Title	A Novel MAC Scheduling Algorithm for Bluetooth System
Author(s)	Liu, C; Yeung, KL; Li, VOK
Citation	Conference Record / Ieee Global Telecommunications Conference, 2003, v. 1, p. 86-91
Issued Date	2003
URL	http://hdl.handle.net/10722/46454
Rights	©2003 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

A Novel MAC Scheduling Algorithm for Bluetooth System

Changlei Liu, Kwan L. Yeung and Victor O.K. Li

Department of Electrical and Electronic Engineering
The University of Hong Kong
Hong Kong, PRC.
Tel: (852) 2857-8493   Fax: (852) 2559-8738   E-mail: {clliu, kyeung, vli}@eee.hku.hk

Abstrat – **Data exchange within a Bluetooth piconet is master-driven. The channel/slot utilization thus depends on the efficiency of the scheduling algorithm adopted by the master. In this paper, a novel MAC layer scheduling algorithm, called Floating Threshold (FT), is proposed. Unlike existing approaches, FT allows the master to estimate the backlog queue status at each slave accurately based only on a single feedback bit and a floating threshold. The master can then derive an optimized packet transmission schedule. Using simulations, we show that FT outperforms existing algorithms in terms of channel utilization, packet delay and packet dropping probability.**

Keywords: Bluetooth, Media Access Control (MAC), Scheduling, Feedback, Time Division Duplex (TDD), Buffer Management

I. INTRODUCTION

Fueled by the desire for mobility and coupled with the demand for low-cost low-power connection between different electronic devices, Bluetooth [1], a fast frequency hopping Time Division Duplex (TDD) wireless system, has undergone rapid development in recent years. To resolve the collisions over wireless medium, data exchange in Bluetooth is designed to be master-driven. A *master* is a Bluetooth unit that initializes the setup of a *piconet*. Each master can simultaneously communicate with up to seven other units, or slaves. Fig. 1 shows a piconet consisting of 1 master and 5 slaves. The configuration of master-driven TDD scheme vests the master with the task of scheduling; this simplifies the implementation of a Bluetooth system.

The scheduling resources for the master are slots of 625μs in length and each corresponds to a hop frequency. Data transmission in Bluetooth is characterized by full duplex coupling: a slave is allowed to begin its transmission to the master only after receiving a packet from the master (or just being polled by the master with a null packet) in the preceding slot. In other words, the transmission always occurs in pairs.

In [1], two types of RF links have been defined: Asynchronous Connectionless Link (ACL) and Synchronous Connection-Oriented (SCO) Link. SCO link is applied for time-bound services such as voice, with TDD frames reserved at regular intervals. ACL links support packet-switched, point-to-point or point-to-multipoint data connections. In this paper, we are only interested in scheduling packets on ACL links. An ACL link can support packets with three different sizes, 1, 3, or 5 time slots.

It can be seen that slot wastage occurs (on the ACL links) in the following situations:

(1) When both the master and the slave do not have packets destined to each other, both the polling slot from master-to-slave and the reply slot from the slave-to-master will be wasted. The resulting wastage is said to be 100%.

(2) When either the master or the slave does not have packets destined to the other side, the wastage is 50% (assuming the same packet size in each direction).
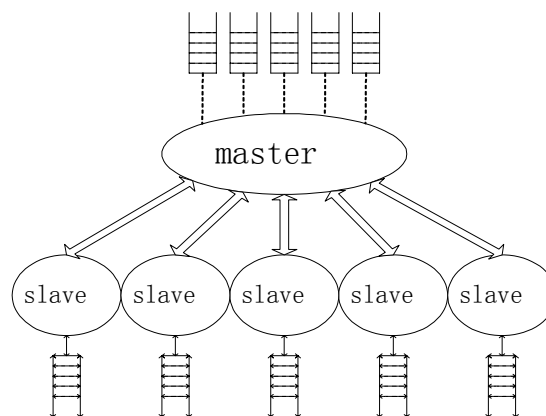

Fig. 1: A piconet consisting of one master and five slaves.

