A Locally Coordinated Scatternet Scheduling Algorithm

Godfrey Tan and John Guttag
MIT Laboratory for Computer Science
Cambridge, MA 02139
{godfreyt, guttag}@lcs.mit.edu

Abstract

There is growing interest in wireless personal area networks built from portable devices equipped with short-range radio interfaces such as Bluetooth. These small networks (called piconets) can be internetworked to form larger scatternets *by means of bridge nodes that participate in more than one piconet on a time division basis. How well this works depends to a large part on the mechanism used to schedule communication across piconets.*

In this paper, we present a novel online scatternet scheduling algorithm, LCS, that effectively coordinates one-hop neighbors to converge to an efficient scatternet-wide communication schedule. Unlike previous work, LCS is robust and responsive to network conditions, dynamically adjusting the schedule based on varying workload conditions. We demonstrate that LCS has good performance on throughput, end-to-end packet latency and energy usage under various traffic loads.

1. Introduction

Bluetooth [2] is emerging as one the RF technologies of choice for short-range communication between low-power devices. The standard specifies mechanisms for establishing connection with nearby devices in an *ad hoc* manner [2, 5, 3]. Bluetooth achieves robustness against interference from nearby devices by employing a frequency hopping technique and time division multiplexing (TDM). This facilitates high densities of communicating devices, making it possible for dozens of small networks (called piconets) to co-exist and independently communicate in close proximity without significant performance degradation. It is also possible to internetwork multiple piconets to create a larger *scatternet*. In this paper, we identify the challenges in effectively scheduling Bluetooth communication links in scatternets and present an efficient scheduling algorithm for both intra-piconet and inter-piconet communication. While some aspects of the work is Bluetooth-specific, most of our results are applicable to any TDM-based link technology.

Scheduling communication links in Bluetooth scatternets presents an interesting challenge. Two properties of scatternets make this task a difficult one. First, devices in a Bluetooth piconet communicate using a polling scheme in which one side of a link plays the role of *master* and other the role of a *slave*. A slave is allowed to transmit only if it has been polled by the master in the preceding time slot. Second, piconets are interconnected via bridge nodes that participate in multiple piconets on a time division basis.

There are two types of bridge nodes: slave and master. In Figure 1, master bridge D participates as slave in A's piconet and as master in its own piconet. Slave bridge B communicates as slave in both A and C's piconets. Coordination among nodes is required to successfully and efficiently transfer packets within the scatternet.

This paper presents a locally coordinated scheduling algorithm (called *LCS*) for Bluetooth scatternets. LCS

- Coordinates one-hop neighbors to efficiently converge to an efficient scatternet-wide link schedule,
- Dynamically adjusts for each link both the duration of and the interval between communication events based upon changing traffic patterns,
- Tolerates intermittent connectivity, and
- Allows for schedules that are optimized for system-wide throughput, end-to-end communication latency, or energy consumption.

LCS achieves an efficient scatternet wide schedule by

1. Computing the duration of the next meeting based on queue size and past history of transmissions so that the duration is just large enough to exchange all the backlogged data,
2. Computing the start time of the next meeting based on whether the data rate observed is increasing, decreasing or stable so that it responds to varying traffic conditions quickly without wasting resources,
3. Grouping together meetings with the same traffic characteristics to reduce wasted bandwidth of nodes and end-to-end latency,

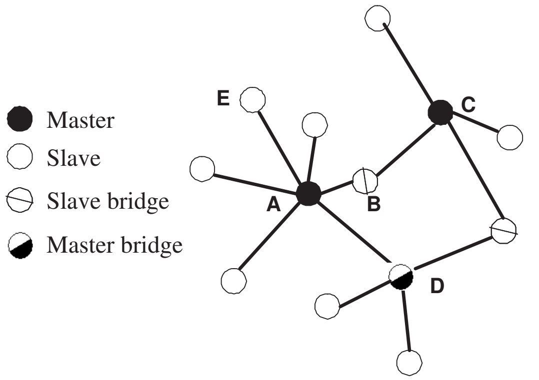


Figure 1. A Bluetooth scatternet

4. Aligning meetings at various parts of the scatternet in a hierarchical fashion so that the number of parallel communication is high, increasing system-wide throughput significantly, and

5. Reducing the amount of time a node spends transmitting packets while the receiver is not ready, thus conserving energy.

The structure of scatternets can greatly influence the design of scatternet-wide scheduling algorithms. Several research groups [9, 14, 13] have discussed the advantages of loop-free topologies, and we designed LCS to work best for such topologies. LCS does not depend on how piconets are bridged. The scatternet formation scheme presented in [13] uses master bridge nodes while the scheme described in [9] uses slave bridge nodes. LCS provides efficient intra and inter-piconet communication for scatternets using both master nodes as bridges (as proposed in [13]) and slave nodes as bridges (as proposed in [9]).

In Section 2 we present the design goals that drove our design. In Section 3, we discuss the prior work in related research areas. We present LCS in detail in Section 4, and evaluate its performance through simulation in Section 5.

2 Design Goals

A scatternet scheduling algorithm must work well for two general classes of application: delay sensitive and throughput sensitive. Delay sensitive applications range from simple applications controlling Bluetooth enabled mice to voice and streaming applications. These applications send packets of fixed-size application data units (ADUs) periodically and have worst case per-packet delay requirements. The delay sensitive applications are usually not bandwidth-intensive since ADUs tend to be small in size and are not generated rapidly by the applications. Of course, there could also be applications such as realtime video that are both latency sensitive and bandwidth intensive.

Throughput sensitive applications include various file transfer applications such as FTP and email applications. Throughput sensitive applications are concerned with transmitting large ADUs as quickly as possible. They tend to use reliable transport protocols and are bursty in nature. These applications are more concerned with the average throughput available than the per-packet delay perceived.

Since many network nodes will be battery powered, energy efficiency is also an important consideration. The energy required to actually transmit and receive a bit of application data is independent of scheduling algorithm. However, in many cases the energy consumed by communication overhead is significant or even dominant. In particular, "listening" for packets that don't arrive consumes a great deal of energy. We therefore evaluate the energy efficiency of a schedule as the ratio of the time the node has been active to the number of application bytes a node successfully transmits and receives.

