Available online at www.sciencedirect.com

ScienceDirect

Computer Communications 30 (2007) 1569–1587

computer communications

www.elsevier.com/locate/comcom

Practical channel state aware and cooperative packet scheduling disciplines for coordinating colocated Bluetooth and IEEE 802.11b devices

Hoi Kit Yip, Yu-Kwong Kwok *,1

Department of Electrical and Electronic Engineering, The University of Hong Kong, Pokfulam Road, Hong Kong

Received 28 September 2005; received in revised form 9 January 2007; accepted 17 January 2007
Available online 23 January 2007

Abstract

Attempts to satisfy the demand for ubiquitous communications have resulted in a proliferation of hand-held short range communication devices based on the ISM (Industrial, Scientific, Medical) band technologies, most notably Bluetooth and IEEE 802.11b. However, coexistence between Bluetooth and IEEE 802.11b has become a critical issue that could severely hinder the performance achieved by user devices. In this study we performed a detailed implementation of a Linux based network access point (NAP), in which Bluetooth and IEEE 802.11b interfaces are colocated. Such an NAP is crucial in supporting "hot-spot" systems targeted to serve nomadic users carrying either a Bluetooth or an IEEE 802.11b device. Specifically, the goal of our study is to investigate the efficacy of a software-based interference coordination approach, through a detailed *actual implementation* so as to identify system issues which are difficult to obtained by simulations.

We considered a wide range of common scheduling algorithms as the possible solutions in a Linux environment to estimate the interference effects as viewed from the network layer perspective. Upon our investigation, two wireless scheduling algorithms based on Channel State Independent Fair Queueing (CIFQ) were implemented in Linux to test their empirical performance under this NAP application. Finally, guided by our practical findings, we proposed and implemented two new packet scheduling algorithms in Linux to provide the best trade-offs to colocated Bluetooth and IEEE 802.11b traffics, as well as QoS support for different applications. Our results show that dynamic priorities and cooperative transmissions between Bluetooth and IEEE 802.11b traffic can effectively protect both interfaces from interference. We also compared our proposed scheme with two MAC layer approaches.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Coexistence; Linux; Queueing disciplines; Scheduling; Network access point (NAP); Dual-mode wireless access; Wireless routers; Bluetooth; IEEE 802.11b

1. Introduction

The unprecedented demand for ubiquitous personal communications has boosted the great advancements of various short range wireless communication technologies. In particular, two of such technologies, namely Bluetooth [12] and IEEE 802.11b [4], are particularly important due to their rapid proliferation in various mobile electronic devices such as mobile phones, PDAs, notebooks, etc. This phenomenon stems naturally from their physical layer designs based on the license-free 2.4 GHz ISM (Industrial, Scientific and Medical) frequency spectrum, leading to low cost of deployment.

However, with the high popularity of both Bluetooth and IEEE 802.11b, it would be of tremendous commercial interest to build a *general* network access point (NAP) which can interface with these two technologies at the same time. Indeed, this *dual-protocol* NAP could enhance "hot-spot" systems which are already highly popular in many

* Corresponding author. Tel.: +852 28598059; fax: +852 25598738. *E-mail address:* ykwok@hku.hk (Y.-K. Kwok).
[1] This research was supported by a grant from the Hong Kong Research Grants Council under project Nos. HKU 7162/03E and HKU 7157/04E.

cyber-cafes, restaurants, shopping malls, airports, conventions centers, etc. Such a dual-protocol NAP can transparently serve nomadic users carrying either a Bluetooth or an IEEE 802.11b device, and most importantly, can allow a seamless bridging of these two groups of users.

In this scenario, transceivers of both Bluetooth and IEEE 802.11b devices will "coexist" in close proximity. Such coexistence, however, if not coordinated carefully, would inevitably lead to unwanted and possibly detrimental interference between them. Specifically, this unwanted interference may cause severe performance degradation (i.e., low effective bandwidth) of both wireless technologies, as demonstrated by recent research results in [17,19] and [35].

Many academic researchers and practitioners in the commercial sectors have worked on a number of research projects to tackle the coexistence problem [13,17,24,32]. The significance of this interference problem is further recognized with the establishment of the Coexistence Task Group 2, which is set up by and coordinated from Bluetooth SIG (Special Interest Group) and IEEE 802.15 to fight against this problem [40]. Previously proposed solutions can be broadly divided into two categories: *Collaborative* and *Non-collaborative*.

Collaborative mechanisms [13] require that Bluetooth and IEEE 802.11b transceivers communicate with each other about their traffic to avoid interference. For instance, in MEHTA (MAC Enhanced Temporal Algorithm) proposed in [25], traffic information is exchanged between the Bluetooth and IEEE 802.11b firmware to calculate the accurate timings at the MAC layer, avoiding interference by proper synchronization. Technically, it is an interference avoidance scheme in the *time* domain. Currently there are many commercial products using this kind of collaborative coexistence approach, namely the Blue802 by Silicon Wave and Intersil [34], TrueRadio by Mobilian [29], Wireless Coexistence System (WCS) by Intel [14], a Coexistence Package by Texas Instruments [37] and UltimateBlue by Silicon Wave and Intersil [33].

Non-collaborative mechanisms [19] do not involve such communications but the transceivers sense the existence of other type of wireless transmissions by estimating the channel conditions frequently. For example, the Adaptive Frequency Hopping (AFH) approach described in [18] classifies frequency channels as *good* or *bad* according to the probabilities of interference, and only the *good* ones are allowed for the Bluetooth to use.

The aforementioned coexistence mechanisms all suffer from one realistic drawback. With these schemes, modifications of the firmware and/or the hardware of the Bluetooth and IEEE 802.11b transceivers are required. However, with the wide spread use of these two technologies nowadays, there are many existing Bluetooth and IEEE 802.11b transceivers that cannot use such coexistence approaches.

Using a collaborative approach, our work aims to build a prototype of the aforementioned dual-protocol network access point, *without* requiring modifications in the client devices. We believe that in order to obtain useful insights in the interference effects, we need to deal with the coexistence problem of Bluetooth and IEEE 802.11b using an experimental approach.

In our study, we choose the open source developed, network ready, and inexpensive Linux operating system [2] as the platform for the dual-protocol NAP prototype. Our motivation is that a Linux based dual-protocol NAP would be much more cost-effective for networking devices manufacturers. We make use of the BlueZ [11] suite as the Bluetooth support in Linux. BlueZ supports the Bluetooth protocol stack from device drivers, Linux kernel interface, to the protocol stack just below the network layer. Thus, it is commonly referred to as the official Linux Bluetooth protocol stack. However, its inability in *channel state extraction* from the Bluetooth device hardware was a major technical difficulty encountered in this study. In our implementation, we bring the Bluetooth link up at the IP layer with the Bluetooth Encapsulation Protocol (BNEP) and the PAN profile implementation of BlueZ. When the Bluetooth link is at the IP layer, both the master (the NAP in our study) and the client have a network interface named `bnep0`.

The Linux support for IEEE 802.11b is not as centralized as the case in Bluetooth. At the Linux kernel [9,10,15], wireless extensions are developed by Jean Tourrilhes [38,39]. The extensions are implemented as a generic API in Linux kernel, allowing an IEEE 802.11b device driver to connect to the user space configuration and statistics. Our implementation benefits considerably from this existing software architecture which has inspired us on a method of channel state extraction from WLAN device drivers. For device drivers, there are three types: the HostAP [27], the WLAN-NG [1] by AbsoluteValue Systems, and the early IEEE 802.11 drivers embedded in the Linux kernel. The last two groups are irrelevant in this study as we do not employ those corresponding IEEE 802.11b devices.[2]

The rest of the paper is organized as follows. In Section 2, we review some packet scheduling techniques implemented in the Linux platform. We also review the well-known Channel State Independent Fair Queueing (CIFQ), which was first implemented in Linux in our study to conduct performance measurement reference tests. A detailed presentation of our proposed packet scheduling algorithms is given in Section 3. Section 4 describes in detail our testing environment and experiment configurations, and wireless transmission scenarios. Performance results are presented and discussed in Section 5. Finally, we conclude in Section 6.

2. Overview of practical packet scheduling mechanisms

2.1. Packet scheduling in linux

Linux supports packet scheduling by its traffic control functions [6,31], which are implemented inside the Linux

[2] The Linux literature prefers to use the term "WLAN" to refer to the IEEE 802.11b. In the subsequent sections of this paper, we will use these terms interchangeably.

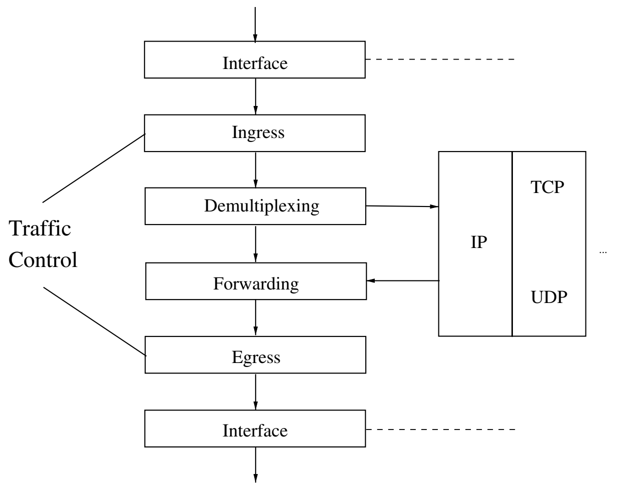


Fig. 1. Traffic control in Linux.

kernel. In the Linux networking model, each network transceiver is treated as an *interface*. For example, *eth0* is the system software reference name for the first Ethernet networking *device* (i.e., hardware) in the system. The high scalability of Linux enables it to have several such interfaces activated at the same time. Linux handles all the interfaces in a round robin manner. For each interface, Linux provides four main services: Ingress, Demultiplexing, Forwarding, and Egress.

Fig. 1 shows how the Linux kernel provides the four services to packets in a round robin manner. The packets enter the host at the Ingress side and leave at the Egress side. In particular, Linux traffic control manifests at the Ingress and Egress operations. However, at the Ingress, Linux provides a limited set of operations such as removal of undesired packets or preliminary packet classification. At the Egress, the *full range* of traffic control is available, including *packet scheduling*[3] [3].

A user space utility called `tc` is required for Linux traffic control. This `tc` program can be obtained from the `IPROUTE2` package [23], as instructed in Linux Advanced Routing and Traffic Control Project [22]. Fig. 2 shows the relationship of `tc`, the Linux kernel and the user space.

Traditionally, packet scheduling is the problem of prioritizing packet transmissions which share the *same* network interface. However, in our study, it involves packet scheduling on more than one interface, and thus, creating a new dimension in packet scheduling. This is also motivated by McHardy's work, Intermediate Queueing Device (IMQ), which enhances the packet scheduling ability of Linux to handle more than one network interface [28].

The concept of IMQ is simple. It first creates a Virtual Network Interface in the Linux system with Linux kernel

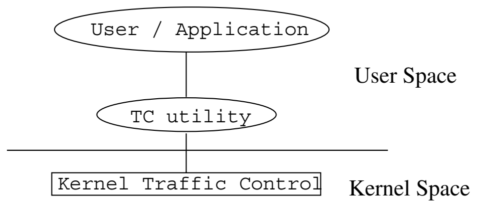


Fig. 2. The structure of Linux Traffic Control.

support. Next, users use the user space tool `iptables` to invoke the packet mangling feature in the Linux kernel to divert packets transmitted from the real network interfaces to the Virtual Network Interface. As a result, packet scheduling on more than one network interface becomes possible by attaching packet scheduling algorithms to the Virtual Network Interface, which is usually named as `imq0` in Linux systems. With IMQ, we can attach different packet scheduling settings[4] to investigate the efficacy of the software-based approach to performing interference avoidance between Bluetooth and IEEE 802.11b.