Therefore, a good MAC (Medium Access Control) scheduling mechanism should minimize such slot wastage. At the same time, it is desirable to have small packet delay, and a low packet dropping probability (as a result of buffer overflow).

In this paper, we focus on designing MAC layer packet scheduling algorithm in a piconet. A comprehensive review of the existing MAC layer scheduling algorithms is conducted in the next section. In Sections III & IV, a novel MAC scheduling algorithm, called Floating Threshold (FT), is proposed. FT allows the master to determine (with high accuracy) the backlog queue status at all slaves, based only on a single feedback bit and a floating threshold. (Note that it is not possible to have the exact slave queue size based only on a single feedback bit.) An optimized packet transmission schedule can then be determined. The performance of the FT algorithm is evaluated via simulations in Section V. It is found that FT outperforms the existing algorithms in throughput, delay as well as packet dropping probability. Finally, we conclude the paper in Section VI.

II. EXISTING MAC SCHEDULING ALGORITHMS

In a Bluetooth piconet, the master is vested with the task of scheduling. The most primitive scheduling algorithm works in a Round Robin (RR) manner, i.e., the slots are allocated to slaves in a strict cyclic order. The bandwidth is equally divided but the slot wastage is excessive because slots will be wasted if a slave with an empty queue is polled. As a result, RR generally performs poorly.

A good Bluetooth MAC scheduling algorithm should be designed based on the current traffic status (i.e. the amount of backlog traffic) at both the master and all of its slaves. The queue status at the master for packets destined for each slave can be readily obtained. The problem is to collect the queue status from the slaves. To achieve this, two approaches are generally followed: history-based [5,6] and feedback-based [2-4]. In the former approach, the master uses historical data collected from the slaves to predict the possible arrival of new packets in the subsequent time slots at the slaves. In [5,6], various polling schemes following this approach are examined and compared.

The feedback-based approach makes use of the explicit feedback information carried by the packet transmitted from the slave to the master (i.e. upstream packets) to update the slave queue status stored at the master. As compared with the history-based approach, more accurate estimation (if not exact) of the slave queue size can be obtained. Such feedback mechanisms are adopted in [2-4]. Studies showed that feedback-based scheduling algorithms outperform those following the history-based approach, but at the expense of possible feedback overhead.

To be more specific, two history-based polling schemes are proposed in [6], namely, Limited Exhaustive Round Robin (LERR) and Limited Weighted Round Robin (LWRR). LERR has a fixed cyclic order and the scheme is exhaustive, i.e., the master does not switch to the next master-slave pair until both the master and the slave queues are emptied. Parameter "*t*" is used to limit the number of transmissions (tokens) that can be performed by each pair per cycle. This in return limits the cycle length and avoids slave starvation, i.e., the denial of transmissions to/from a slave. The LWRR adopts a weighted round robin algorithm with weights dynamically changed according to the observed queue status. Each slave is assigned a weight equal to MP (Maximum Priority) at the beginning, and this weight varies within the range of 1 to MP subsequently. Details about the two schemes can be found in [6].

In [3], a feedback-based scheduling algorithm called Priority Policy (PP) is proposed. PP uses a single feedback bit to convey the slave queue size information to the master. This single bit is piggybacked on the upstream packets. Due to the single feedback bit, a slave can only indicate whether its current queue size is empty ('0') or backlogged ('1'). Using PP,

the master classifies all master-slave pairs into one of the four states, namely, 1-1, 1-0, 0-1 and 0-0. As an example, 1-0 means the master has packets for the slave, whereas the slave does not have packets for the master. The highest transmission priority, denoted by p, is then given to the 1-1 pair which has 100% slot utilization. PP is known to have the potential problem of slave starvation. Suppose a master-slave pair is classified as belonging to 0-0, the master may never poll the slave again (for getting slave queue size update) if other pairs always have packets to send.