As we demonstrate in Section 5, there is often a tradeoff between communication latency and energy consumption. When two nodes fail to recognize that they could communicate, latency is increased. On the other hand, energy is wasted when

1. An end node is active on a link while the other is busy communicating on another link, or

2. Two end nodes establish a communication link with each other but don't exchange data during the entire period the link is active, or

3. Bridge nodes switch between piconets with poorly synchronized master clocks incurring the so-called *bridging overhead*.

The key to managing this tradeoff is the way in which the scheduler coordinates end nodes communicating on multiple links. The coordination can be done statically or dynamically.

A static scheme can be advantageous when traffic patterns are known in advance and vary little over time. However, we don't expect this situation to prevail in most cases.

LCS is a dynamic and distributed scheduling algorithm. Nodes coordinates communication among neighboring nodes while allocating bandwidth based on current local traffic conditions. LCS optimizes the overall efficiency of the scatternet, in terms of throughput, latency and energy, by minimizing wasted and missed communication opportunities and piconet switches. We discuss LCS in detail in Section 4 after presenting some background material and an overview of related work in the next section.

3. Background and Related Work

As discussed above, a Bluetooth scatternet connects a set of piconets. Each piconet consists of one master and up to seven slaves. Bluetooth is a time-slotted system where each transmission time slot takes 625 microseconds. The master and slaves use alternate transmission slots. In even slots, the master sends a frame to a single slave, which responds in the following odd slot. The Bluetooth standard calls for links to have a maximum capacity of 1Mbps. However, the effective data rate varies depending on the type of Bluetooth Baseband packets used. The Baseband data packets can be 1, 3 or 5 slots long. Note that an ADU may be segmented into multiple smaller Bluetooth Baseband packets. Special POLL and NULL packets are used to poll (by master) and respond (by slave) when there is no data packet available. Thus a POLL-NULL sequence indicates that neither the master nor slave has data for the other.

Although protocols for intra-piconet scheduling have been studied extensively [15, 8, 6, 7], there has been little research work done for scatternet-wide link scheduling. The authors in [1] present a scatternet scheduling scheme that provides fair link bandwidth allocation. Each node makes an independent decision to communicate with other nodes depending on its fair share of bandwidth. Due to the lack of coordination, their scheme does not scale as the number of forwarding hops along the communication path increases beyond a couple of hops, and is not responsive to varying traffic conditions. Racz *et al.* present a pseudo-random scheduling scheme (called PCSS) for Bluetooth scatternets [11]. In PCSS, every node randomly chooses a communication checkpoint that is computed based on the master's clock and the slave's device address. When both end nodes show up at a checkpoint simultaneously, they communicate until one of the nodes leaves to attend to another checkpoint. In order to adapt to various traffic conditions, PCSS measures the link utilization in a coarse grain manner and adjusts the checking period accordingly.

The advantage of PCSS is that it achieves coordination among nodes with very little overhead. However, since PCSS is based on a randomized scheme, as the density of nodes grows, there will be scheduling conflicts among various checkpoints resulting in missed communication events in which one end node is actively waiting for another node that is busy communication with some other node. LCS coordinates nodes in a manner that eliminate all scheduling conflicts. In response to changing traffic on a link, PCSS increases or decreases the interval between two successive communication events on that link by a fixed multiple. It does not change the duration of communication events nor does it coordinate with other links. This makes it less responsive to bursty traffic than LCS, which adjusts both the intervals between communication events and the duration of those events.

4. LCS: Locally Coordinated Scheduling

LCS is based on the concept of scheduled meetings called appointments. End nodes on a particular link meet according to their appointment and exchange data during the meeting. Before the meeting is terminated, the two end nodes[1] negotiate the $start$ time and the minimum $duration$ of their next meeting:

- A parent sends a list of possible future meeting start and finish times to a child, and
- The child replies with desired start and finish times of one future meeting, which fall within one of the meeting periods suggested by the parent.

The choice of meeting time and duration by both parent and child dictates the effectiveness of the link schedule. LCS monitors the traffic characteristics associated with each link and attempts to arrive at an efficient scatternet-wide schedule based on them. The rest of this section describes LCS in detail and explains how it converges to an efficient scatternet-wide schedule.

4.1. Protocol

LCS is distributed with each node running the same procedure as shown in Figure 2, RUN. Each node keeps an ordered (by start time) list of outstanding appointments, $mlist$, and each appointment is denoted by a pair (s_l, f_l), where l is the communication link with which the appointment is associated and s_l and f_l represent the start and finish times of the future meeting. Observe that there is always exactly one outstanding appointment associated with each link since two nodes negotiate for it at the end of the previous meeting.

This first meeting between the two nodes occurs when they establish a new communication link. LCS requires the nodes to remain active on that link for several time slots. During this period, the nodes follow connection establishment procedures required by higher Bluetooth layers such as L2CAP. The nodes remain communicating until either i) one of them needs to attend another meeting, ii) there is no more data to exchange, or iii) there is no more buffer space left at the receiving node to store packets. In each case, one node declares the meeting over and then both nodes negotiate the start time and the duration of the next meeting. Each node can then attend another meeting or sleep to save power.

An end node indicates its desire to end the meeting by sending a single-slot LCS packet (see SENDPKT). There are

[1] End nodes can be assigned parent and child roles starting from a designated root node in any loop-free networks for which LCS is designed to work best.

```
PROCEDURE RUN() {
do forever
    (s_l, f_l) ← pop(mlist)
    sleep for max((s_l − clock), 0) slots
    communicate on link l
    if(wishes to end the meeting)
        choose type based on the reason for termination
        SENDPKT(type)
    if(a LCS packet p is received)
        RECVPKT(p)
}
PROCEDURE SENDPKT(type){
    UPDATELINKSTATES(tx, qsz)
    d^{t+1} ← COMPUTEDURATION(type, qsz)
    s^{t+1} ← COMPUTERECESS(type) + clock
    s^{t+1} ← PARENTCONVERGE(s^{t+1})
    for each i = 1 to N_meet
        (v_s^i, v_f^i) ← vacant period after s^{t+1}
    send a type pkt containing qsz, d^{t+1} and (v_s^i, v_f^i)
}
PROCEDURE RECVPKT(p){
if(p is from child)
    SENDPKT(p.type)
else if(p.type = LCS_REP)
    modify mlist with (p.s^{t+1}, p.f^{t+1})
else     /* LCS_REQ, LCS_QFULL, or LCS_NONE */
    d^{t+1} ← COMPUTEDURATION(type, qsz + p.qsz)
    (s^{t+1}, f^{t+1}) ← CHILDCONVERGE(d^{t+1}, p)
    send a LCS_REP pkt containing s^{t+1} and d^{t+1}
    modify mlist with (s^{t+1}, f^{t+1})
}
```