2.2. HTB + SFQ scheduling

This configuration makes use of the *nesting* feature, allowing more than one Queueing Discipline to be attached to one network interface in a hierarchical tree structure, as described in the Advance Linux Routing HOWTO [22]. We used HTB (Hierarchical Token Bucket) as the first level to divide the traffic depending on the interfaces, i.e., WLAN and Bluetooth. Then, within each queue, an SFQ (Stochastic Fairness Queueing) is attached. As indicated in Fig. 3, the WLAN interface is at class 1:2 while the Bluetooth interface is at class 1:3. According to the HTB property, each class is assigned a bandwidth allocation in the "at least" sense. In our set-up, the class 1:2 (for WLAN) is assigned to have a bandwidth of at least 11 Mbits/sec while the class 1:3 (for Bluetooth) is assigned to have a bandwidth of at least 1 Mbits/s. This also means that the class 1:0 (aggregate of WLAN and Bluetooth) could have a bandwidth of at least 12 Mbits/s.

On the other hand, if the traffic in a class does not make use of all the bandwidth assigned, excess bandwidth from that class is created. In our set-up, any excess bandwidth from class 1:2 or class 1:3 will be shared by them, as they are under the class 1:0. It should be noted that all those 1:*n* figures above are arbitrary and merely act as the Identification Number of a Queue. In the example, the Queue 1:0 is subdivided into two smaller Queues called 1:2 and 1:3, where 1:2 and 1:3 are in the same hierarchy level.

Such setting was decided in the hope that excess bandwidth from classes 1:2 and 1:3 resulting from colocated interference can still be shared among the two interfaces. In other words, when one interface's bandwidth drops due to interference, its excess bandwidth will be transferred

[3] For historical reasons, the Linux literature refers packet schedulers or packet scheduling algorithms as *queueing disciplines*. This is fully evident by the programming structure named `struct Qdisc`, which contains pointers to scheduling routines and necessary variables for a queueing discipline. In this paper, these terms are used interchangeably.

[4] FIFO is the default scheduling setting in Linux.

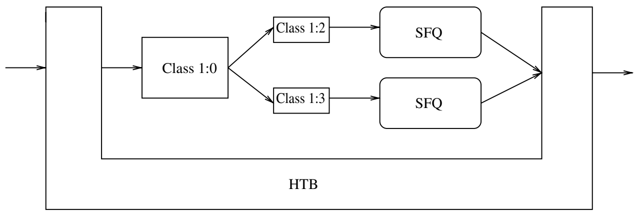


Fig. 3. The HTB + SFQ configuration.

to another interface as an attempt to protect the aggregate bandwidth of the two interfaces.

2.3. PRIO + TBF scheduling

This configuration is well documented in the Linux traffic control literature. It is suggested in [3] due to the widely conceived fact that Bluetooth transmission survives more robustly than the IEEE 802.11b transmission in a colocated environment. Moreover, WLAN raw data rates are much higher than that of Bluetooth. Thus, we use the PRIO (Priority Queue) queueing discipline in Linux to give the WLAN connections a higher priority class. However, in order not to starve the traffic with lower priority (the Bluetooth traffic in this case), we attach a TBF (Token Bucket Flow) scheduling algorithm to each class, as illustrated in Fig. 4.

2.4. SFQ scheduling

In this configuration, we simply use SFQ to schedule the WLAN and Bluetooth traffics at the IMQ interface in a mixed manner. We set ten seconds as the period for the system to change its hash functions to increase the randomness of the algorithm.

Recently, the performance study of existing packet scheduling algorithms in Linux for interference coordination between Bluetooth and WLAN in [41] suggested that a hierarchical-based packet scheduling algorithm which differentiates between Bluetooth and WLAN traffics performs better. In particular, a combination of Priority Queueing (PRIO) and Token Bucket Flow (TBF) gave the best performance results in terms of instantaneous bandwidth. This performance was even better by another scheduling approach which is based on Hierarchical Token Bucket (HTB) by [16].

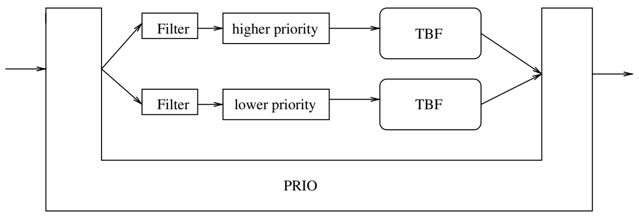


Fig. 4. The PRIO + TBF configuration.

2.5. Scheduling techniques tailored for a wireless environment

Channel State Independent Fair Queueing (CIFQ) is a well-known and theoretically efficient wireless scheduling algorithm considered in our study.

CIFQ uses the Two-State Markov Chain Channel Model as the wireless channel model. This channel model is widely used in academia [7,8,26]. Under this channel model, any wireless channel has only two states – either *good* or *bad*. A *good* channel state represents the perfect channel state in which wireless transmissions can be done with full bandwidth. On the other hand, a *bad* channel state represents the non-perfect channel state, but the model simply assumes that no packets can be transmitted in this state. This channel model simplifies the scheduling problem significantly and hence many techniques in wired packet scheduling can be applied to the situations with the *good* channel state.

On the scheduling mechanism, CIFQ employs the system of Start-Time Fair Queueing [21] as the reference system. The network traffic is considered to be a number of *flows* operating at the same time, sharing the same wireless channel. It uses the data rate of each flow of traffic as the Channel State Information for that flow. The reference system is used to classify flows into *leading* or *lagging*. A *leading flow* is a flow which gains more share of the network resources than the reference system while a *lagging flow* is a flow which obtains less share than the reference system. This *leading* or *lagging* property of each flow is stored in a variable *lag* which counts the number of bytes of a flow which is lagged from the reference system. As the network resources are fixed, we have: $\sum_i lag_i = 0$.

Two versions of implementation exist when CIFQ was first proposed by Ng et al. [30], which were named as *Simple Version* and *Full Version*. Their main difference is that the Simple Version does not include the feature of graceful degradation of leading flows in the Full Version. Also, the Full Version holds additional information of normalized amounts of service received by the leading flows, additional service received by the leading flows and additional service received by the non-lagging flows. In our study, these two versions of CIFQ are implemented in the Linux kernel to measure their efficacy to fight against the interference between colocated Bluetooth and WLAN. The results are shown in Section 5, where we use CIF-Simple and CIF-Complete to represent Simple Version and Full Version, respectively.

3. Proposed algorithms

Based on the empirical findings discussed before, we define the following performance metrics to evaluate packet scheduling algorithms for coordinating colocated Bluetooth and WLAN network traffics:

- *Steadiness*: It measures the degree of downtrend/uptrend of the data rates of both interfaces over time. Here, by data rate, we mean the application level data rate, which does not count the header information bits for the network layer and below.
- *Bandwidth Stability*: It measures the degree of fluctuation of instantaneous data rates experienced by both interfaces.
- *Bandwidth Utilization*: It measures the data rates achieved by both interfaces.

In the subsequent sections, we present our proposed packet scheduling algorithms designed with the goal of achieving no gradual downtrend of data rates, minimizing fluctuation of instantaneous data rates, and maximizing data rates utilized by both interfaces.

3.1. BoTh-WiN

In view of the robust performance of the PRIO + TBF scheduling (from [41]) and the disappointing performance from complex CIFQ, we design an IMQ-based queueing discipline in Linux called BoTh-WiN (BlueTooth and WlaN Both Win). BoTh-WiN provides Bluetooth and WLAN traffics with dynamic priorities between the two interfaces, as shown in Fig. 5. This follows the conclusion of [41] that a hierarchical-based packet scheduling approach is more suitable for interference coordination. Indeed, a proper shift in priorities between Bluetooth and WLAN can minimize the effect of interference and provide steady data rates. The detection of interference is based on instantaneous rate measurement of traffic flows.

Unlike CIFQ and like PRIO, BoTh-WiN considers Bluetooth traffic flows and WLAN traffics separately in the scheduling process. And like most other existing queueing disciplines in Linux, BoTh-WiN uses the software-based virtual interface IMQ [28] to merge the flows from Bluetooth and WLAN together and then schedule at the IMQ interface, thereby performing packet scheduling on two real networking interfaces.

As indicated in Algorithms 1 and 2, BoTh-WiN uses the `jiffies` variable for time duration measurement in order to calculate the instantaneous traffic rate at each interface. The calculation is implemented with the two-element array `struct btwn_flow_data flowtype[2]`, where `flowtype[0]` and `flowtype[1]` refer to the Bluetooth and WLAN interfaces, respectively. The `struct btwn_flow_data` is a data type used to hold the data for each flow of traffic. It provides the scalability to expand the rate measurement precision up to per flow level, as in the case of the CIFQ-based algorithms. But in the NAP scenario considered in this paper, we believe that finding the instantaneous bandwidth at the interface level is enough for BoTh-WiN to have channel state considerations and to be more computation-efficient than the CIFQ-based algorithms where *per flow* level of rate measurement is found.

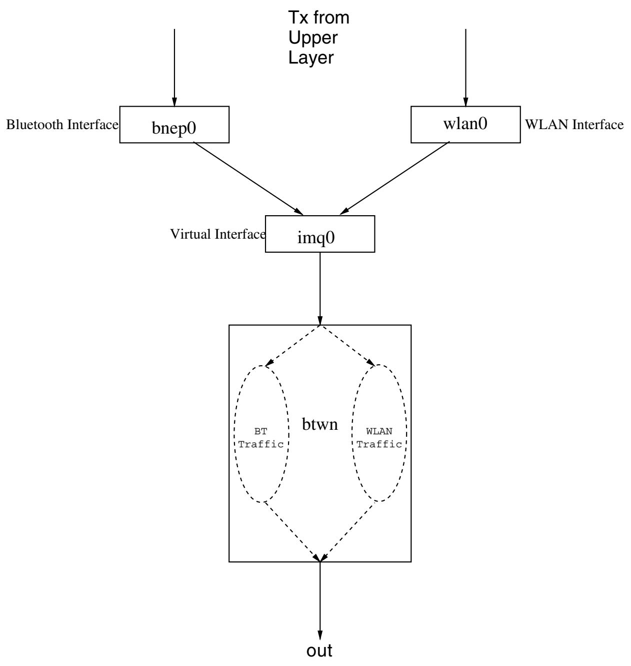


Fig. 5. The design of BoTh-WiN.