Another feedback-based scheduling scheme, called HOL-Priority Policy (HOL-PP) [2], is also proposed. In HOL-PP, packets may be of different sizes, 0, 1, 3, or 5 time slots. Packet size 0 means the queue is empty. The master schedules based on the head of line (HOL) packet size at both the slave and master. Two feedback bits are required to differentiate the 4 possible HOL packet sizes. Like [3], high system throughput (or, channel utilization) can be obtained.

More recently, another type of feedback-based MAC scheduling algorithms aimed at optimizing TCP performance over Bluetooth systems is proposed in [4]. Instead of monitoring the MAC layer buffer size, the flow bit in the payload header (please refer to [1] for the details of the Bluetooth packet format) is adopted to monitor the L2CAP protocol layer buffer occupancy, where L2CAP is on top of the MAC layer. In [4], the flow bit is set to '1' by the slave if its L2CAP buffer size exceeds a pre-defined threshold. Then based on the flow bit status at both the master and slave, the polling interval is calculated. Since the threshold for the L2CAP buffer is preset, the polling schemes will degenerate into Round Robin if the traffic load is below the threshold. Therefore, the value for the threshold should be carefully chosen. In [4], the optimal threshold value is obtained by trial and error from simulations. Good heuristics for threshold setting (or even adaptive threshold setting) are needed.

Since [4] focuses on transport layer optimization, the direct impact of a MAC scheduling scheme on the system performance is not clear. In this paper, we concentrate on the MAC layer scheduling as in [2,3]. This can avoid distraction caused by, e.g. TCP congestion control on the link layer performance.

III. FLOATING THRESHOLD ALGORITHM

In this section, based on the single bit feedback mechanism, a novel scheduling algorithm, called Floating Threshold (FT), is proposed. We use the *flow bit* in the payload header [1] to carry the feedback information from each slave. As compared with PP [3], FT allows the master to get *more* accurate slave queue size estimation. This is achieved by maintaining a floating threshold on the current queue size of each slave at the master. Floating threshold represents the *minimum* queue size currently at a slave. Note that queue size is measured by time slots, e.g. a 3-slot packet will occupancy a 3-slot buffer.

To be more specific, for each master-slave pair, the current floating threshold on the slave queue size is stored at the master using a variable T_{master}, which is updated according to the value of the feedback bit generated by the slave. (We will introduce the threshold updating rules shortly.) At the slave side, the slave sets the feedback bit according its copy of the floating threshold T_{slave}, and its instantaneous queue size. It should be noted that the values of T_{master} and T_{slave} must be synchronized via feedback bits.

Without loss of generality, we assume that before the actual data exchange starts, T_{master} and T_{slave} are initialized to 0. As new packets destined for the master arrive at the slave queue, if they cannot be sent to the master in time, they will queue up in the slave's buffer. Each time the slave sends a packet to the master, it updates its T_{slave} and sets the feedback bit accordingly. The feedback bit is piggybacked to the master to inform the master about the update at the slave. When the master receives the packet, based on the feedback bit value, T_{master} is updated such that the values of T_{master} and T_{slave} are synchronized.

Let the step-size for updating the floating threshold be m slots and the size of the packet be s slots. The set of rules for floating threshold update at both master and slave is summarized below. In the next section, an adaptive scheme will be designed to dynamically adjust the value of m for better performance.

Floating Threshold Updating Rules:
1. When a slave is ready to send a s-slot packet to the master
    a) If its queue size is greater than $(T_{slave} + m)$ (after the current transmission), send the packet with feedback bit = 1; update $T_{slave} = T_{slave} + m$. (Note that the current queue size remains greater than the updated T_{slave}.)
    b) Otherwise, send the packet with feedback bit = 0; update $T_{slave} = \max\{T_{slave} - s, 0\}$. (Since the queue size will decrease by s slots after transmission, T_{slave} is updated correspondingly. Still the current queue size will be no smaller than the updated threshold T_{slave}.)
2. When a s-slot packet is received by the master
    c) If the piggybacked feedback bit = 1, the master updates $T_{master} = T_{master} + m$ and it knows that the queue size at this slave is at least $(T_{master} + 1)$ slots.
    d) Otherwise, the master knows that at the time this feedback bit was generated, the slave queue size is at least T_{master} slots; then it updates $T_{master} = \max\{T_{master} - s, 0\}$. (This is because the slave queue size has decreased by s slots after transmission.)