Figure 2. Pseudo-code of the LCS protocol

```
PROCEDURE UPDATELINKSTATES(tx, qsz) {
    avg_tx ← tx × γ + (1 - γ) × avg_tx
    avg_qsz ← qsz × γ + (1 - γ) × avg_qsz
    d^t ← max(clock, f^t) − s^t
    r^t ← s^t − f^{t−1}
    u^t ← tx ÷ d^t
    update avg_r and avg_u based on past N_r
    values of r^t and u^t
}
PROCEDURE COMPUTEDURATION(type, qsz) {
    d^{t+1} ← max(avg_tx, avg_qsz)
    if(type = LCS_REQ and qsz > 0)
        d^{t+1} ← d^{t+1} + qsz × K_qsz
    d^{t+1} ← max(d^{t+1}, qsz)
    d^{t+1} ← min(D_max, max(d^{t+1}, D_min))
    return d^{t+1}
}
PROCEDURE COMPUTERECESS() {
    n_r ← n_r + 1
    if(idle for long time and u^t > 0)
        tar_r ← R_init
    else if(n_r ≥ N_r)
        tar_r ← average r value
        if(no data transmitted or data rate is decreasing)
            tar_r ← tar_r × α
        else if(data rate is increasing)
            tar_r ← tar_r ÷ α
        if(tar_r is unmodified and data rate is stable)
            tar_r ← (tar_u ÷ avg_u) × tar_r
    r^{t+1} ← min(R_max, tar_r)
    return r^{t+1}
}
```

Figure 3. Pseudo-code to compute r and d

three different kinds of termination packet, each indicating a different reason for terminating the meeting. When a node needs to attend another meeting, it informs the other node by sending a LCS_REQ packet. When the agreed duration is longer than necessary to transmit the back-logged data, the node sends a LCS_NONE packet. As described before, the POLL-NULL sequence indicates the absence of data to exchange. Finally, when the receiver's buffer is overflowing, the receiver sends a LCS_QFULL packet to the sender. Note that other than the case when there is no data to exchange, two nodes communicate for at least $(f_l - s_l)$ time slots.

Sending a LCS packet of any kind marks the beginning of the negotiation process for the future meeting. Note that either parent or child can initiate the negotiation process which includes two steps: i) the parent suggests possible future appointments and ii) the child chooses the most suitable one among them. When the parent node realizes that the current meeting is terminating, it calls appropriate routines to update per-link states and to compute the suitable meeting times and durations, and then, sends a LCS packet containing a list of possible meeting start and finish times. The child replies with a LCS_REP packet containing the most suitable meeting time and duration. Each node then modifies $mlist$ accordingly and sleeps until it is time to attend another meeting. The rest of this section explains the protocol in detail.

4.2. Per-link States

The critical part of LCS is how it computes a suitable future meeting time and duration. Computing the start time and duration of a future meeting on a link depends not only on the current and expected future traffic loads associated with that link but also on the current and expected future loads of adjacent links. Scheduling the meeting at the earliest time may result in a waste of resources when there is no data to exchange. Similarly, scheduling a long meeting could result in a waste of bandwidth if there is not enough data to be exchanged during that period. On the other hand, scheduling short meetings increases the bridging overhead.

To deal with dynamic traffic conditions, each node monitors the traffic characteristics of every adjacent link. Each node maintains for each link l [2] the averages of: i) the number of slots used to transmit or receive data packets during a meeting, tx, ii) the combined output link queue size, qsz, iii) the recess interval, r, i.e., the time between the start time of the current meeting and the finish time of the preceding meeting associated with the same link, and iv) the link $utilization$, $u = \frac{tx}{d}$, where d is the duration of the current meeting. $u = 1$ means that the nodes are utilizing all of its allocated link bandwidth whereas $u = 0$ means that there is no data to exchange.

When the negotiation process for a future meeting begins, the link states are updated as described in UPDATELINKSTATES in Figure 3. avg_tx and avg_qsz are the exponentially weighted moving averages associated with the link and $0 < \gamma < 1$ is the weight assigned to the most recent value. avg_r and avg_u are the unweighted averages of the last N_r values of r and u respectively. We will explain how these averages are used in Section 4.4

All variables associated with the link have time-slot units. Based on the link states, LCS computes the duration, d^{t+1}, and the recess interval, r^{t+1}, of the next meeting. The start time of the next meeting, s^{t+1}, is simply $r^{t+1} + clock$, where $clock$ is the current clock value. Together d and r dictate the bandwidth available to each link, per-packet latency and energy efficiency. Imagine that two nodes are meeting periodically to exchange a fixed amount of data. The bandwidth allocated to the node is $\frac{d}{d+r}$ and can be increased or decreased by adjusting either d or r. Also note that the higher the value of r, the larger the end-to-end packet latency and the more energy is conserved. In the next few subsections, we explain how LCS computes d and r based on the link states.

4.3. Computing Meeting Duration

Procedure COMPUTEDURATION in Figure 3 calculates the desired duration of the next meeting, d^{t+1}. The goal is to allocate the minimal number of transmission slots for the two nodes to exchange data when they meet the next time. Simply put, we want to choose suitable d^{t+1} so that the link utilization is high. LCS computes the expected amount of data to be transmitted in next meeting as the function of the weighted average of the past transmissions, avg_tx, and of the combined link queue size, avg_qsz. This allows LCS to respond quickly to sudden changes in traffic conditions. We use exponentially weighted moving averages since the most recent values give the best indication of the immediate future.