Algorithm 1 BoTh-WiN: Status Checking

```
1:  INPUT: $skb$ (the packet to enqueue);
2:  OUTPUT: Status of Enqueue Operation;
3:  Procedure BTWN-ENQUEUE(skb)
4:  $i$ = select-flow((src-ip, dst-ip, src-port,
    dst-port, interface) in $FLOWS$);
5:  $now$ = jiffies;
6:  if $i \notin FLOWS$ then
7:      $FLOWS = FLOWS \cup i$;
8:      $vt_i = now$;
9:  end if
10: $status$ = skb-enqueue(skb, flow[$i$]);
11: check-flow-types(); /* Video, Voice, or others
    */ /* Video and Voice flows are given a lower
    virtual time $vt$ */
12: $FLOWTYPE[\text{interface}]_{\text{lastdeq}} = now$;
13: return $status$;
```

Algorithm 2 BoTh-WiN: Packet Sending

```
1:  INPUT: InterferenceFlag[this interface];
2:  OUTPUT: $skb$ (packet to send); OR NULL (if not
    to send);
3:  Procedure BTWN-DEQUEUE()
4:  if $BluetoothInterference > 3$ OR
    $WLANInterference > 3$ then
5:      return NULL;
6:  end if
7:  Read among the queues for Bluetooth,
8:  $j$[BT] = Pick the flow with lowest $vt$;
9:  Read among the queues for WLAN,
10:    j[WN] = Pick the flow with lowest $vt$;
11:    if j[BT] is valid and j[WN] is invalid then
12:        $skb$ = skb-dequeue($FLOWS[j[BT]]$);
13:        $vt_{FLOWS[j[BT]]} += skb \rightarrow len / 100$;
14:    else if j[BT] is invalid and j[WN] is valid then
15:        skb = skb-dequeue($FLOWS[j[WN]]$);
16:        $vt_{FLOWS[j[WN]]} += skb \rightarrow len / 100$;
17:    else if j[BT] is valid and j[WN] is valid then
18:        Consider the following THREE conditions:
19:        (WLAN.rate / Bluetooth.rate $\geqslant 10$); /* 1 */
20:        (WLAN.lastdeq > Bluetooth.lastdeq); /* 2 */
21:        (WLANInterference > BluetoothInterference); /* 3 */
22:        if the two or more of the above conditions are true then
23:            $skb$ = skb-dequeue($FLOWS[j[BT]]$);
24:            $vt_{FLOWS[j[BT]]} += skb \rightarrow len / 100$;
25:        else
26:            $skb$ = skb-dequeue($FLOWS[j[WN]]$);
27:            $vt_{FLOWS[j[WN]]} += skb \rightarrow len / 100$;
28:        end if
29:    end if
30:    rate-update(BluetoothInterface);
31:    rate-update(WLANInterface);
32:    if Bluetooth.rate < Bluetooth.rate$_{ma}$ then
33:        AtomicInc(BluetoothInterference);
34:    else
35:        AtomicSet(BluetoothInterference, 0);
36:    end if
37:    if WLAN.rate < WLAN.rate$_{ma}$ then
38:        AtomicInc(WLANInterference);
39:    else
40:        AtomicSet(WLANInterference, 0);
41:    end if
```

On using the rate measurement technique, we face an important difficulty in that we cannot determine which bandwidth level of the interfaces, either for Bluetooth or WLAN, is free from interference, or vice versa, in absolute terms. It is because the factors of the surrounding environment like distance, wireless channel fading, the presence of some microwave absorbing substances etc., may change the wireless channel conditions and hence, the instantaneous bandwidth of each interface.

However, in our results of FTP tests in Section 5.1, it is always the case that the start of interference simply gives the existing wireless transmission a persistent drop in bandwidth instantly. In particular, the case of FIFO scheduling creates gradual deterioration of bandwidth in both interfaces, even after the drop in bandwidth right at the start of the interference. Thus, this motivates one of our design requirement for the queueing disciplines in handling interference between the two interfaces, which is the maintenance of bandwidth stability under the interference environment over time.

The key observation is that only an instant big drop in bandwidth does not imply interference. Thus, we propose using the *moving average* (MA) concept as the key for BoTh-WiN to detect interference by instantaneous rate measurement. Fig. 6 illustrates this MA concept. In general, we may classify the bandwidth fluctuation of any wireless channel into three cases, namely steady, decreasing and increasing. When the bandwidth is steady, it is still fluctuating horizontally, as shown in Fig. 6(a). The MA curve is a smoothed version of the fluctuating bandwidth and it becomes more or less like a horizontal line in the case of a steady bandwidth.

In order not to falsely detect an interference, we design BoTh-WiN to generate an interference detection if *three consecutive* samples of the instantaneous rate is lower than the MA, indicating a persistent drop in bandwidth. This situation is shown in Fig. 6(b). On the other hand, if the instantaneous rate is increasing, as caused by the start of a transmission or disappearance of interference, the MA curve will follow the rise but at a slower pace. Here, we do not have any interference, as seen from Fig. 6(c). When an interference due to coexistence is confirmed, BoTh-WiN performs a *no dequeue* for one packet time at the IMQ interface. This in turn gives no transmission for both Bluetooth and WLAN at the same time, avoiding interference.

To provide balanced treatment towards Bluetooth and WLAN, BoTh-WiN separates the Bluetooth and WLAN traffics when performing packet scheduling. At the `dequeue` function, BoTh-WiN first examines which interface is having no packets to send at that instant (i.e., no backlogged flows at that interface). Then BoTh-WiN will concentrate on the (other) interface with backlogged flows. It will select the flow with the smallest virtual time value and it will advance the virtual time value vt as follows: `vt += skb->len/100`. This formula will be further explained in Section 3.2.

Because of the large discrepancy between the data rates of Bluetooth and WLAN, the above situation happens quite often and BoTh-WiN will serve the WLAN side. Yet, there still exists tricky cases in which both Bluetooth and WLAN interfaces contain backlogged flows. In this case, BoTh-WiN tests the following three criteria to select which interface to get the service:

- *Bandwidth Balancing*: WLAN Bandwidth/Bluetooth Bandwidth $\geqslant 10$?
- *Round Robin Balancing*: The WLAN interface got service last time?
- *Interference Consideration*: The WLAN Interface will be sooner to detect an interference?

The above three criteria are specially designed in that an affirmative answer to any one of them means that the system needs to put more attention to the Bluetooth interface. On the Bandwidth Balancing test, our experimental results show that the bandwidth for Bluetooth

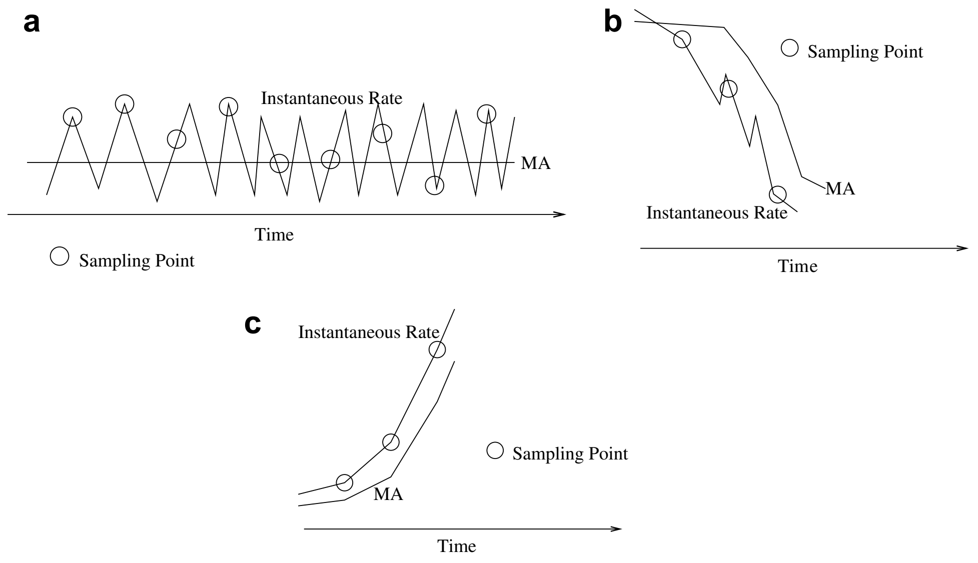


Fig. 6. BoTh-WiN moving average cases. (a) Case 1: steady, (b) Case 2: decreasing, (c) Case 3: increasing.

without interference is about 70 kbytes/s while that for WLAN is about 700 kbytes/s. Thus, we set the ratio number 10 for the Bandwidth Balancing test. If this ratio is maintained over the long term, both interfaces are subjected to the same level of punishment due to interference in terms of bandwidth reduction percentage. The Round Robin Balancing condition is to enable BoTh-WiN to have some TBF effect on the WLAN traffic as in the PRIO + TBF queueing discipline setting. Finally, the Interference Consideration is actually a comparison of the channel states experienced by the Bluetooth and WLAN interfaces.

Using the aforementioned interference detection mechanism, BoTh-WiN keeps track of the number of incidence of the instantaneous rate lower than the moving average of the rate consecutively. The higher the number of such incidence, the sooner will the interface detect an interference and the poorer the channel quality has the interface been experiencing. On the other hand, BoTh-WiN, grants the interface with better channel state a higher priority. To avoid errors in making decisions based on the three criteria, BoTh-WiN chooses the Bluetooth interface to get service when either two true values are obtained from the three tests above. In this manner, dynamic priorities are assigned to Bluetooth and WLAN interfaces to provide balanced services between them.

Lastly, it should be noted that using rate measurement to estimate the Channel State Information (CSI) encounters a system limitation so that this rate measurement cannot be done too frequently. Otherwise, too many threads will access the system time variable `jiffies` at the same time, potentially causing `ksoftirq` system crash error. Lines 30 and 31 are for the *not so frequent* sampling technique employed to avoid this problem.

3.2. Channel adaptive WLAN (CAWN)

We also propose Channel Adaptive WLAN (CAWN) to realize a better cooperation between two network interfaces (i.e., Bluetooth and WLAN in this paper) without the use of Intermediate Queueing Device (IMQ), thereby removing the delay due to IMQ and increasing the system performance. The cooperation is realized based on our novel software architecture design which allows communication between queueing disciplines in different network interfaces inside the Linux kernel.

Besides the cooperation property achieved, CAWN also integrates seamlessly with the wireless extensions by Tourrilhes [38]. Fig. 7 shows the design of CAWN. We can see that CAWN extracts the CSI of the WLAN transmission from the WLAN device drivers via the wireless extensions API. Using the CSI extracted, CAWN determines whether interference has occurred. If interference is detected, CAWN at the WLAN interface will acknowledge the CAWN counterpart at the Bluetooth interface. Both interfaces will then choose not to dequeue any packet unless the packet size is small (e.g., Voice packets). This `no dequeue` operation is done once and the CAWN devices attached to both interfaces will acknowledge each other about *interference handled* and resume normal operations. At first glance, it seems strange that the `no dequeue` operation is done only once. However, practical results show that further increase in the number of such operation simply induces unnecessary bandwidth reduction in both interfaces.

Algorithm 3 CAWN

```
1:  INPUT: $skb$ (the packet to enqueue);
2:  OUTPUT: Status of Enqueue Operation;
3:  Procedure CAWN-ENQUEUE(skb)
4:  $i$ = select-flow((src-ip, dst-ip, src-port, dst-port) in
    $FLOWS$);
5:  $now$ = jiffies;
6:  if $i \notin FLOWS$ then
7:     $FLOWS = FLOWS \cup i$;
8:     $vt_i = now$;
9:  end if
10: $status$ = skb-enqueue(skb, flow[$i$]);
11: check-flow-types(); /* Video, Voice, or others */
    /* Video and Voice flows
    are given a lower virtual time $vt$ */
12: if interface is WLAN then
13:    $success$ = IW_SPY-Operation-Read-Channel-State();
14:    if $success$ then
15:       update-channel-state-statistics();
16:    end if
17:    if Bluetooth Interface exists then
18:       if CAWN attached to the Bluetooth Interface then
19:          $FLAG$ = AtomicRead(InterferenceFlag
             [this interface]);
20:          $BluetoothFlag$ = AtomicRead
             (InterferenceFlag[Bluetooth]);
21:          if check-interference() and !FLAG then
22:             AtomicSet(InterferenceFlag[this interface], ON);
23:             /* Alert Bluetooth Interface */
24:             AtomicSet(InterferenceFlag[Bluetooth] ON);
25:          else if $FLAG$ and !$BluetoothFlag$ then
26:             AtomicSet(InterferenceFlag[this interface],
                OFF); /* Interference Handled */
27:          end if
28:       end if
29:    end if
30: end if
31: return $status$;
```

Algorithm 4 CAWN_DEQUEUE

```
1:
2:  INPUT: InterferenceFlag[this interface];
3:  OUTPUT: $skb$ (packet to send); OR NULL
    (if not to send);
4:  Procedure CAWN-DEQUEUE()
5:  $FLAG$ = AtomicRead(InterferenceFlag[this interface])
6:  $j = \min_{vt_k} \{k \in FLOWS — !EmptyQueue(k)\}$
7:  $skb$ = skb-dequeue($j$);
8:  if $FLAG$ then
9:     if $skb \rightarrow len \geqslant 500$ then
10:       AtomicSet(Interference[this interface], OFF);
          /* Interference Handled */
11:       skb-queue-head($j$, $skb$);
12:       return NULL;
13:    end if
14: else
15:       $vt_j += skb \rightarrow len/100$;
16: end if
    STATE return $skb$;
```

As shown in Algorithms 3 and 4, CAWN first extracts the CSI at the WLAN interface using one of the `IW_SPY` operations of the wireless extensions API [38], instead of the regular routine

`sch->dev->get_iw_quality()` as defined in `$KernelSrc/include/linux/netdevice.h:get_iw_quality()`.