From the above set of updating rules, we see that the master estimates the slave queue size based on the feedback bit (which carries the information about the slave queue size when this bit was generated) and the historical data kept by original T_{master}. Although this estimation may not exactly reflect the current slave queue status, we indeed extend our knowledge

about the queue size as compared with the pure binary feedback scheme used in [3]. This extended knowledge is proved to be useful by extensive simulation results in Section 5.

Using the above set of updating rules, the floating threshold T_{master} (as well as T_{slave}) will never exceed the current queue size at the slave. In this regard, we are conservative because we just try to find a *lower bound* which is as close to the actual slave queue size as possible. We have ignored the packets that arrived at the salve queue after the current feedback bit was generated. An alternative approach is to take such possible packet arrivals into consideration. We have indeed investigated along this direction but found no obvious performance gains. As such, we choose not to include those results here.

With more accurate knowledge on individual slave queue sizes, the master can then decide which slave (or packet) should be polled (or sent) next. In our FT algorithm, the scheduling algorithm is similar to the original PP scheme. First, all master-slave pairs are classified based on their slot utilization. If both master and slave have packets for each other, the slot utilization is 100%. If only one side has packet for the other, the slot utilization is 50%. If none of them have packet to send, then there is no need to consider that pair. Scheduling priority is given to the node pairs with 100% slot utilization. If there are more than one node pairs in this category, break the tie by choosing the node pair that has the largest sum of the master queue size and the *estimated* slave queue size T_{master}. (Note that for the PP algorithm [3], node pairs in the same class are served in a round robin fashion.) If all node-pairs with 100% utilization have been served, node pairs with 50% utilization are considered in a similar fashion.

To avoid the starvation of low priority pairs (i.e. those labeled with 50% or 0 slot utilization), a forced slave update-period is devised to urge the master to visit all slaves *at least* once in every u slots. In other words, parameter u functions like the guaranteed polling interval as defined in the Bluetooth specification [1]. This helps the master to update its knowledge about each slave; otherwise the information maintained by the master would become obsolete if certain slaves are ignored for a long time. The value of u can be pre-determined or negotiated by the Link Manager [1] protocol during the connection setup phase.

IV. ADAPTIVE STEP-SIZE DESIGN

The performance of the Floating Threshold (FT) algorithm depends on the value of the step size m. If m is fixed (as we have assumed before), it is hard, if not impossible, to find a single value of m that remains optimal/good in all scenarios.

Generally speaking, if m is small and the variation in slave queue size is large (e.g. due to bursty traffic), it will take a long time for the floating threshold at the master (T_{master}) to catch up with the actual queue size since each feedback can at most increase T_{master} by m. On the other hand, a large m tends

to be aggressive and this may give a coarse estimation of the actual queue size.

It can be seen that FT achieves the best performance by dynamically adjusting its step size value according to some relevant factors, such as the gap/error between the current estimation and actual length, the possible packet arrival rate, etc. However, such information is not readily available to the master. To have a simple yet efficient way of adjusting the step size m, we propose to tune m in a probing fashion based on the current floating threshold value T_{master}.

When T_{master} is increased at an *accelerating* rate, m should also be increased to track the actual queue size faster. On the other hand, when the estimated threshold value approaches the actual slave queue size, a finer/smaller value of m should be used. This way, a good estimation of the actual queue size at a slave can be obtained. In the following, we combine the step size adaptation scheme together with the floating threshold updating rules.