When the meeting is terminated as a result of one of the nodes needing to attend a different meeting and there is still data in the current link's queue, LCS increases the next meeting duration (thus, channel bandwidth), d^{t+1}, by $K_{qsz} \times qsz$ slots. This adjustment improves efficiency by reducing the number of piconet switches and negotiation overhead. We show in Section 5 that setting $K_{qsz} = 0.5$ works well for various traffic patterns.

Having a long communication interval at each hop will increase buffering delay, and thus, per-packet latency, and also starve other nodes. To avoid this, d^{t+1} is upper-bounded by D_{max}. Thus, if multiple links are busy, each of them will be able to transmit for D_{max} slots. Furthermore, to ensure that the two nodes can communicate again in the future, d^{t+1} is lower-bounded by D_{min}.

4.4. Computing Recess Interval

Computing the recess interval before the next meeting begins, r^{t+1}, is more complicated than computing d^{t+1}. The main challenge is to find a method that works well for both bandwidth sensitive applications and latency sensitive applications.

r^{t+1} is adjusted according to the data rate and the nature of the traffic flows going through the link. If bandwidth intensive flows are going through the link, we aggressively allocate more bandwidth to the link. However, if the data rate is decreasing as a result of some flows stopped transmitting, we improve energy efficiency by reducing bandwidth allocation. Furthermore, if the flows are latency or energy sensitive, we provide flexibility to tradeoff between latency and energy.

LCS detects a change in traffic conditions by comparing the link characteristics of the two most recent intervals. Based on tx, r, and u, LCS determines whether the data rate is i) increasing, ii) decreasing or iii) stable.[3] If the data rate is increasing, LCS increases the bandwidth allocated to the link by reducing r. Conversely, if the data rate is decreasing, r is increased to avoid meetings where no useful data is exchanged.

COMPUTERECESS computes the next recess interval according to current traffic conditions and stores it in tar_r (for target recess). The first *if* statement increases responsiveness in a way described shortly. Unlike COMPUTEDURATION, COMPUTERECESS adjusts tar_r once every N_r meetings and r^{t+1} is simply assigned to tar_r. This allows nodes to coordinate effectively as they settle on a stable meeting schedule for a certain period before the recess

[2] We are omitting the subscript l for each of these variables for clarity.

[3] Exactly how LCS determines the nature of the traffic is subtle, and omitted due to space constraints.

interval is adjusted again. Note that tar_r is reset to the average r value measured every N_r meetings. This feedback mechanism enables LCS to adjust tar_r dynamically based on the current traffic conditions. As we shall see in Section 4.5, the measured r value between two meetings could be different from tar_r.

Increasing tar_r will result in reduced bandwidth on that link as nodes meet less frequently whereas decreasing tar_r will result in increased bandwidth. When there is no application data exchanged during the last window, LCS increases tar_r by a factor of $\alpha > 1$. Thus, if a link has been idle for a long time (i.e. no data exchanged), tar_r will become big. When a node receives data after a long idling period, LCS increases the responsiveness quickly by setting tar_r to R_{init} (the first *if* statement). When LCS detects the decline in data rate, it increases tar_r by a factor of α. When LCS detects the increase in data rate, it reduces tar_r by a factor of α.

LCS constantly monitors the traffic condition to determine whether the data rate has become stable. The data rate observed on a link can become stable for two different reasons: i) the link is saturated, i.e. end nodes are already transmitting data at a maximum capacity allocated, or ii) the applications are non-bandwidth intensive and are transmitting data at a steady low rate. If it is the former, we do not adjust the recess interval since it will have no useful effect. If it is the latter, we can decrease the per-packet latency by activating the link more frequently. Doing so decreases the scheduling delay at each hop and thus, reduces end-to-end packet latency. Unfortunately, this will also result in higher energy usage since more frequently, nodes will not have data to exchange.

Hence, when the data rate is stable, LCS adjusts tar_r based on the predefined parameter tar_u. tar_u is a target utilization parameter that users can initialize, and LCS adjusts r^{t+1} periodically so that the average utilization, avg_u, will be close to tar_u. For low data rate applications, avg_u is a direct measure of how often the nodes meet compared to the actual rate of the data. If the end nodes meet as frequently as the data packets are transmitted, avg_u, will be close to 1. $avg\_u = 0.5$ means that the nodes are meeting roughly twice as fast as the data transmission rate since at half of the meetings, there will be no data to exchange.

We now explain how tar_u is used to control the end-to-end latency and energy usage. Observe that the difference between the time that a packet is enqueued and the time that the two nodes meet could be as high as tar_r. We call this difference the *scheduling delay*. Since this delay adds up over each hop, an effective way to reduce end-to-end latency is to reduce the scheduling delay at each hop. This can be achieved by activating links more frequently, i.e. reducing tar_r. However, as explained before, reducing tar_r decreases average utilization, and thus, increases

PROCEDURE PARENTCONVERGE(s^{t+1}) {
    if(tar_r is unmodified)
        $\forall$ link $j \mid tr_j$ and tr are within δ and $l \neq j$
            $(f_k^{t+1} - s^{t+1}) \leftarrow \min(f_j^{t+1} - s^{t+1})$
            $s^{t+1} \leftarrow f_k^{t+1}$
            return s^{t+1}
}
PROCEDURE CHILDCONVERGE(d^{t+1}, p) {
    $d^{t+1} \leftarrow \max(d^{t+1}, p.d^{t+1})$
    for each $(p.v_s^i, p.v_f^i)$
        $\forall j \mid v_s'^j \geq p.v_s^i$ and $v_f'^j \leq p.v_f^i$
            $e^k \leftarrow \min(e^j)$
            $s^{t+1} \leftarrow v_s'^k$
            $f^{t+1} \leftarrow s^{t+1} + \min(v_f'^k - v_s'^k, d^{t+1})$
            return (s^{t+1}, f^{t+1})
}

Figure 4. Pseudo-code to converge to an efficient schedule.