Though both the `IW_SPY` operation and the `get_iw_quality()` routine can serve the purpose of getting `struct iw_quality` structures for extraction of channel state information, the HostAP driver by [27] does not support the `get_iw_quality()` routine in an atomic/interrupt context. On the contrary, the `IW_SPY` operation is supported in the interrupt context in which the queueing discipline runs. What is more encouraging is that it can give as many as `IW_MAX_SPY` `struct iw_quality`'s from `IW_MAX_SPY` clients. Currently, `IW_MAX_SPY` is defined as eight in `$KernelSrc/include/linux/wireless.h`. Therefore, this method can handle as many as eight pieces of CSI from eight WLAN clients.

On invoking the `IW_SPY-Operation-Read-Channel-State()` function at line 13 of Algorithm 3, it simply calls the supporting routine at the WLAN device driver which copies the `iw_quality` and `sockaddr` structures to the memory space starting from `q->buffer`. As indicated in Fig. 8, `q->buffer` is the place to which a queueing discipline `q` (i.e., CAWN) copies the CSI of WLAN clients. The `sockaddr` structures are used to hold the MAC addresses of the WLAN clients. Fig. 8 shows how these structures are aligned after the successful completion of the `IW_SPY` operation. The corresponding pair of `iw_quality` and `sockaddr` holds the CSI of the WLAN client indicated by the `sockaddr`.

The lines from 19 to 27 in Algorithm 3 are for cooperation between two queueing disciplines attached to Bluetooth and WLAN. Since we have the mechanism to extract the CSI at the WLAN interface via the wireless extensions API, the cooperation realized is WLAN driven. For system flexibility, we cannot assume the existence of the Bluetooth interface during the initialization of the queueing discipline attached to the WLAN interface. Thus, we have to check the existence of any Bluetooth interface during run time. Furthermore, we cannot be sure whether the queueing discipline attached to the Bluetooth interface (which exists under the name `bnep0` in this case) is also CAWN, we have to check this too. The lines 19–27 are

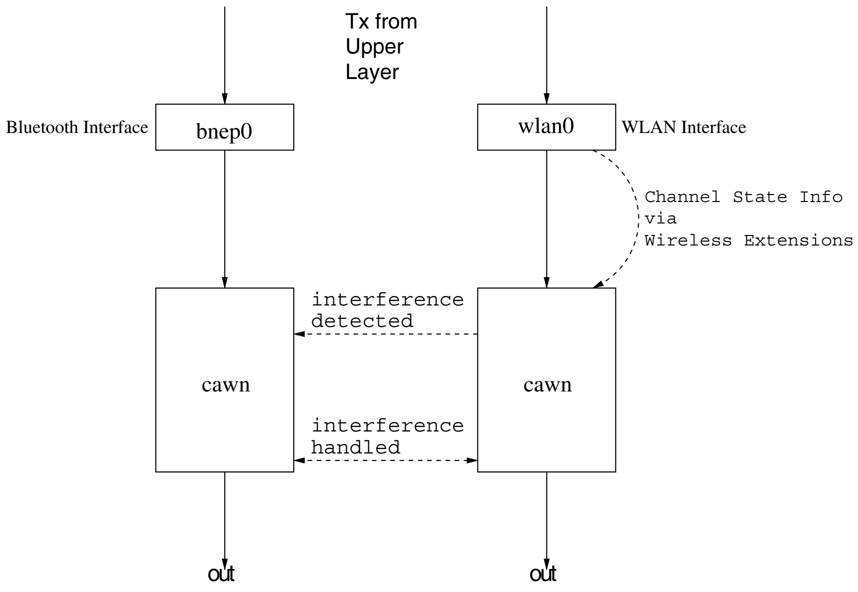


Fig. 7. The CAWN design.

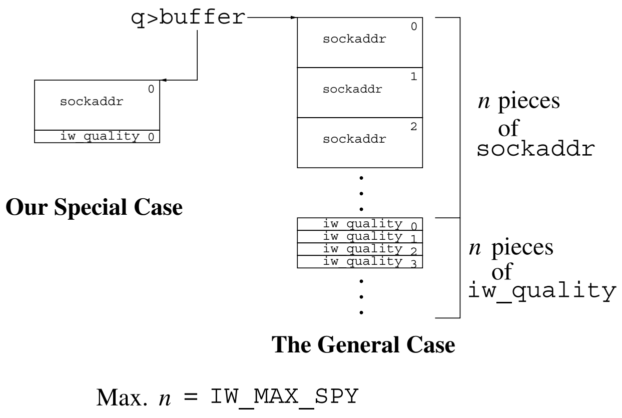


Fig. 8. Channel state extraction: IW_SPY.

to be run at the CAWN attached to the WLAN interface. Actually, there are also similar codes for the CAWN to run at the Bluetooth interface, but the main functionality is just to acknowledge the CAWN attached to the WLAN interface about *interference handled*. This is to allow both interfaces to resume normal operations at the same time.

The structure `struct iw_quality` has one important member variable called `updated`. It is set by the WLAN device driver to indicate whether the values regarding the CSI is updated by the hardware. This is critical because of the dynamic nature of the channel state of WLAN transmissions and hence the high volatility of the memory holding such information in the WLAN device driver. Also, such a flag would also indicate the reliability of the values read.

On updating the channel state statistics at line 15 of Algorithm 3, the queueing discipline will only consider the updated values of `struct iw_quality` as the CSI for further processing. If the queueing discipline fails to get the *updated* `struct iw_quality`, it will simply continue its normal operations, rather than looping there to wait for the *updated* `struct iw_quality`. Such a design has taken into account not only the interrupt context in which the queueing discipline is working, but also multiple instances of the same queueing discipline running in the system simultaneously but in different phases, in the multi-thread/multi-processing environment under Linux. Fig. 9 below shows this concept graphically. For the sake of simplicity, only three instances of the same queueing discipline are shown. Although instances B and C might have missed the chance to read the *updated* `struct iw_quality`, instance A still have the chance since it has just started the execution among all the three instances (as indicated by the arrows).

CAWN uses the following code to perform the update of virtual time on the selected flow in a simple manner:

```
vt += skb->len/100;
```

`vt` is a `u32` type variable taking four bytes and its initial value is dependent on `jiffies`, which is updated by the kernel timer interrupt every 10 ms and counts from zero since the system startup. `skb->len` is a value no bigger than 1500 since 1500 bytes is the size of the Maximum Transfer Unit (MTU) of all the Ethernet-like interface in Linux. With the maximum value of `vt` being $2^{32} - 1$, and the maximum length of the packet dequeued being 1500 bytes, we have the following:

The life time of valid `vt` before overflow = $(2^{32} - 1)/15/100 \geqslant 2863311\text{s} \geqslant 30$ days.

This life time is much longer than enough when compared to the life time of a network traffic flow. Unless for critical and rare applications, it is hardly required for any traffic flow to last over 3 days.

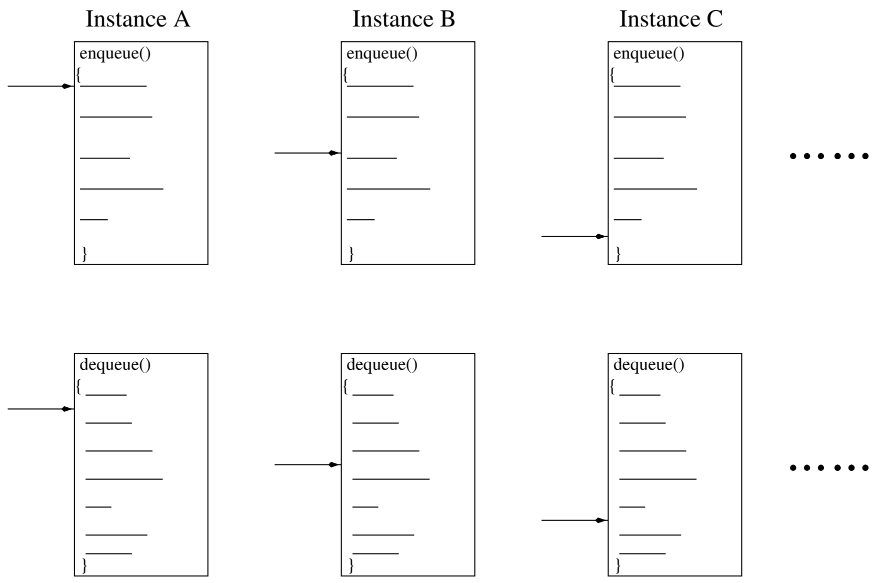


Fig. 9. Multiple queueing discipline instances.

4. Experimental environment

In this section, we first describe the experimental setup of the Linux system and hardware interfaces we used in our study. We then describe the testing traffic configurations in all the experiments.

4.1. Setup

The design of the Linux traffic control architecture contains ready-made classes and filters for IP traffic. In our study, we use the environment as shown in Fig. 10. Physically, all the three machines shown in the figure form an equilateral triangle with sides of one meter in length. Throughout all the tests in this paper, all these three devices are *stationary* for simplicity since our focus is on interference but not location-dependent errors. In the scalability tests, we use variable number of client devices.

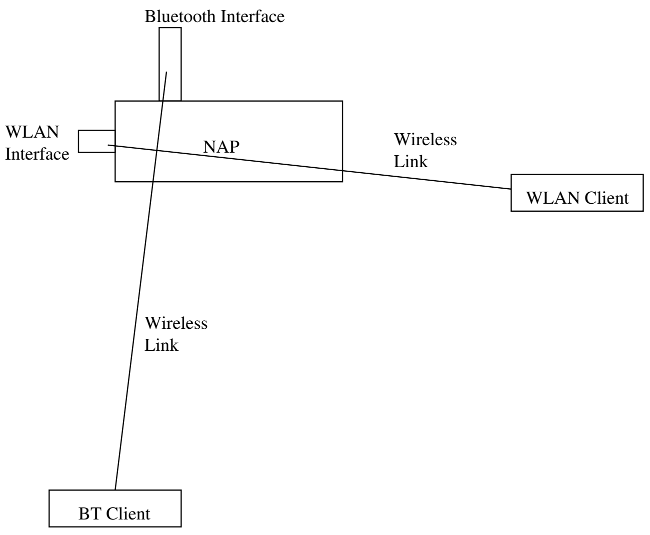


Fig. 10. The testing environment.

All the wireless links are brought up to the IP level. The two clients each downloads a 600 MB file from the NAP (i.e., the Linux machine) via the FTP protocol. For the NAP (or FTP server), the outflow transmission was preprocessed by some QoS techniques in the Linux kernel. As mentioned above, the queueing disciplines (`qdisc`) considered are: HTB, PRIO, SFQ, and TBF. All the Linux traffic control settings done in the tests are for *egress* traffic only – we only scheduled the out-bound traffics of the NAP (e.g., the FTP download traffics of the clients). Nevertheless, the cases where there are uplink traffic flows are also considered. The physical layer parameters are shown in Table 1.

For the WLAN driver, we used the HostAP driver [27]. This driver enables an easy configuration of the AP (access point) mode of the IEEE 802.11b interface. Indeed, in all the tests done, the WLAN interface of the NAP is set in AP mode, which uses the point coordination function (PCF) as the MAC layer scheme. Moreover, the WLAN transceivers used are the Linksys WPC11 PCMCIA cards.