Modified Floating Threshold Updating Rules:

1. When a slave is ready to send a s-slot packet to the master
   a) If the last feedback bit sent = 1, $m = \max(2^{\lfloor \log_2(T_{slave} - T'_{slave}) \rfloor - 1}, 1)$
   b) Otherwise, $T'_{slave} = T_{slave}$; reset $m = 1$.
   c) If the slave's queue size is greater than $(T_{slave} + m)$ (after the current transmission), send the packet with feedback bit = 1; update $T_{slave} = T_{slave} + m$.
   d) Otherwise, send the packet with feedback bit = 0; update $T_{slave} = \max\{T_{slave} - s, 0\}$.

2. When a s-slot packet is received by the master
   a) If the piggybacked feedback bit = 1,
      (i) If the last piggybacked feedback bit received = 0, $T'_{master} = T_{master}$; reset $m = 1$.
      (ii) Else $m = \max(2^{\lfloor \log_2(T_{master} - T'_{master}) \rfloor - 1}, 1)$
      $T_{master} = T_{master} + m$; the master knows that the queue size at the corresponding slave is at least $(T_{master} + 1)$ slots.
   b) Otherwise, $m = 1$; the master knows that at the time this feedback bit was generated, the corresponding slave's queue size is at least T_{master} slots; then it updates $T_{master} = \max\{T_{master} - s, 0\}$ and $T'_{master} = T_{master}$.

In Step 1 above, sub-steps (a) and (b) are added for dynamically setting the value of m. Similarly, sub-steps 2.a)(i)&(ii) are added at the master side for synchronizing the values of m with the slave.

Next we explain the idea behind our adaptive step size design by focusing on the operations at the master. (The operations at the slave can be argued similarly.) Consider a stream of *consecutive* feedback bits with values equal to 1 that are

generated by the slave and subsequently received by the master. (This implies that the slave queue size is increasing rapidly.)

When the first feedback bit = 1 of the stream arrives, step size $m = 1$ is used. For the subsequently arrived feedback bits of value 1, m is adjusted according to $m = \max(2^{\lfloor \log_2(T_{master} - T'_{master}) \rfloor - 1}, 1)$. Note that T_{master} is the instantaneous threshold value at the master when the current feedback bit arrives, and T'_{master} is the *old* threshold value that was taken just before the first feedback bit = 1 arrived. Their difference is the minimum increase in queue size at the slave during the measured period (i.e. from the moment the first feedback bit = 1 arrived, to the moment the current feedback bit = 1 arrives). The larger this difference is, the larger value of m should be used in order to closely track the actual slave queue size. Setting $m = 2^{\lfloor \log_2(T_{master} - T'_{master}) \rfloor - 1}$ is roughly equivalent to setting m to be half of the difference between T_{master} and T'_{master}.

If the next feedback bit arrived at the master has a value of 0, that means the current slave queue size becomes less than $(T_{slave} + m)$ slots. In other words, the estimated queue size is now very close to the actual value. So we reset m to 1 to slow down the speed of in creasing the floating threshold. As long as the subsequently arrived feedback bits are equal to 0, m remains at 1.
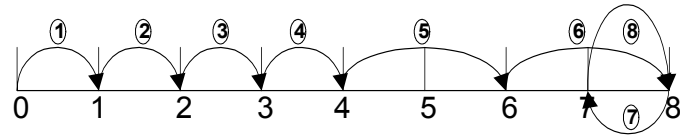

Fig. 2: An example using adaptive step size m.

As an example, Fig. 2 delineates the process of adapting step size m. The x-axis denotes the floating threshold T_{master}. When a feedback bit arrives, the master updates its floating threshold T_{master} according to the rules we defined earlier; this corresponds to a curved arrow transition in Fig. 2. The range spanned by the arrow on the axis indicates the value of m being used for that update. The number (in the circle) on top of each curved arrow indicates the order of feedback bit arrivals.