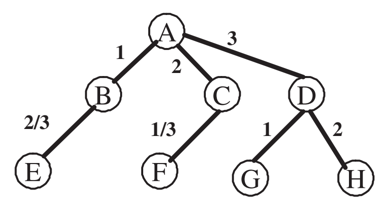


Figure 5. A scatternet containing 8 nodes where A is the root. The numbers on the links denote the communication rounds associated with the links.

the energy usage. LCS allows users to set tar_u so that desired latency and energy requirements can be achieved. COMPUTERECESS modifies tar_r whenever avg_u is not within δ percent of tar_u. We will examine this latency and energy tradeoff through simulation in Section 5. Lastly, tar_r is upper-bounded by R_{max} so that the two nodes will meet every R_{max} time slots even if they don't have any data to exchange. Although doing so could potentially result in wasteful communication events, this is necessary to ensure that nodes can communicate again in a timely fashion should data start arriving.

4.5. Converging to an Efficient Schedule

So far, our computation of d^{t+1} and r^{t+1} of a link l solely depends on the traffic conditions observed on that link. However, since each link is activated on a time division basis, the overall efficiency of the node depends on

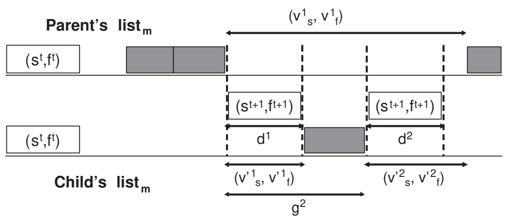


Figure 6. Parent and child's meeting lists

how well various meetings on multiple adjacent links can be coordinated. It is important that the scheduling of a new meeting does not result in cancellation of existing meetings since doing so will result in unsuccessful communication events. LCS never schedules overlapped meetings and thus, eliminates scheduling conflicts. Most importantly, LCS arrives at an efficient scatternet-wide link schedule by coordinating one-hop neighbors.

As explained in Section 4.1, either the parent or the child may send a LCS packet (see SENDPKT) to begin the negotiation process for determining the next meeting time and duration. First, the parent node computes s^{t+1} and d^{t+1} and sends a packet containing the following fields: i) the vacant periods, (v^i_s, v^i_f), where $1 \leq i \leq N_{meet}$, ii) the recommended duration for the future meeting, d^{t+1}, and iii) the current link queue size, qsz. Note that qsz is the combined size of both end nodes' queues. Each pair of (v^i_s, v^i_f) represents the start and finish times of a vacant period or gap in the meeting list $mlist$ of the parent (see Figure 6). The child then picks the most suitable meeting time and duration and informs the parent via a LCS_REP packet. Both nodes update their meeting list and continue executing RUN.

Keep in mind that LCS is designed to be used to schedule loop-free scatternets. It takes advantage of the following facts to converges to an efficient scatternet-wide link schedule:

1. Starting from a root node, scheduling links in a hierarchical fashion yields a scatternet-wide schedule involving a maximal matching of active links, and

2. The efficiency of the scatternet-wide schedule depends on how tightly meetings are scheduled at every level.

Consider the loop-free scatternet in Figure 5. Observe that starting from the root node A, a maximal matching of links can be scheduled simultaneously in each communication round. A matching is $maximal$ if no matching can result by adding an extra link to it. For simplicity, assume that this scatternet is constructed using master bridge nodes and that A is a master node and B, C, and D are master bridge nodes. Starting from A, we can assign a communication round to each link so that no two adjacent links have the same communication round assigned. Observe that the total number of links that can be activated in each communication round is also the maximum possible. The higher the degree of parallelism, the larger the system-wide throughput.

Since each meeting is scheduled independently based on the traffic characteristics associated with the link, there could be gaps between meetings in the node's meeting list, $mlist$. Each gap can potentially become unused bandwidth reducing the efficiency of the link schedule. Observe that the higher the degree of fragmentation in $mlist$, the harder it is for two nodes to negotiate for a mutually convenient meeting. Thus, it is important that the number of gaps between meetings remains small. To achieve this, LCS organizes the meetings with similar traffic characteristics and schedules them consecutively. As shown in the PARENT-CONVERGE routine in Figure 4, LCS modifies s^{t+1} of link l so that the meeting begins right after the closet meeting with similar data rate has ended. Of course, this adjustment only happens if there is no change in traffic conditions, i.e., tar_r is left unmodified by COMPUTERECESS.

The parent node then chooses N_{meet} vacant intervals after s^{t+1}. Each vacant interval (v^i_s, v^i_f) represents the gap between two meetings whose start times are greater than s^{t+1} (see Figure 6). The start time of the last vacant interval is always the maximum of the finish time of the last meeting and s^{t+1}, and its duration is ∞. The parent then sends the list of (v^i_s, v^i_f) along with d^{t+1} and qsz to the child. Observe that the child will always be able to pick a meeting whose start and finish times fall between one of the vacant intervals suggested by the parent.

When the child receives a LCS packet, it figures out the most suitable meeting time and duration as shown in RECVPKT. The child first updates its link states and calculates its own desired meeting duration. Sometimes, the duration suggested by the parent may be different from the one desired by the child and so the child simply picks the maximum of the two as d^{t+1}. For each vacant interval (v^i_s, v^i_f) suggested by the parent, the child looks in its $mlist$ to see there is any vacant interval (v'^j_s, v'^j_f) that falls within v^i_s and v^i_f. If there is more than one vacant interval that meets the requirements, the child picks the one with the highest efficiency, e, defined as follows:

$$e^j \leftarrow \frac{0.5 \times d^j}{(d^j + g^j)} + \frac{0.5 \times d^j}{(v'^j_f - clock),} \tag{1}$$

where $d^j = min(d^{t+1}, v'^j_f - v'^i_s)$, $g^j = (v'^j_s - max(f', v^1_s)) + (v'^j_s - v^1_s)$, and f' is the finish time of the meeting preceding the vacant interval (v'^j_s, v'^j_f). d^j represents the duration of a possible future meeting whereas g^j represents the sum of the gap at parent and that at the child if the future meeting is to take place between v'^j_s and v'^j_f. Therefore, e^j ensures that the child schedules the meeting at the earliest time (first term) that will result in the smallest combined gap (second term) at both parent and child's $mlist$. Figure 6 shows the meeting lists of both child and parent during the negotiation process. In the figure, there are two possible vacant intervals in the child's $mlist$ during which the next meeting can be scheduled. According to the Equation 1, the child will pick the first one. The child then sends a LCS_REP packet containing the desired meeting start and finish times (s^{t+1}, f^{t+1}). Both nodes update their meeting list properly, terminate the meeting, and either go to sleep or attend a meeting associated with a different link.