For the Bluetooth interface, we used the Billionton USB Bluetooth Adapter as the transceiver for both the NAP and

Table 1
Physical layer parameters

Parameter	Setting
diameter of coverage	2 m
path loss and channel model	AWGN
Bluetooth system loading	100%
Bluetooth packet types	DH5
Bluetooth transmitted power	1 mW
IEEE 802.11b raw data rate	11 Mbps
IEEE 802.11b transmitted power	25 mW

the client. BlueZ [11] driver is the software driver of the devices. The Bluetooth connection is brought up to the IP level with the PAN (Personal Area Network) profile [12] of Bluetooth, in which Bluetooth Network Encapsulation Protocol (BNEP) and Bluetooth logical link control and adaptation protocol (L2CAP) [12] are employed. As the NAP is the master and the client is the slave, we deliberately set DH5 as packet type for network transmission at the NAP because DH5 gives the fastest data rate for the Bluetooth as the baseband layer at 723 kbits/s, which is about 90.375 kbytes/s.

4.2. Wireless transmission scenarios

In a broad sense, we study two transmissions scenarios. One is using single flow at each interface to measure the interference effects on the bandwidth of each interface. The other is using multiple flows at each interface to carry out a performance study of packet scheduling algorithms for different applications under interference.

4.2.1. Single flow scenario

The single flow scenario uses simultaneous FTP downloads of a 600 MB file from the NAP to the clients (see Fig. 10). All the three machines used in our experiments are Linux PCs. The FTP server (at the NAP) used is the WU-2.6.1-16 bundled with Redhat Linux 7.1 and the FTP client software is the NCFTP client. We use the modified `xnetload` [36] program to measure the bandwidth in bytes per second of the Bluetooth and WLAN interfaces at the NAP (only the out-bound traffic). The sampling period is around 700–1000 s in all tests.

We assume that the network transmissions are performed in the *best effort* manner. Thus, the trend of the peak bandwidth could indicate the channel condition. In our study, we assume that the channel condition varies according to the interference between WLAN and Bluetooth. It follows that a higher bandwidth obtained at an interface indicates a better channel condition for that interface and vice versa. We also study the range of fluctuation of bandwidth, or the *bandwidth stability* over time.

4.2.2. Multiple flow tests

The multiple-flow scenario is done with the help of the Distributed Internet Traffic Generator (DITG) [5]. DITG is a powerful thread-based network traffic performance measurement tool allowing multiple flows generation at the same time. It allows the user to specify a wide range of parameters including number of packets in one second, packet size, packet size distribution, transmission duration, transport protocol used (TCP/ UDP), etc. In particular, it can generate G.711-based Voice-over-IP (VoIP) flow, which is widely deployed in Internet Phone applications. With DITG, we generate three types of network traffic flows: Video, Voice and TCP. Table 2 shows the parameters used. We also perform the tests under three levels of Bluetooth traffic loading.

Table 2
Parameters for traffic flow generation for DITG

Flow type	Transport protocol	No. of packets/s	Packet size (bytes)
Video-BT	UDP	10	1500
Video-WLAN	UDP	45	1500
Voice	UDP	–	120
FTP	TCP	100	1500

5. Experimental results

In this section, we present the experimental results we obtained and our interpretations on these results. We first describe the results for FTP tests, in which the single-flow downlink performance is investigated. We then describe the results for the DITG tests, in which the multiple-flow performance is examined. To investigate the performance of the system in the presence of both uplink and downlink traffic flows, we also performed experiments in which there are multiple near-by IEEE 802.11b interference sources. Our final set of results aims at investigating the scalability of the NAP in that we tested it with more than two clients. More results can be found in [41].

5.1. FTP tests

Fig. 11 shows that BoTh-WiN could provide coexistence between Bluetooth and WLAN with a steady bandwidth. The Bluetooth traffic could gain back to its steady state within three minutes after the start of the WLAN traffic. Since then, no major fluctuation is seen from both the data rates of Bluetooth and WLAN in the tests. Though we cannot see any uptrend/downtrend of bandwidth, the data rates of both WLAN and Bluetooth could stay at their respective high levels throughout most of the time of interference. Thus, for systems which are unable to obtain the channel state information easily, BoTh-WiN is preferable as it is based on instantaneous traffic rate calculation. Though the lost Bluetooth bandwidth cannot be totally re-gained, this scheduling algorithm can stop any further deterioration of interference effects such as excess volatility and gradual deterioration of data rates by dynamically assigning priorities between Bluetooth traffic and WLAN traffic at appropriate times.

Fig. 12 shows the results of CAWN. Fig. 12(a) shows the results when the Bluetooth interface starts transmissions first. Though it takes longer for the Bluetooth to gain back its steady state after the start of the WLAN interference, the steady state for Bluetooth bandwidth under interference is above 40 kbytes/s. At the beginning, we can see persistent bandwidth drop for the Bluetooth since the CAWN at the WLAN is working, issuing the “no dequeue for one packet time” request quite frequently when the WLAN starts its transmission. Yet, after reaching a steady state, both are steady at high data rates seen under interference.

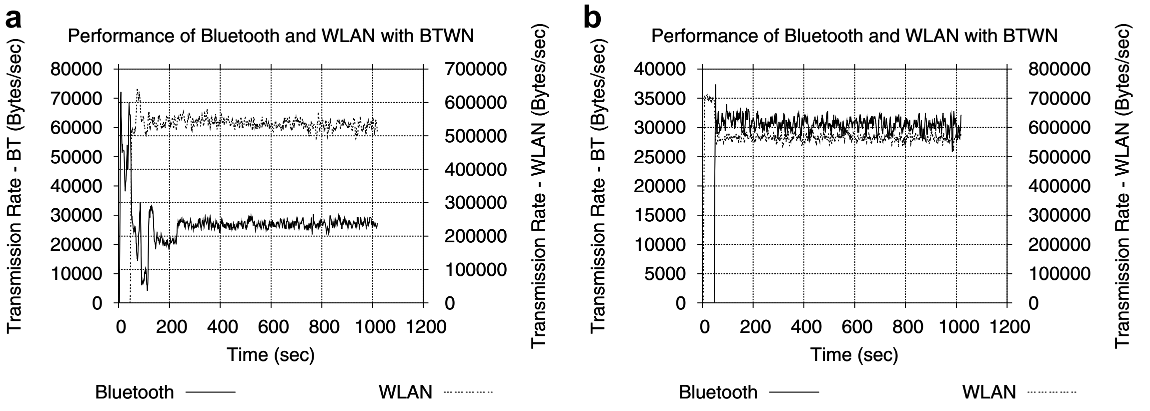


Fig. 11. Performance of BoTh-WiN. (a) Bluetooth first, (b) WLAN first.

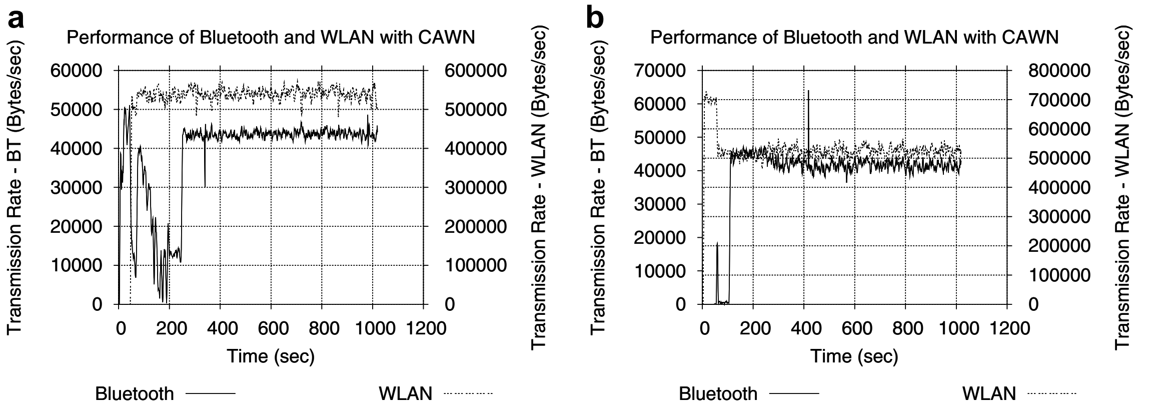


Fig. 12. Performance of CAWN. (a) Bluetooth first, (b) WLAN first.

The bandwidth for WLAN in this case is also as high as above 500 kbytes/s.

With reference to Fig. 12(b), we can see the performance results of CAWN when the WLAN transmission starts first. When we initiate the FTP download at the Bluetooth client side 45 s after the start of the WLAN FTP download, the Bluetooth client experiences a time period of about one minute delay after exchanging FTP requests, acknowledgements, FTP login information, etc. These exchanges can be seen from the short term spike below 20 kbytes/s mark at the beginning of the Bluetooth transmission. This is resulted from the cooperative *no dequeue* policy implemented by CAWN whenever it detects a sharp, significant and fast deterioration of WLAN channel quality obtained from the `IW_SPY` operation. This hinders the Bluetooth from gaining a high bandwidth at the very beginning. After some time, with the Bluetooth still communicating at a very low bandwidth (several hundred bytes/s), the WLAN channel quality stabilizes and CAWN detects the "interference" signal much less frequently as before.

With this observation, we may argue that the Bluetooth interference effects on WLAN do not depend very much on the real Bluetooth bandwidth attained. With the *no dequeue* policy fading out, Bluetooth transmission shows its robustness under WLAN interference by gaining back its bandwidth as high as above the 40 kbytes/s mark almost instantly at 112 s. Since then both interfaces show impressive bandwidth stability at high bandwidth levels till the end of the test. On the whole, the WLAN bandwidth does not drop below the 500 kbytes/s mark throughout the test.

In our study, we also implemented the Channel State Independent Fair Queueing (CIFQ) in Linux to test its empirical ability in interference coordination. Its performance results and analysis in the FTP tests are shown below.

As we can see from Fig. 13(a), CIF-Simple allows the WLAN bandwidth to reach 550 kbytes/s and stay well above the 500 kbytes/s mark under Bluetooth interference. At the same time, the existing Bluetooth bandwidth only drops from 70 kbytes/s to about 50 kbytes/s. Meanwhile, CIF-Simple fails to enable the Bluetooth bandwidth stability over long time, which is evident by the persistent drop in Bluetooth rate after 700 s.

Fig. 13(b) gives us the most special results in this bandwidth based test. First of all, when the Bluetooth client starts its FTP transfer 45 sec after the WLAN transmission starts, the FTP transfer to the Bluetooth cannot start at all. But at the same time, we found that the WLAN bandwidth

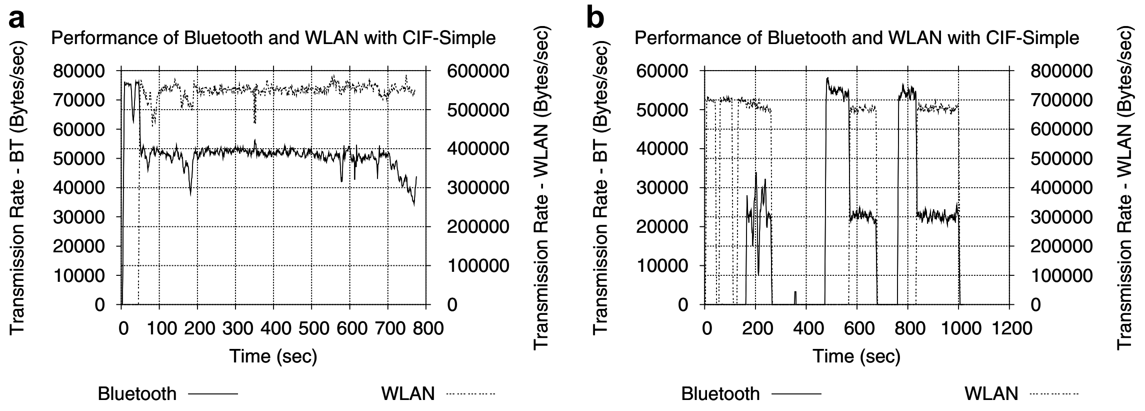


Fig. 13. Performance of CIF-Simple. (a) Bluetooth first, (b) WLAN first.

drops to zero due to interference. Though the WLAN bandwidth can rise back to the 660 kbytes/s level as if there was no interference, it drops to zero again as the Bluetooth client initiates the FTP request again by sending some packets of several hundred bytes. Finally, the Bluetooth client can start its FTP transfer after 3 min of the start of the WLAN transmission and the Bluetooth bandwidth attained is more than 20 kbytes/s. At 266 s of the figure, both the Bluetooth and WLAN data rates drop to zero amid interference and CIF-Simple's inability to schedule the two sides' traffics with huge bandwidth difference. Later in the test we can see that CIF-Simple merely adopts the *either or* approach in between the Bluetooth and WLAN traffics. When the Bluetooth bandwidth is as high as 70 kbytes/s from 500 to 600 s in the graph, the WLAN bandwidth drops to zero at the same time.