Assume that the slave has 16 one-slot packets in its queue and no more packets will arrive. Let $T_{master} = T'_{master} = 0$. To update the floating threshold at the master, a stream of six consecutive "1" feedback bits will arrive at the master, following the order from ① to ⑥. Using the adaptation rule 2.a), the sequence of the step size used for each update is {1, 1, 1, 1, 2, 2}, and the resulting $T_{master} = 8$.

At the slave side, in determining the value for the next feedback bit to be piggybacked, the slave checks if $T_{slave} + m$ is smaller than the current queue size. Note that $T_{slave} = T_{master} = 8$, and the value for m is now 4 according to Step 1.a). So $T_{slave} + m = 12$. At this moment there are 10 packets left in the slave

queue (6 packets has already been sent to the master), which is less than 12. The slave sends back a feedback bit 0.

When this bit arrives at the master, the master realizes that its T_{master} is very close to the actual slave queue size, thus m is reset to 1 and $T_{master} = T_{master} - 1$ (as indicated by arrow ⑦). At the same time, T'_{master} is updated to 7 according to Step 2.b).

V. PERFORMANCE EVALUATION

A. Simulation Model

The performance of the proposed FT algorithm is evaluated based on the piconet shown in Fig. 1, which consists of one master and five slaves. (We have also tried other configurations and the results are consistent and thus not shown here.) For performance comparisons, the following MAC layer scheduling algorithms are implemented:

- Floating Threshold (FT) with adaptive step size m, and forced slave update period $u = 100$ slots. (We have simulated other values of u and found that the performance of FT is generally not sensitive to u if u is set larger than, e.g. 50.)
- Simple Round Robin (RR) scheduling.
- Priority Policy (PP) [3] with fairness parameter $p = 4$, and forced slave update interval $u = 100$ slots. Note that the original PP algorithm does not have this forced slave update interval. We incorporate this into PP to enhance its performance and thus to have a fairer comparison.
- Limited Exhaustive Round Robin (LERR) algorithm [6] with t, the number of transmissions that can be performed by each master-slave pair per scheduling cycle, set to 4.
- Limited Weighted Round Robin (LWRR) algorithm [6] with MP (maximum priority) set to 4.

The above parameter values are recommended either implicitly or explicitly by the original authors of the respective papers.

Assume the packets arrive at each queue in bursts (e.g., an IP datagram will be segmented into a burst of L2CAP packets for transmission over a piconet), following a Poisson process with a mean of λ bursts per time slot. Let the burst size be geometrically distributed with a mean of 2 packets. Since there are 10 traffic sources/queues in the simulated network, the total system load (bi-directional communications) is given by 10 x 2 x λ = 20λ packets/slot. The packet size is of 1, 3, or 5 time slots with equal probability.

Each point of simulation data shown in Figs 3-5 is collected by simulating 50000 time slots with the initial 1000-slot statistics ignored.

B. Delay vs channel utilization curve.

First we consider the case that the buffer sizes at both master and slave are sufficiently large, so no buffer overflow will occur. Fig. 3 shows the average packet delay (which is measured from the moment a packet arrives at a queue until it is successfully sent) against the channel utilization, i.e. the percentage of time that the channel is busy in carrying data packets. (One may also consider the channel utilization as the system throughput.) By varying λ, we can vary the total system load, and thus alter the channel utilization.