4.6. Fault Tolerance, Fairness and Implementation

In practice, links may temporarily fail due to crashes or mobility. Therefore, an end node may not show up during the agreed-upon meeting period. A master node detects the absence of a slave node when it does not receive any response after polling it for N_{poll} times consecutively. Similarly, a slave node assumes that the master is absent after not being polled in N_{poll} consecutive even slots. This grace period is necessary to cater for clock drifts especially when two nodes haven't met for a long period.

When end nodes do not meet in the agreed meeting period, they both reschedule the future meeting automatically based on the last agreed meeting start time, s^{last}. Each node chooses an appropriate future meeting beginning at $s^{last} + 2^k \times T_{wait}$, where T_{wait} is the waiting period and k is the number of rescheduling attempts.

If a node detects that the other end node misses the meeting for several times, it will assume that its counterpart has disappeared and disconnect the Bluetooth link. If a node desires to re-connect to the scatternet, it must again go through the discovery procedures required by the scatternet formation scheme as described in [13].

We are currently investigating to integrate a suitable fair queuing algorithm with our scheduling scheme. Several existing algorithms such as Fair Queuing [4] and Deficit Round Robin (DRR) [12] exist to provide fairness among competing flows going through routers. Both schemes require proper classification of flows and a significant amount of memory and thus, may not be suitable for Bluetooth devices with small memory. Fair bandwidth allocation in Bluetooth is further complicated by the facts that forwarding nodes communicate on adjacent links on a time division basis.

LCS can be implemented within the existing Bluetooth specification. The only mechanism required by LCS is how to activate and deactivate communication links as needed. The Bluetooth specification describes two mechanisms to achieve that goal: $Hold$ and *Sniff*. LCS can work with either mechanism, but we omit the details due to space constraint.

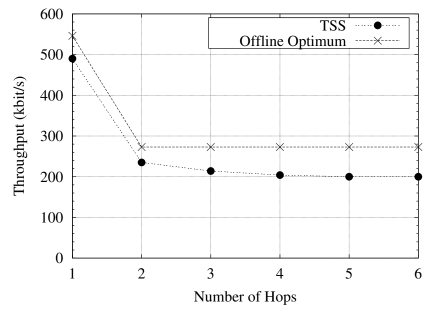


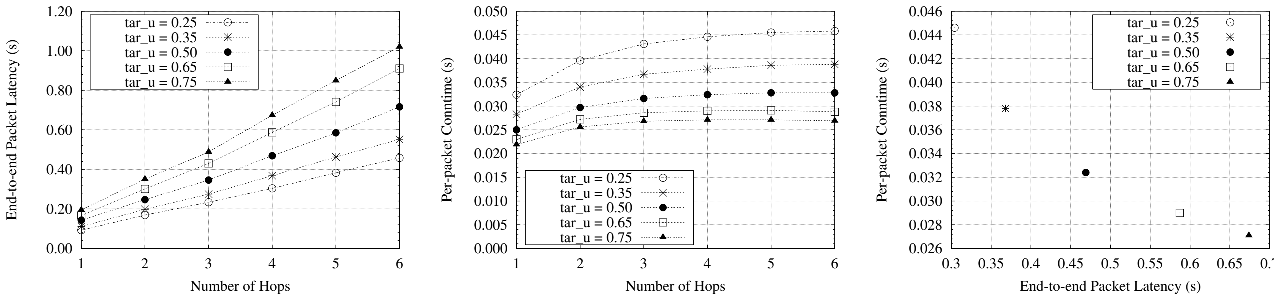
Figure 7. Achieved throughput of a TCP flow.

5. Performance Evaluation

To evaluate the effectiveness of our algorithm, we developed a Bluetooth simulator as an extension to the Network Simulator (ns) [10] and implemented LCS in this simulator. We conducted simulation runs under several different scatternet topologies and various traffic loads and measured average throughput available, end-to-end packet latency and energy usage. The results show that LCS responds quickly to changing traffic conditions and scales well with the increase in the scatternet size and the number of flows. The rest of this section discusses the simulation setup and performance results in detail.

We use the scatternet formation scheme described in [13] to generate loop-free topologies of various sizes. For each scatternet size n, where n is the total number of nodes in the network, we run simulations on 20 different topologies and use the average in evaluating LCS's performance. Applications are started simultaneously several seconds after a single connected scatternet of size n has been formed. We use two different types of applications with varying needs: FTP and CBR. FTP applications use TCP (New Reno) as data transport and attempt to send as much data as possible within the simulation period. CBR applications send fixed-size packets at a low rate of 5kbps. The size of each TCP packet is 512 bytes and that of CBR packet is 335 bytes. Each flow sends data for about $200s$.

We set the LCS parameters as follows: $D_{min} = 25$, $D_{max} = 200$, $R_{max} = 1000$, $\alpha = 1.5$, $K_{qsz} = 0.5$. D_{min} and R_{max} dictate the amount of overhead for maintaining active links with no data to exchange whereas D_{max} determines the maximum communication period on any link. α decides how quickly r is increased or decreased when there is a change in traffic conditions and K_{qsz} is a factor by which d is increased when the link is busy. We use $tar\_u = 0.5$ for all the simulations except when other wise noted.



(a) CBR per-packet delay

(b) CBR per-packet conntime

(c) Delay v.s. conntime (4 hops)

Figure 8. End-to-end delay and per-packet conntime of a CBR flow

5.1. Effects of Number of Hops

In this subsection, we analyze how the number of forwarding hops along the communication path impacts throughput, latency, and energy consumption.

We start by looking at the available throughput for TCP flows between a random source and destination pairs in a scatternet containing 20 nodes. Figure 7 compares the throughput available to the application (noted as LCS) and the optimal offline throughput. For the one hop case, the offline optimal throughput is simply the maximum TCP data rate that can be achieved on a Bluetooth link using 512-byte packets. We can see that LCS achieves 79% of the optimal throughput. The main overhead of LCS in this case is maintaining other active links that do not have any data to exchange. (Recall that each idle link is scheduled for D_{min} time slots every R_{max} time slots.)

