. How can this be prevented (or detected)?
- *Multiple tokens*: TSF–T considers the use of two simultaneous tokens to improve connectivity. Each token is handled in a separate overlay of Raymond's Tree Algorithm. It is possible that greater performance improvement can be obtained by adding more tokens. But overlaying more Raymond trees is not a feasible solution. Other algorithms should be considered and analyzed. [10] provides an multiple node extension to Raymond. Simulations should be done as to determine if allowing more than two nodes to circulate at the same time leads to better performance results.
- *Simulation with random arrival times* Both this study and the original TSF paper do not simulate devices entering at random times. It would be interesting to compare the protocols when devices do not all arrive at the same time, as well as when devices leave the scatternets.

6 Conclusion

This paper presented TSF–T, a new Bluetooth scatternet formation algorithm that introduces the concept of using tokens to improve connectivity when forming scatternets. TSF–T efficiently forms scatternets by using tokens (circulating according to Raymond's Algorithm) to determine which nodes are able to connect with other nodes at a given time. Unlike other scatternet formation algorithms, TSF–T is able to efficiently form scatternets even if not all nodes are in range of each other.

Simulations show that TSF–T performs significantly better than TSF with respect to connectivity and connection time. TSF–T takes into consideration the mobility, processing, and power consumption characteristics when forming scatternets, which makes TSF–T more fault-tolerant than TSF. The scatternets formed with TSF–T have less master's than with TSF. This is shown with an analysis of the slave percent metric introduced in the paper. TSF–T is a promising algorithm for Bluetooth scatternet formation.

Acknowledgements

We would like to thank Dr. Jennifer Hou for helping us formulate the idea for this project and suggesting the token passing approach.

References

[1] The Bluetooth Special Interest Group. `http://www.bluetooth.com`.

[2] J. Haartsen, "Bluetooth—The universal radio interface for ad hoc, wireless connectivity," *Ericsson Review*, no. 3, 1998.

[3] G. Tan, A. Miu, J. Guttag, and H. Balakrishnan, "Forming Scatternets from Bluetooth Personal Area Networks," Technical Report MIT–LCS–TR–826, MIT Laboratory for Computer Science, October 2001.

[4] T. Salonidis, P. Bhagwat, L. Tassiulus, and R. LaMaire, "Distributed Topology Construction of Bluetooth Personal Area Networks," in *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, 2001.

[5] C. Law, A. K. Mehta, and K.-Y. Siu, "Performance of a New Bluetooth Scatternet Formation Protocol," in *Proceedings of the ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc) 2001*, October 2001.

[6] K. Raymond, "A Tree–Based Algorithm for Distributed Mutual Exclusion," *ACM Transactions of Computer Systems*, pp. 61–77, February 1989.

[7] The *ns*-2 Network Simulator. `http://www.isi.edu/nsnam/vint`.

[8] Bluetooth Extension for *ns*-2. `http://oss.software.ibm.com/bluehoc`.

[9] Blueware Project. `http://www.nms.lcs.mit.edu/projects/blueware`.

[10] J. Walter, G. Cao, and M. Mohanty, "A K–Mutual Exclusion Algorithm for Ad Hoc Wireless Networks," in *Proceedings of the First Annual Workshop on Principles of Mobile Computing (POMC 2001)*, August 2001.