CIF-Simple is considered to be *pro-WLAN* at the beginning because the starting-first WLAN traffic has a lower virtual time value. In addition, it is unable to allow Bluetooth to gain its network share because of its low bandwidth and hence CIF-Simple does not think that the Bluetooth is *lagging* very much at all and hence simply ignores it. After a period of time (e.g., 3 min in this case), CIF-Simple begins to feel the *lagging* of Bluetooth traffic and hence starts to allow the Bluetooth to transmit. When the Bluetooth interface gains its share, the WLAN interface loses its share but it takes some time to drop its bandwidth to zero due to its high bandwidth. CIF-Simple chooses the flow to dequeue by finding the lowest value of lag_i/r_i, where lag_i depends on the length of packets sent while r_i is the rate of the flow i. Thus, a flow with high bandwidth can gain its network share more easily. After the WLAN bandwidth drops to zero, it again takes some time for the *lagging* effect to be felt by the CIF-Simple system and hence for the WLAN to gain back its network share. When one interface gains back its network share, the other must suffer loss of network share. Also, the lack of graceful degradation feature of CIF-Simple further worsens the situation. Thus, we notice the *either or* effects between the two interfaces under CIF-Simple and this reflects perfectly the Two-State Markov Chain Channel Model assumed in the mind of CIFQ algorithms. Consequently, CIF-Simple cannot be a solution to Bluetooth and WLAN coexistence.

Fig. 14 shows the results of CIF-Complete. The results in Fig. 14(a) show that CIF-Complete could allow the WLAN bandwidth to be as high as 660 kbytes/s under existing Bluetooth interference but sacrifice the Bluetooth bandwidth to drop to as low as 10 kbytes/s in 200 s.

The complex calculation involved in CIF-Complete fails to properly handle the coexistence of Bluetooth and WLAN. The Bluetooth bandwidth could only stay below 20 kbytes/s during most of the time of interference despite that Bluetooth should have been given a higher priority to dequeue due to its smaller virtual time value (it started before the WLAN traffic). Together with the results of PRIO and CIF-Simple, we may conclude that giving Bluetooth a higher priority to dequeue would potentially create more undesired interference.

On the other hand, the WLAN traffic could be steady at about 660 kbytes/s with such a slow Bluetooth traffic around, which does not create significant interference effects for the WLAN. (See Fig. 14(a).)

When the Bluetooth client starts its FTP download under WLAN interference, unlike the case for CIF-Simple, the Bluetooth bandwidth can reach as high as 40 kbytes/s within 1 min and we do not see any delay of the start of the FTP transfer. One reason for this is that the CIF-Complete algorithm chooses among the two interfaces mainly by the virtual time and this virtual time advances by the length of the packet sent l. But CIF-Simple advances the value of the virtual time with l/r_i. This causes a big difference with the huge discrepancy between the Bluetooth and WLAN data rates. Moreover, the feature of graceful degradation of *leading* flows in CIF-Complete helps the situation a bit. Fig. 14(b) shows that the Bluetooth bandwidth decreases its volatility over time while the WLAN can maintain its bandwidth stability most of the time. Together with the results of the CIF-Simple, virtual time advancement without involving the rate of the flow i provides a more balanced priority between Bluetooth flows and WLAN flows in packet scheduling. That is also the reason why the virtual time advancement technique in

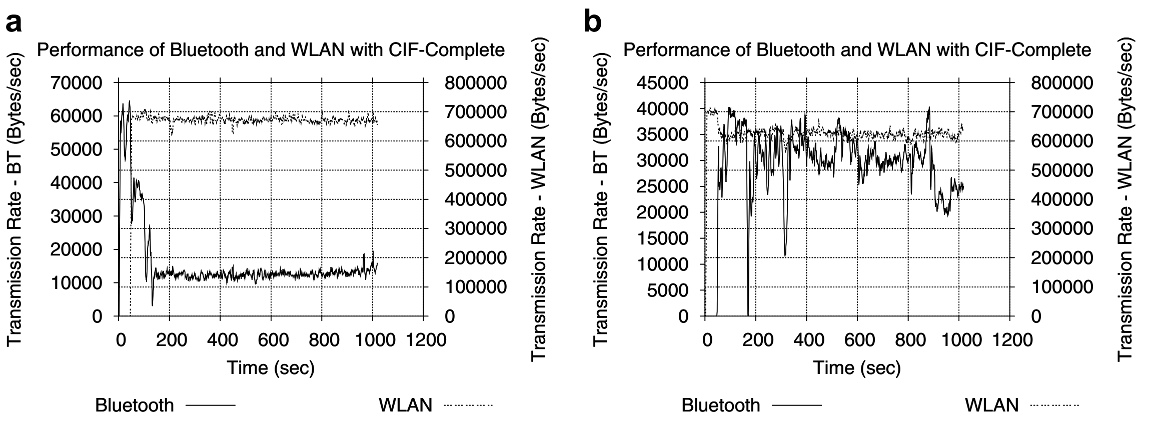


Fig. 14. Performance of CIF-Complete. (a) Bluetooth first, (b) WLAN first.

our proposed schemes, BoTh-WiN and CAWN, is simply dependent on the length of the packet sent.

5.2. DITG tests – voice

This section presents the performance of our proposed schemes, CIFQ algorithms and FIFO in multiple-flow environments with simultaneous Bluetooth and WLAN traffics at the NAP. In all the tests described in this section, we generate three types of flows: Video, Voice and FTP. Table 2 shows the parameters used. We also perform the tests under three levels of Bluetooth traffic loading.

Fig. 15 shows the results of packet drop rate/percentage when we vary the number of Voice flows at the WLAN side. The details of this flow composition is shown in Table 3. In particular, the configuration of 1 Voice flow, 1 Video flow and 1 FTP flow is known as the *basis case* in this paper.

Because of the lower drop percentage for Voice flows and the smaller packet size of VoIP packets, VoIP traffic is more resistant to interference due to its packet size. On the other hand, it is worthy noting that the drop rate with CAWN is low on both Bluetooth and WLAN traffics in this case. In particular, the drop rate for WLAN with CIF-Complete, our proposed BoTh-WiN and CAWN is zero in this case. On the other extreme, FIFO scheme shows a 99% + drop rate when the number of Voice flows in WLAN varies from one to three. CIF-Simple also demonstrates a low drop rate, with the highest drop rate of 3.48% occurring when the number of Voice flows is three.

Nonetheless, the attractive low drop rate at the WLAN side comes from sacrificing the Bluetooth traffic among all queueing disciples tested, except CAWN. The famous CIFQ algorithms suffer from having high drop rates of above 90%, losing completely in this case. The simple FIFO scheme could also yield a “low” drop rate of about 50%. (In our study, virtually all drop rates measured are close to the high end.) On the contrary, our proposed BoTh-WiN scheme yields a lower drop rate than the CIFQ algorithms most of the time, achieving a lowest drop rate of as low as 20%.

The above results show that detecting the channel state of WLAN with the wireless extensions API is a good approach to avoiding interference under a Light Load Bluetooth environment. (See Figs. 15(a) and (b).)

The drop percentage results under Medium Load Bluetooth environment are shown in Figs. 15(c) and (d). Like the case in Light Load Bluetooth environment, the WLAN drop rates are low while those of Bluetooth are high. Our proposed BoTh-WiN and CAWN still give lower Bluetooth drop rates among the queueing disciplines tested. Also, from the Bluetooth drop rate curve of CAWN in Medium Load Bluetooth environment, increasing the number of WLAN flows can decrease the Bluetooth drop rate, as shown in Fig. 15(c). This is because of the increased sampling frequency of the WLAN channel state and hence better interference detection and avoidance can be done.

While FIFO achieves zero WLAN drop rate in the basis case, its drop rate increases significantly to 90% when the number of Voice flows in WLAN increases. (See Figs. 15(e) and (f).)

5.3. DITG tests – video

In this section, we present the DITG test results of video flows under medium Bluetooth load. In terms of average delay, Fig. 16(a) shows that the IMQ-based BoTh-WiN cannot prevail this time. While BoTh-WiN can be the scheduling algorithm giving the lowest average delay at the WLAN interface (from Fig. 16(b)), it generates relatively high average delay at the Bluetooth (highest = 22.18 s) when the number of Video flows at the WLAN is small.

Besides the apparently abnormal results from BoTh-WiN, FIFO is the one giving highest average delay among the others. In particular, it increases the average delay at the WLAN interface in a much faster pace than all others (including BoTh-WiN) as the number of flows in the WLAN increases.

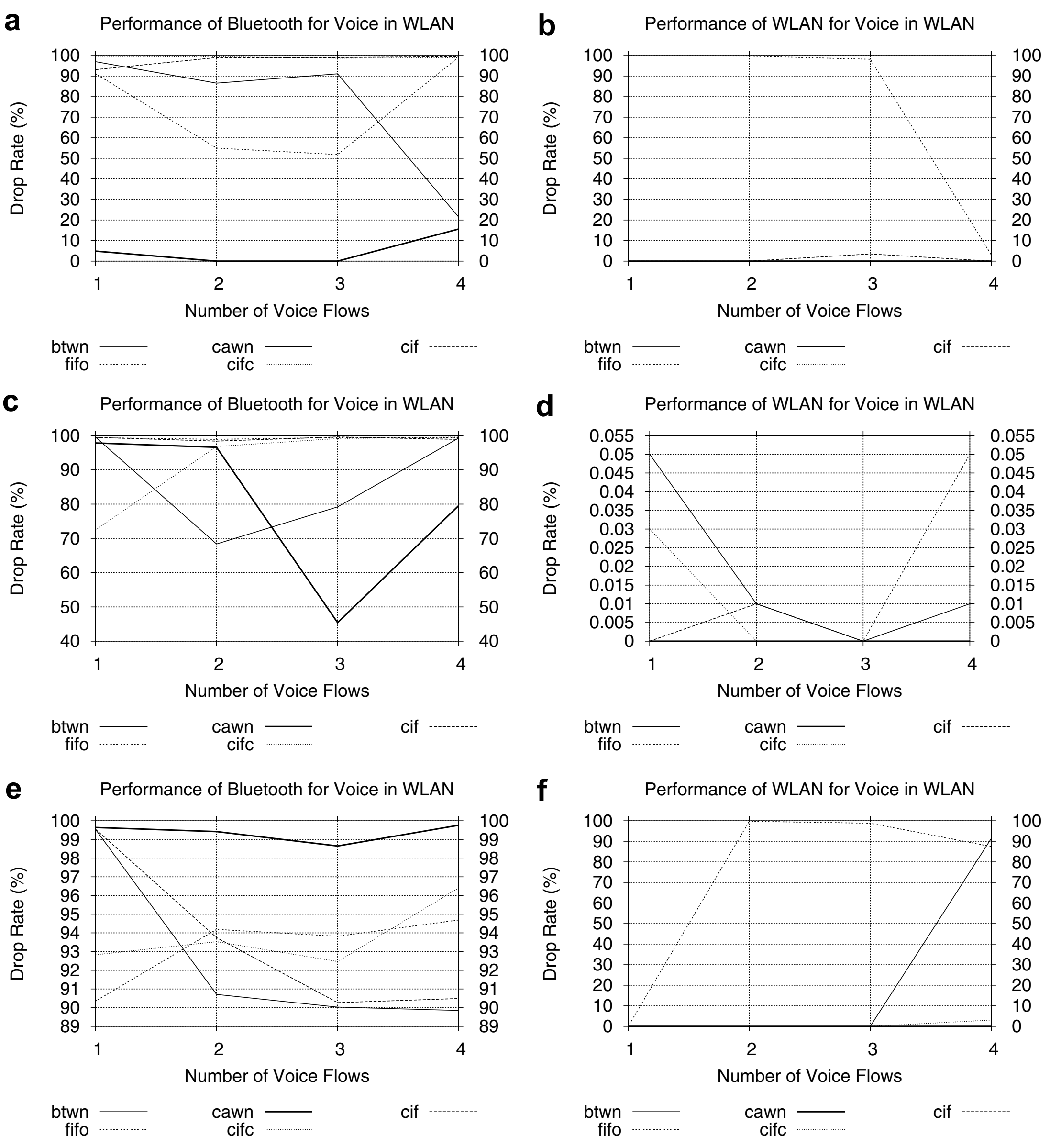


Fig. 15. Varying Voice flows in WLAN with various load levels of Bluetooth traffic. (a) Drop % (Light Bluetooth), (b) drop % (Light Bluetooth), (c) drop % (Medium Bluetooth), (d) drop % (Medium Bluetooth), (e) drop % (Heavy Bluetooth), (f) drop % (Heavy Bluetooth).