From Fig. 3, we can see that FT and PP, with the cost of feedback, outperform the two history-based polling schemes and the simple round-robin (RR). As expected, RR shows the worst performance because it does not consider the slave queue size in scheduling. For the two feedback-based algorithms, FT consistently outperforms PP and the performance gap becomes larger when the channel utilization (i.e. the system load) is high. This improved performance is due to the enhanced slave queue size estimation mechanisms we introduced in this paper.
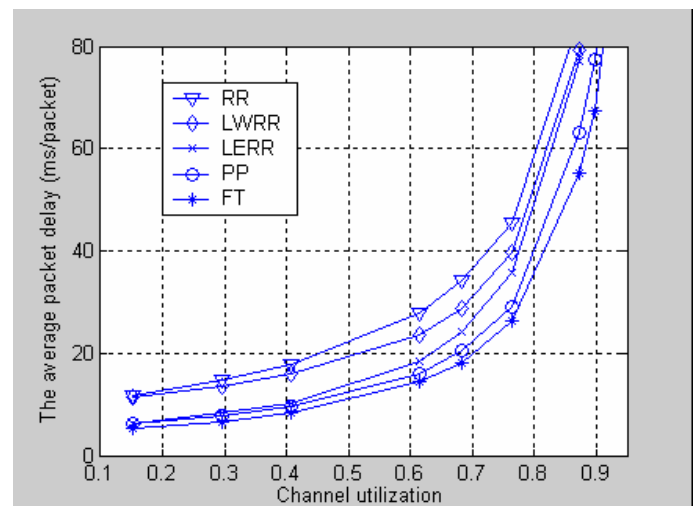

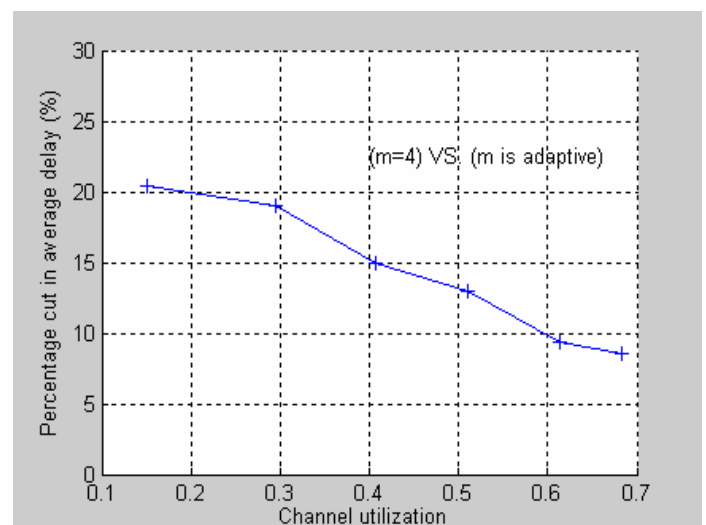Fig. 3 Mean packet delay against channel utilization.

Fig. 4 Percentage cut in delay performance when adaptive step size is used.

To justify the adoption of adaptive step size in our FT algorithm, we compare the performance of using adaptive step size with that of a fixed step size. Fig. 4 shows the percentage cut in average packet delay if adaptive step size is used instead of fixing $m = 4$ slots. The same traffic model and parameters as that in obtaining Fig. 3 are used. We can see that with adaptive step size, the average packet delay can be cut down from 8% to 22%. It is interesting to note that the performance gap decreases as the channel utilization increases. This is expected because when the traffic load is light (i.e. low channel utilization) a finer/smaller m is desirable (thus $m = 4$ is too aggressive), while as the traffic increases (i.e. channel utilization is high) a large value of m is preferred for tracking the slave queue size.
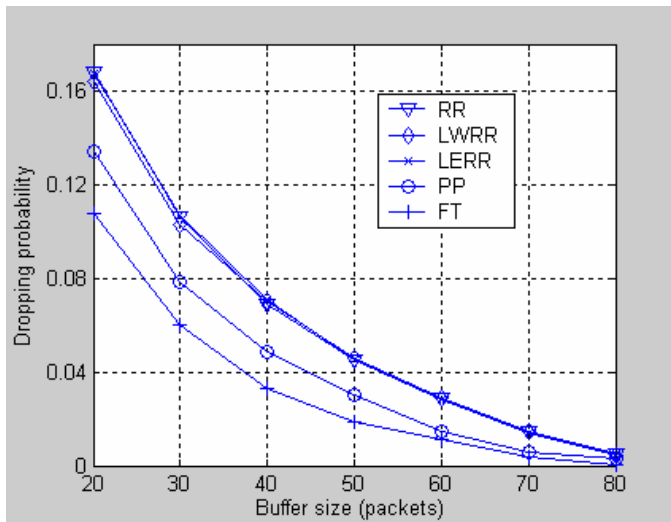
C. Dropping behavior