The optimal throughput is halved when the number of forwarding hops is more than one. This is because each bridge node that is forwarding packets needs to divide its time between receiving packets from one link and transmitting it over a different link. Ideally, all forwarding nodes will either be receiving or forwarding data simultaneously and hence, the optimal multi-hop throughput available will be one-half of the optimal one-hop throughput. As Figure 7 shows, the TCP throughput achieved by LCS is not far from the optimal. The distance from optimal increases only slowly as the number of forwarding hops increases.

The throughput achieved by LCS is superior to that achieved by PCSS described in [11]. The authors in [11] conducted simulations to measure the achieved throughput of a single TCP flow going through various numbers of nodes connected in a chain. The published results show that PCSS achieves about 145kbps for a 2-hop TCP flow compared with 240kbps achieved by LCS (see Figure 7). We note that the comparison will be much more meaningful if we have integrated PCSS in our Bluetooth simulator and we plan to do so in the future.

Figure 8(a) shows the end-to-end packet latency observed by a single CBR flow sending a 335-byte packet every $0.536s$ for various tar_u values. Recall that tar_u is the target utilization parameter that can be defined by applications. $tar\_u = 0.5$ means that LCS nodes along the forwarding path will be communicating twice as often as the data rate. As expected, the end-to-end latency increases as the number of hops increases and the lowest tar_u value yields the lowest latency.

To evaluate energy efficiency, we look at per-packet *conntime*–the ratio of the average time each busy node spends in the $Connected$ state communicating with neighbors to the number of data packets communicated during each simulation run. Since the amount of energy is needed for a node in active state to transmit, receive or listen for data packets is comparable, *conntime* is a good measure of energy efficiency.

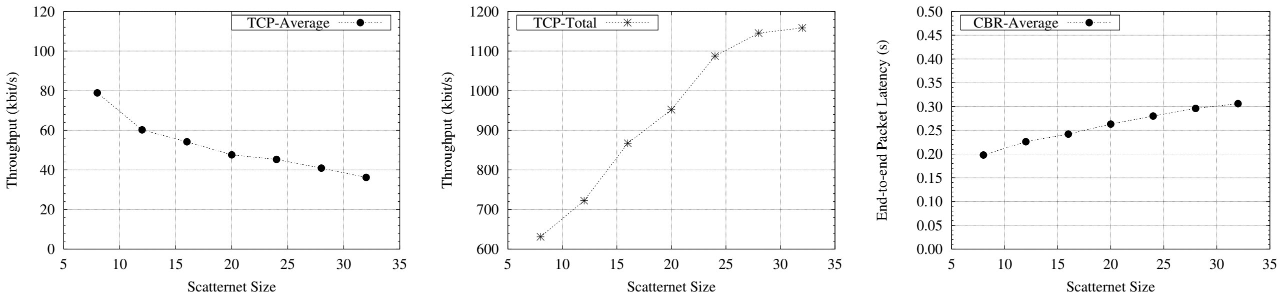
As shown in Figure 8(b), the curve for the smallest tar_u value yields the highest *conntime* since it stays active for the most amount of time to transfer the same amount of data. Figure 8(c) shows the tradeoff between end-to-end packet latency and per-packet *conntime* for 4-hop flows. By adjusting tar_r appropriately, applications can achieve the desired balance of packet delay and energy usage.

5.2. Effects of Scatternet Size

The size of the scatternet impacts both throughput and latency. The experiments covered in this section suggests that LCS scales well with the increase in scatternet size.

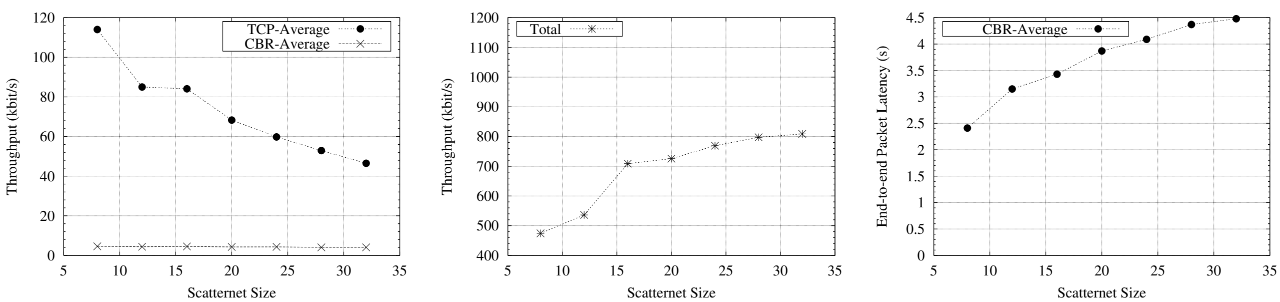
We evaluate this impact under the assumption that every node in the scatternet both sends and receives data. In particular, each node sends data to a random destination node and no two source nodes send to the same destination node. Thus, for a scatternet with n nodes, there are exactly n flows.

First, we examine the average throughput per flow when all flows are TCP flows. As shown in Figure 9(a), the av-



(a) Average throughput (b) Total throughput (c) Average packet latency

Figure 9. Throughput and latency achieved by all TCP flows and all CBR flows respectively



(a) Average throughput (b) Total throughput (c) Average packet latency

Figure 10. Throughput and latency achieved by mixed TCP and CBR flows

erage throughput goes down as the size of the scatternet increases. This is because as the scatternet size n increases the number of applications (also n) increases. Thus, the average throughput goes down. However, the total throughput increases with the increase in scatternet size (see Figure 9(b)). This interesting behavior is because the scatternet formation scheme we used produced topologies where average hop counts between any two nodes grows logarithmically as the scatternet size increases [13]. Since we setup n TCP flows from random source and destination pairs, the average path length of the flows also grows logarithmically with the increase in scatternet size. As a result, the number bottleneck links through which multiple flows go also increases logarithmically with the increase in scatternet size. The system-wide throughput improves, and in fact, the total throughput achieved by all the flows increases linearly with the increase in scatternet size.