Unlike the case for the Light Bluetooth environment, the CIF algorithms give a more steady and consistent average delay this time. It is because the number of flows for Bluetooth and that for WLAN are more comparable this time. The number of Bluetooth flows are fixed to be four while that for the WLAN runs from three to six.

It is common for BoTh-WiN, CAWN and CIF Simple, all of which do not include any graceful degradation, to suffer a dip in average bit rate when the number of Video Flows in the WLAN is two, or when there are four flows in the WLAN, same number of flows in Bluetooth. However, further increase of WLAN flows also increases the average bit rate for the Bluetooth interface, whose number of flows does not change at all. Among them, CAWN gives the best average bit rate for the Bluetooth. Increasing the number of flows of WLAN increases the frequencies of CAWN to monitor the channel condition. This helps to minimize the interference effects between Bluetooth and WLAN. This can be confirmed from Fig. 16(d), which shows that the WLAN average bit rate remains high when the number of flows at the WLAN interface is at the high end.

Fig. 16(f) shows a similar situation for the WLAN interface as for the case of Light loaded Bluetooth traffic. However, from Fig. 16(e), we notice a little decrease in the drop rate in general, except for the CAWN, whose drop rate for Bluetooth is above 95% all the time. It could be deduced that the queueing disciplines become more Bluetooth favorable when the number of Bluetooth flows increases, as compared to the Light Bluetooth traffic case. Finally,

Table 3
Number of flows and flow types employed in the three levels of Bluetooth loading

	Light		Medium		Heavy	
Bluetooth	Video	0	Video	1	Video	2
	Voice	1	Voice	2	Voice	2
	FTP	1	FTP	1	FTP	2
	Video	0	Video	1	Video	2
WLAN	Voice	1–4	Voice	1-4	Voice	1–4
	Video	0	Video	1	Video	2

FIFO drop rate is below 80% when the number of Video flows at WLAN is two or three.

While CAWN cannot give a low drop rate for Bluetooth, it can decrease the drop rate at WLAN to below 90% (though still very high) when there are three or more Video flows at the WLAN traffic. (See Figs. 16(e) and (e).) This performance at the WLAN is similar to that case in Light loaded Bluetooth traffic. However, the CIF Complete algorithm also suffers from high drop rate starting from the 2-Video flows onwards, unlike that for Light loaded Bluetooth interface.

5.4. Heavily congested interference environment – multiple interference sources

We also performed experiments to examine the performance of the proposed CAWN scheme under the influence of a nearby *independent* IEEE 802.11b network consisting of three client devices communicating with an AP. We

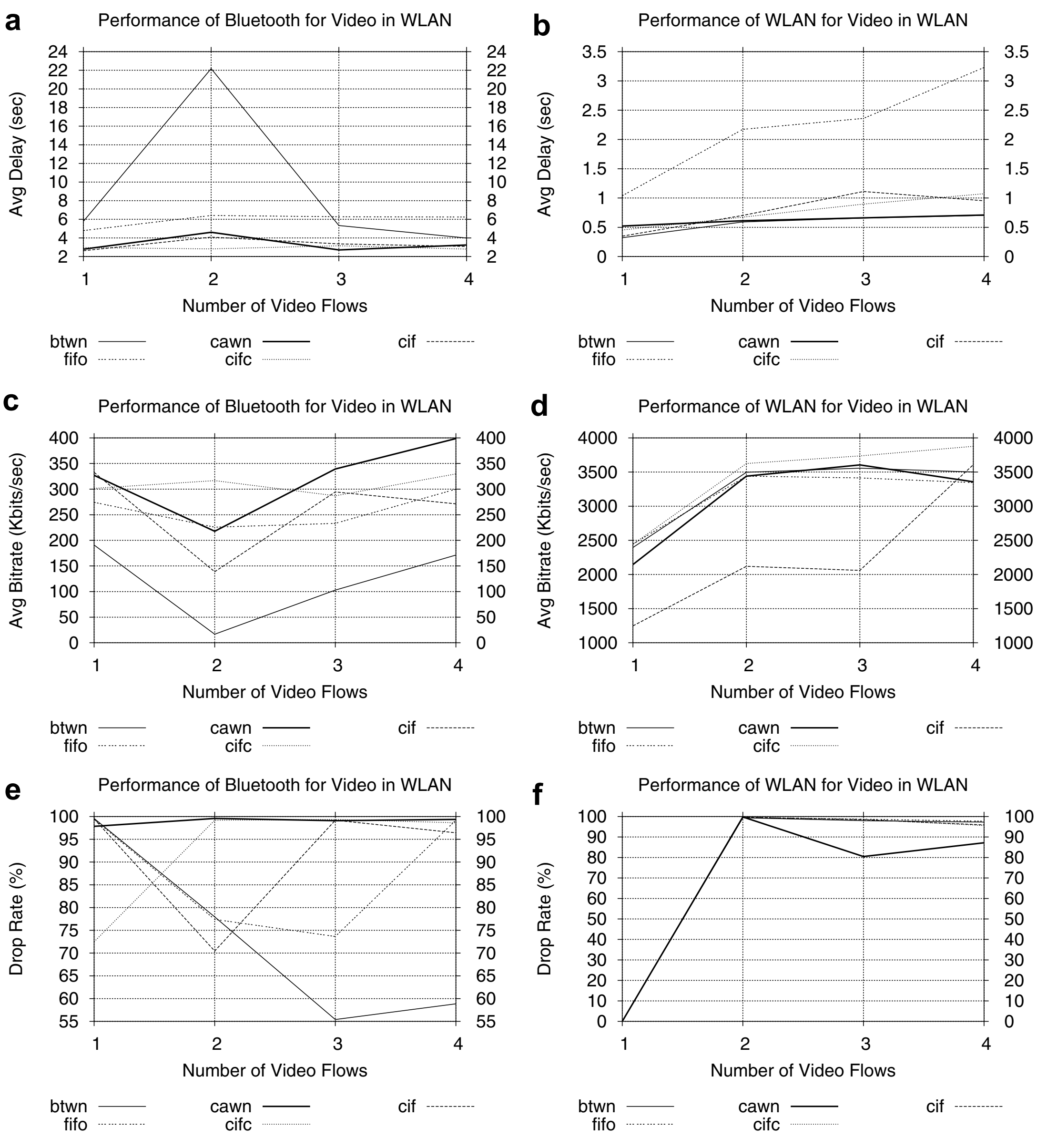


Fig. 16. Varying Video Flows in WLAN with Medium Load of Bluetooth traffic. (a) average delay, (b) average delay, (c) average bandwidth, (d) average bandwidth, (e) drop %, (f) drop %.

linearly scale the loading of such a "background" IEEE 802.11b by controlling the number of active flows in the network. At the 100% level, there are 10 FTP flows (sending files of 50 Mbytes) between each client device and the AP.

Furthermore, for the dual-protocol NAP under investigation, we also added uplink traffic flows, which are continuous uploading of large files of size 100 Mbytes. Thus, these tests aim to test the robustness of the CAWN scheme under a heavily congested interference environment.

Another objective of these experiments is to compare the performance of the proposed CAWN scheme and two other MAC/physical layer schemes, namely, the BIAS algorithm [20], and the TG2 AFH (adaptive frequency hopping) [18]. The BIAS algorithm is a Bluetooth MAC-layer approach that works by avoiding transmissions on "bad" channels. The TG2 AFH algorithm works by adaptively generating dynamic hopping sequences that consist only of "good" channels.

Fig. 17 shows the average packet loss rates of the Bluetooth flows in the NAP and those of the IEEE 802.11b flows in the NAP. We can see that as the background interference load level increases, the packet loss rates increase. The BIAS algorithm consistently performs the best among the three, while the TG2 AFH is the worst. A plausible explanation is that the AFH algorithm could not easily identify *accurately* enough "good" channels. On the other hand, the interference avoidance approach taken by the BIAS algorithm and our proposed network-layer CAWN scheme works better in such a congested environment.

5.5. Scalability tests

Our final set of experiments involved testing the CAWN scheme with a variable number of client devices: 2, 4, 6, 8, 10, and 12. For each test case, half of the client devices are Bluetooth devices while the other half are IEEE 802.11b devices. For example, for the case of 12 client devices, 6 are Bluetooth and 6 are IEEE 802.11b. This is the largest "fair" population because the NAP can only support 6 independent active Bluetooth devices. We also compared the CAWN scheme with the BIAS algorithm and the TG2 AFH scheme. The background IEEE 802.11b load level was set to be 50%. Uplink flows were also included in the NAP.

Fig. 18 shows the average packet loss rates of the three schemes. We can see that the BIAS algorithm is very robust even under a high load (i.e., more than 10 devices). On the other hand, the TG2 AFH algorithm almost breaks down at 8 client devices, with an average packet loss rate of higher than 50% for IEEE 802.11b. The performance of the proposed CAWN scheme is still quite acceptable at such

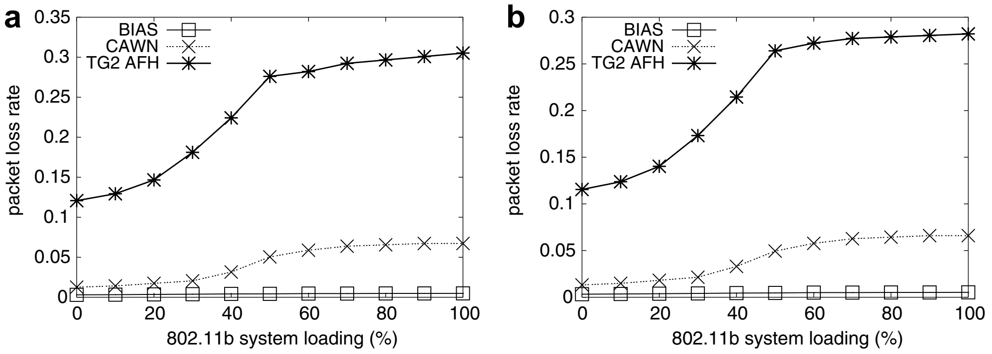


Fig. 17. Comparison of average packet loss rates generated by the three different coexistence mechanisms in the presence of an independent IEEE 802.11b source with varying load levels. (a) Bluetooth average packet loss rates, (b) IEEE 802.11b average packet loss rates.