Fig. 5: The effect of buffer size on the packet dropping probability

Bluetooth is likely to run on small electronic devices, in which memory resources may be scarce, such as Personal Digital Assistants. Packets will be dropped as a result of buffer overflow. By varying the amount of available buffers, we study the packet dropping behavior of using different scheduling algorithms in Fig. 5. Without loss of generality, we assume all packets are of size one time slot. The buffer size can thus be explicitly measured as the number of packets. The total system traffic load is set to 1 packet/slot by having $\lambda = 0.025$ bursts/slot; the other parameters are the same as that in obtaining Fig. 3.

As expected, the FT algorithm is a clear winner and it has the lowest packet dropping probability for all simulated buffer sizes. This is because FT always gives scheduling priority to the master-slave pair with the longest queue size (as derived from the more accurate estimation process). This helps to minimize the chance of buffer overflow. In contrast, history-based algorithms tend to have a coarse estimation on slave queue size, and thus their packet dropping probabilities are higher than feedback-based approaches (PP and FT).

Note that the results we presented above are obtained using a symmetric traffic model, i.e. the same traffic load to all queues. Results on various non-symmetric traffic models have also been obtained. We found that the performance gain in using FT algorithm is even more pronounced in those cases. This is because for other schemes, the packet dropping probability increases dramatically due to the non-uniform distribution of the queue size. In contrast, FT is less adversely affected since longer queues are given more chance to send. Due to space limitation, such results are not included here.

VI. CONCLUSIONS

In this paper, a novel scheduling algorithm, called Floating Threshold (FT), was proposed for data exchange within a Bluetooth piconet. By keeping a floating threshold for each master-slave pair, the master can have more accurate estimation of the slave queue size based on the binary feedback information. The resulting scheduling decision is then optimized. As compared with other existing schemes, FT was shown to exhibit better system performance in terms of channel utilization, packet delay, and packet dropping probability. With minor modifications, the proposed algorithm can be easily extended to other centralized full-duplex systems, such as wireless multiple access, input-output buffered switch designs, etc.

On the other hand, designing efficient scheduling algorithms for Bluetooth scatternet [7] is probably more challenging. We are interested in extending the concept of our FT algorithm to support scatternet-wide scheduling.

ACKNOWLEDGMENT

REFERENCES

[1] Bluetooth Special Interest Group, "Specification of the Bluetooth System 1.1," http: //www.bluetooth.com/

[2] M. Kalia, D. Bansal and R. Shorey, "MAC scheduling and SAR policies for Bluetooth: a master driven TDD pico-cellular wireless system," IEEE International Workshop on Mobile Multimedia communications, 1999 (MoMuC '99), pp. 384 –388

[3] M. Kalia, D. Bansal and R. Shorey, "Data scheduling and SAR for Bluetooth MAC," Vehicular Technology Conference Proceedings, 2000, VTC 2000-Spring, Tokyo. Page(s): 716 -720 vol.2

[4] A. Das, A. Ghose, A. Razdan, H. Saran and R. Shorey, "Enhancing performance of asynchronous data traffic over the bluetooth wireless ad-hoc network," IEEE INFOCOM 2001

[5] N. Johansson, U. Korner and P. Johansson, "Performance Evaluation of Scheduling Algorithms for Bluetooth," IFIP TC Fifth International Conference on Broadband Communications 99, Hong Kong.

[6] Antonio Capone, Mario Gerla and Rohit Kapoor, "Efficient Polling Schemes for Bluetooth picocells," IEEE ICC2001.

[7] C.L. Liu and K.L. Yeung, "A simple adaptive packet scheduling scheme for Bluetooth Scatternet," IEEE VTC 2003, Orlando, USA, Oct., 2003.