We now examine the average end-to-end packet latency observed by all CBR flows. The setup is the same as the previous one except that each node now sends a 335-byte data packet every $0.536s$. Note that the end-to-end packet latency is dominated by the scheduling delay (see Section 4.4). The delay increases slowly as n increases (see Figure 9(c)). Again, this is because the average path length of the flows grows logarithmically with the increase in scatternet size. The average delay slowly increases with the increase in average hop count.

5.3. Effects of Mixed Traffic

In this section, we analyze how well LCS accommodates a mixture of bandwidth intensive and low-bandwidth but delay sensitive applications. We again setup a flow from each node to a random destination. For each scatternet of n nodes, there are $\frac{n}{2}$ TCP flows and $\frac{n}{2}$ CBR flows.

Figure 10(a) depicts the average throughput per flow achieved by each class of flows. Again, the average TCP throughput goes down as the total number of flows increases. Compared to Figure 9(a), the average throughput is higher since there are only $\frac{n}{2}$ TCP flows compared to n flows in Figure 9(a). However, the total throughput of n mixed flows as shown in Figure 10(b) is significantly less than that of n TCP flows in Figure 9(b) for every scatternet size. There are two reasons for this. First, when there are n TCP flows from n nodes, every link is busy exchanging data. But, when there are only $\frac{n}{2}$ TCP flows, some bridge nodes will have links that only carry CBR traffic and as a result, they devote some of their time to carrying non-TCP traffic. Note that each of those link is allocated at least D_{min} slots every meeting. Second, the presence of those links result in a higher degree of fragmentation in bridge nodes' meeting lists and thus, it becomes harder for LCS to coordinate links without some inefficiency. Nevertheless, it is clear that LCS maintains high throughput for the TCP flows while providing the necessary throughput required by the CBR flows.

Figure 10(c) shows the average packet latency achieved by the CBR flows. Compared to Figure 9(c), the average delay here is an order of magnitude worse. The reason for this is the increased buffering delay at each forwarding node as TCP packets fill up the link queues and CBR packets get stuck at the end of the queues. A simple solution to this problem is to employ some fair queuing algorithm such as DRR [12]. Since DRR guarantees max-min fairness, the packets from low bandwidth flows are forwarded shortly after being enqueued. Thus, the buffering delay at each bridge node will be close to zero and the average end-to-end packet latency perceived by the CBR flows will be mainly dependent on the scheduling delay and similar to the results shown in Figure 9(c).

6. Summary

LCS is a dynamic and distributed scheduling algorithm for scatternets. Nodes coordinate communication among neighboring nodes while allocating bandwidth based on current local traffic conditions. LCS optimizes the overall efficiency of the scatternet, in terms of throughput, latency and energy, by minimizing wasted and missed communication opportunities and piconet switches. It also allows nodes to tradeoff between energy efficiency and latency. Finally, LCS tolerates disruption in connectivity by providing a fall-back communication mechanism when nodes are not able to communicate during the agreed upon periods.

Although LCS can work with any scatternet topology, it is optimized for loop-free topologies. We have implemented LCS in a detailed Bluetooth simulator and conducted simulations over several different loop-free topologies of various sizes and with various traffic loads. The simulation results show that LCS achieves high TCP throughput and low packet latency and low node activity time (which corresponds to low energy consumption) for low bandwidth applications.

7. Acknowledgments

We would like to thank Allen Miu, Magdalena Balazinska, and Kyle Jamieson for their valuable insights during many discussions.

References

[1] S. Baatz, M. Frank, C. Kuhl, P. Martini, and C. Scholz. Adaptive Scatternet Support for Bluetooth using Sniff Mode. In *IEEE Conference of Local Computer Networks*, Tampa, FL, November 2001.

[2] Specification of the Bluetooth System. `http://www.bluetooth.com/`, December 1999. Bluetooth Special Interest Group document.

[3] J. Bray and C. Sturman. *Connection Without Cables*. Prentice Hall, 2001.

[4] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *Internetworking: Research And Experience*, 1:3–26, April 1990.

[5] J. Haartsen. The Bluetooth Radio System. *IEEE Personal Communications Magazine*, pages 28–36, February 2000.

[6] N. Johansson, U. Korner, and P. Johansson. Performance Evaluation of Scheduling Algorithms for Bluetooth. In *Fifth International Conference on Broadband Communications*, Hong Kong, November 1999.

[7] M. Kalia, D. Bansal, and R. Shorey. MAC Scheduling and SAR Policies for Bluetooth: A Master Driven TDD Pico-Cellular Wireless System. In *6th IEEE International Workshop on Mobile Multimedia Communications (MOMUC)*, San Diego, CA, November 1999.

[8] M. Kalia, S. Garg, and R. Shorey. Efficient Policies for Increasing Capacity in Bluetooth: An Indor Pico-Cellular Wireless System. In *IEEE Vehicular Technology Conference*, Tokyo, 2000.

[9] C. Law, A. K. Mehta, and K.-Y. Siu. Performance of a New Bluetooth Scatternet Formation Protocol. In *ACM Symposium on Mobile Ad Hoc Networking and Computing*, Long Beach, CA, October 2001.

[10] ns-2 Network Simulator. `http://www.isi.edu/vint/nsnam/`, 2000.

[11] A. Racz, G. Milklos, F. Kubinszky, and A. Valko. A Pseudo Random Coordinated Scheduling Algorithm for Bluetooth Scatternets. In *ACM Symposium on Mobile Ad Hoc Networking and Computing*, Long Beach, CA, October 2001.

[12] M. Shreedhar and G. Varghese. Efficient Fair Queuing using Deficit Round Robin. In *Proc. of ACM SIGCOMM*, August 1995.

[13] G. Tan. Self-organizing Bluetooth Scatternets. Master's thesis, Massachusetts Institute of Technology, Jan. 2002.

[14] G. Zaruba, S. Basagni, and I. Chlamtac. Bluetrees-Scatternet Formation to Enable Bluetooth-Based Ad Hoc Networks. In *IEEE International Conference on Communications*, pages 273–277, 2001.

[15] H. Zhu, G. Cao, G. Kesidis, and C. Das. An Adaptive Power-Conserving Service Discipline for Bluetooth. In *IEEE International Conference on Communications*, New York, NY, April 2002.