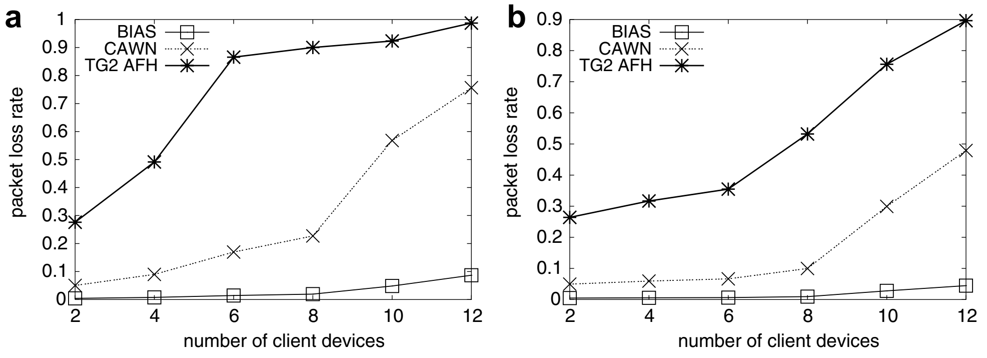


Fig. 18. Comparison of average packet loss rates generated by the three different coexistence mechanisms under different number of client devices in the presence of an independent IEEE 802.11b source with 50% load. (a) Bluetooth average packet loss rates, (b) IEEE 802.11b average packet loss rates.

load level. However, CAWN scheme fails also for the case of 12 client devices.

Summarizing the observations in the scalability tests and in the congested interference tests, we can see that the CAWN scheme outperforms the TG2 AFH algorithm considerably, indicating that a software approach is indeed an attractive solution. Although the CAWN scheme is inferior to the MAC-layer approach (i.e., BIAS algorithm), we believe that CAWN is still a practicable solution because it does not require hardware/firmware modifications as mandated by the BIAS algorithm.

6. Conclusions

In this paper, the effects of packet scheduling algorithms, *without* any help of the existing coexistence mechanisms at the MAC layer, on the interference between Bluetooth and WLAN have been carefully examined. Specifically, a Linux-based Bluetooth and WLAN Network Access Point (NAP) has been developed to test the CIFQ algorithms in real implementation and study their empirical effects to fight against the coexistence problem between Bluetooth and WLAN. It shows from the surprising and unique results of CIF-Simple that the assumption behind CIFQ (i.e., it is performing packet scheduling among different flows in the same wireless channel) is inapplicable to handle the need for coordinating packet scheduling on both the Bluetooth and WLAN interfaces.

Guided by our practical findings, we proposed and implemented two new packet scheduling algorithms in Linux, BoTh-WiN and CAWN, to provide the best trade-offs to colocated Bluetooth and WLAN traffics, as well as QoS support for different applications. The proposed algorithms have the following advantages:

1. The computational complexity is low (as indicated by the code presented in Section 3) and they do not involve any computation-expensive operations like floating point calculations, which are needed in the CIFQ algorithms.
2. Though our study is based on the NAP scenario, the implemented algorithms can also be applied to dual-protocol end-hosts with the same Linux kernel upgrade.
3. The implementations of our proposed algorithms do not require any hardware upgrade on existing Bluetooth and IEEE 802.11b transceivers.

Our show that packet scheduling with differentiation on Bluetooth and WLAN traffics can give a more balanced service share between the two interfaces. Moreover, interference coordination has to be done with the help of Channel State Information (CSI) and seamless cooperation between the two interfaces. Comparing with the MAC/physical layer schemes, the proposed CAWN scheme is still an attractive solution in that it exhibits reasonably good performance while does not require hardware/firmware modifications. Finally, while what is lost in the physical layer could not be re-gained at the network layer, the main contribution of our proposed packet scheduling schemes is to provide high bandwidth stability at high data rates and lower drop rates for both interfaces under interference.

Acknowledgement

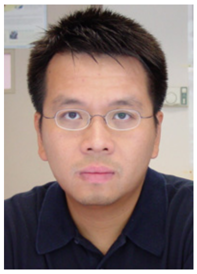
The authors thank the anonymous reviewers for their critical but constructive comments that have helped in improving the presentation of our work. The authors also thank Mr. Jean Tourrilhes, who is the author of the Wireless Extensions [38], for his insightful advice in Linux kernel Programming.

References

[1] AbsoluteValue Systems, Linux-WLAN-NG Driver for Intersil Prism2/2.5/3, <http://www.linux-wlan.org/>, 2004.
[2] W. Almesberger, Linux network traffic control – implementation overview, EPFL ICA (2001).
[3] W. Almesberger, Linux Traffic Control – Next Generation, <http://www.almesberger.net/>, Oct. 2002.
[4] ANSI/IEEE Standard 802.11, Local and Metropolitan Area Networks: Wireless LANs, 1999 Edition.
[5] S. Avallone, D. Emma, A. Pescape, D-ITG – Distributed Internet Traffic Generator, Dipartimento di Informatica e Sistemistica, Universita' degli Studi di Napoli Federico II, <http://www.grid.unina.it/software/ITG/index.php/>, 2004.
[6] L. Balliache, Differentiated Service on Linux HOWTO, <http://opalsoft.net/qos/DS.htm/>, 2004.
[7] V. Bharghavan, S. Lu, T. Nandagopal, Fair queueing in wireless networks: issues and approaches, IEEE Personal Communications 6 (1) (1999) 44–53.
[8] Y. Cao, V.O.L. Li, Scheduling algorithms in broadband wireless networks, Proc. IEEE 89 (1) (2001) 76–87.
[9] M. Beck, H. Böhme, M. Dziadzka, et al., Linux Kernel Programming, third ed., Addision Wesley, 2002.
[10] M. Beck, H. Böhme, M. Dziadzka, et al., Linux Kernel Internals, second ed., Addision Wesley, 1997.
[11] BlueZ, Official Linux Bluetooth Protocol Stack, <http://www.bluez.org/>, 2004.
[12] Bluetooth.org, Bluetooth Specifications, <http://www.bluetooth.org/spec/>, 2004.
[13] C.-F. Chiasserini, R.R. Rao, Coexistence mechanisms for interference mitigation in the 2.4-GHz ISM band, IEEE Transactions Wireless Communications 2 (5) (2003) 964–975.
[14] G. Chinn et al., Mobile PC platforms enabled with intel Centrino Mobile Technology, Intel Technology Journal 7 (2) (2003).
[15] D.P. Bovet, M. Cesati, Understanding the Linux Kernel, First ed., OŔeilly and Associates, 2001.
[16] M. Devera, HTB Linux Queueing Discipline Manual: User Guide, <http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm/>, 2004.
[17] G. Ennis, Impact of Bluetooth on 802.11 Direct Sequences, IEEE 802.11-98/319, 1998.
[18] H. Gan and B. Treister, Adaptive Frequency Hopping Implementation Proposals for IEEE 802.15.1/2 WPAN, IEEE 802.15-00/367r0, <http://www.ieee802.org/15/pub/TG2-Coexistence-Mechanisms.html/>, Nov. 2000.
[19] N. Golmie, R.E. Van Dyck, A. Soltanian, Interference of Bluetooth and IEEE 802.11: simulation modeling and performance evaluation, in: Proc. 4th ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems, pp. 11–18, 2001.
[20] N. Glomie, N. Chevrollier, O. Rebala, Bluetooth and WLAN coexistence: challenges and solutions, IEEE Wireless Communications 10 (6) (2003) 22–29.

[21] P. Goyal, H.M. Vin, H. Chen, Start-time fair queuing: a scheduling algorithm for integrated services, Proc. ACM SIGCOMM 96 (1996) 157–168.
[22] B. Hubert, G. Maxwell, R. van Mook, M. van Oosterhout, P.B. Schroeder, J. Spaans, Linux 2.4 Advanced Routing and Traffic Control HOWTO, <http://lartc.org/>, 2004.
[23] A. Kuznetsov, IPROUTE2, <ftp://ftp.inr.ac.ru/ip-routing/>, 2004.
[24] Y.-K. Kwok, Time-domain, frequency-domain, and network level resource management schemes in Bluetooth networks, in: Mihaela Cardei, Ionut Cardei, Ding-Zhu Du (Eds.), Resource Management in Wireless Networks, Kluwer Academic Publishers, 2004.
[25] J. Linseed, MEHTA: A Method for Coexistence between Co-located 802.11b and Bluetooth Systems, IEEE 802.15-00/360r0, <http://www.ieee802.org/15/pub/TG2.html/>, Nov. 2000.
[26] S. Lu, V. Bharghavan, R. Srikant, Fair Scheduling in Wireless Networks, IEEE/ACM Trans. Networking 7 (4) (1999) 473–489.
[27] J. Malinen, Host AP Driver for Intersil Prism2/2.5/3, SSH Security Communications Corp., <http://hostap.epitest.fi/>, 2004.
[28] P. McHardy, Intermediate Queueing Device, <http://www.linuximq.net/>, 2005.
[29] Mobilian, TrueRadio, <http://www.mobilian.com/.docs/pg/whitepaper.html/>, 2004.
[30] T.S.E. Ng, I. Stoica, H. Zhang, Packet fair queueing algorithms for wireless networks with location-dependent errors, Proc. INFOCOM 98 (1998) 1103–1111.
[31] C. Semeria, Supporting differentiated service classes: queue scheduling disciplines, Juniper Networks (2001).
[32] Siemens, blue2net, <http://www.siemens.at/bluetooth/indexen.htm/>, 2004.
[33] Silicon Wave, UltimateBlue, <http://www.siliconwave.com/coexistence.html/>, 2004.
[34] Silicon Wave & Intersil, Blue802 Technology, <http://www.intersil.com/cda/home/>, 2004.
[35] S. Shellhammer, Packet Error Rate of an IEEE 802.11 WLAN in the Presence of Bluetooth, IEEE 802.15-00/133r0, 2000.
[36] R.F. Smith, xnetload, <http://www.xs4all.nl/rsmith/software/>, 2004.
[37] Texas Instruments, Bluetooth and 802.11 Coexistence, <http://www.ti.com/bluetooth80211/>, 2004.
[38] J. Tourrilhes, Linux Wireless Extensions, <http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Linux.Wireless.Extensions.htm/>l, 2004.
[39] J. Tourrilhes, Linux Wireless Tools, <http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html/>, 2004.
[40] B. Treister, H.B. Gan, K.C. Chen, H.K. Chen, A. Batra, and O. Eliezer, Components of the AFH Mechanism, IEEE 802.15-01/252r0, <http://www.ieee802.org/15/pub/TG2-Coexistence-Mechanisms.html/>, May 2001.
[41] H.K. Yip, Packet Scheduling Techniques for Coordinating Colocated Bluetooth and IEEE 802.11b in a Linux Machine, M.Phil. thesis, The University of Hong Kong, Aug. 2004.

Hoi Kit Yip received his B.Eng degree in Computer Engineering and M.Phil. degree in Electrical and Electronic Engineering from the University of Hong Kong in 2002 and 2004, respectively. His research interests are in distributed mobile computing and Linux operating systems. He is now working as an immigration officer in the Government of the Hong Kong Special Administrative Region.



Yu-Kwong Kwok is an associate professor in the Department of Electrical and Electronic Engineering at the University of Hong Kong (HKU). Before joining the HKU in August 1998, he was a visiting scholar for one year in the parallel processing laboratory at the School of Electrical and Computer Engineering at Purdue University. He recently served as a visiting associate professor at the Department of Electrical Engineering–Systems at University of Southern California from August 2004 to July 2005, on his abbatical leave from HKU. He received his B.Sc. degree in computer engineering from the University of Hong Kong in 1991, the M.Phil. and Ph.D. degrees in computer science from the Hong Kong University of Science and Technology (HKUST) in 1994 and 1997, respectively. His research interests include distributed computing systems, wireless networking, and mobile computing. He is a Senior Member of the IEEE. He is also a member of the ACM, the IEEE Computer Society, and the IEEE Communications Society. He received the Outstanding Young Researcher Award from HKU in November 2004.