IMPLEMENTATION ISSUES FOR A VIDEO STREAMING SERVER IN IEEE 802.11E WLANS

Gabriel LAZAR Tudor Mihai BLAGA Virgil DOBROTA
Communications Department, Technical University of Cluj-Napoca, Romania
E-mails: {Gabriel.Lazar, Tudor.Blaga, Virgil.Dobrota}@com.utcluj.ro

Abstract: Recent years have seen a proliferation of real-time multimedia traffic over a more and more heterogeneous Internet. Video streaming at high, consistent quality over wireless links proves to be a difficult task. Several optimization techniques have been proposed and studied, mostly through theoretical analysis and simulation. This article describes the implementation of a cross-layer H.264 video streaming server and the evaluation of its performance in IEEE 802.11e WLANs. Measurements present the benefits of employing several key cross-layer mechanisms which aim to improve the video transmission quality over wireless links. A cross-layer signaling solution is implemented, which addresses important QoS issues between user-space and the kernel traffic control and device driver components. Network support for efficient multiqueue transmission is enabled in the Linux network driver. In addition, the paper discusses the implementation of an enhanced scheduling scheme for the receive-side, in order to provide prioritization of video streams over elastic traffic, and also for better control of latency and improved throughput for multimedia flows.

Keywords: *cross-layer optimization, H.264, IEEE 802.11e, multiqueue hardware, quality of service, video streaming*

I. INTRODUCTION

Recent years have seen a proliferation of real-time multimedia traffic over an increasingly heterogeneous Internet. A common case is that of video streaming over hybrid wireless-wired IP networks. The low-cost solution of over-provisioning simply cannot be used reliably in this situation, because real-time traffic has very stringent delay, jitter and bandwidth requirements. These parameters cannot be guaranteed when data is handled in a best effort manner; moreover, wireless environments have much more limited resources (e.g. bandwidth) due to channel characteristics, making over-provisioning a prohibitive solution. On top of that, users expect high service quality independent to the underlying network access technologies [1].

Therefore, real-time media streaming while maintaining a high level of perceived quality for the entire duration of the stream transmission requires cooperation of all involved "actors": each node (intermediate or terminal), and each layer from the protocol stack must collaborate and optimize functionality to achieve this goal. The strong demands imposed on video codecs and the wireless links used to transport media streams gave birth to a new paradigm in network architecture design: the cross-layer design (CLD). Wireless video streaming must ensure that variable rate data is delivered at the destination despite changing conditions, while maintaining a high user perceived quality. Applying CL optimization to multiple layers allows for optimal adaptation of the network [2].

Several cross-layer techniques have been proposed and studied, mostly through analysis and simulation. In this paper we continue our initial work presented at IEEE LANMAN 2008 [16]. We are focused now on a detailed practical evaluation of video streaming performance over IEEE 802.11e enabled wireless devices. The idea is to employ the MAC-centric CL architecture: the application layer passes its traffic information to the MAC, which decides the priority of the flows based on their QoS requirements.

The experiments were performed on a Linux based H.264 streaming server, capable of CL signaling between the sender application and the wireless card driver. Several practical issues in building a wireless, QoS and CL-enabled server are described, together with their solutions. Following an analysis regarding the interaction between QoS mechanisms and data transmission over wireless links, the correct queue discipline is enabled on the wireless interface. In addition, network support for efficient multi-queue transmission is enabled in the Linux wireless card driver.

After a detailed description of how frames are currently handled at the client endpoint, the article discusses the implementation of an enhanced scheduling scheme at the receive-side, in order to better differentiate between high priority streams and elastic traffic. While video transmission is usually performed by the multimedia server, and frame reception at the client, both mechanisms

Manuscript received March 19, 2013; revised May 28, 2013

take place and interact on routers or peer-to-peer nodes.

Measurement results present the effects of employing the proper QoS technique on the received video stream, and the benefit of driver support for multiqueue-enabled transmission hardware. A separate set of experiments further illustrates the practical outcome of changing H.264 and IEEE 802.11e parameters in the player and in the driver software respectively.

II. CROSS-LAYER AND QOS LINUX ISSUES (TRANSMISSION SIDE)

High-speed multimedia communication over wireless links is made possible through the development of the IEEE 802.11g (54 Mbps) and 802.11n (100 Mbps) physical layers. In addition, audio/video data can benefit from the new IEEE 802.11e QoS-based MAC layer to receive a suitable, real-time treatment at the Data Link Layer [3].

The most used access channel mechanism in 802.11 WLANs is the distributed coordination function (DCF). The need to provide better, preferential service for multimedia applications has led to the standardization of new 802.11e MAC layer operation; DCF is enhanced through the use of the hybrid coordination function (HCF) and its successor access method becomes the enhanced DCF channel access (EDCA). Service differentiation in EDCA is obtained using access categories (ACs). Traffic assigned to a category is sent to a separate transmission queue and each AC has different channel access parameters: contention windows ($CW_{\min}$, $CW_{\max}$), arbitrary interframe space (AIFS) and transmission opportunity duration limit ($TXOP_{\lim}$). The latter represents the maximum value of an interval used to choose a transmit opportunity time (TXOP) during which a station can send as many frames as possible. Traffic from a category with smaller $CW_{\min}$, $CW_{\max}$ or AIFS will wait less, on average, before being sent than traffic from a category where parameters have bigger values.

The standard defines four access categories, from AC0 (lowest priority) to AC3 (highest priority). AC3 and AC2 are usually reserved for multimedia transmission, while AC1 and AC0 should receive best effort and background traffic.

A. *Linux Network Subsystem*

When a packet is passed from the application layer into the kernel space using sockets, a *socket kernel buffer* is created (called `skb` for short). The socket buffer is passed through each layer of the network stack before reaching the device driver in order to be transmitted. This data structure (`struct sk_buff`) represents a packet during its entire processing lifetime inside the kernel space.

In the Linux network subsystem, the interface between protocols implemented in software and the network adapters is accomplished by the concept of network devices. The `net_device` structure represents each network device inside the Linux kernel. One of the pointers stored in this structure is the QoS-related queuing discipline associated with the packets transmitted through this network adapter (the `qdisc` item).

In the normal transmission process, the protocol instances of the higher network layers will use the function `dev_queue_xmit(skb)` to send a packet. The network device is specified in the socket buffer structure by the `skb->dev` parameter. If there is no queue management defined for that device (the `.enqueue` operation is NULL), the packet is put immediately into the hardware queue through a call to the `dev->hard_xmit_method()` method, if the driver's hardware Tx queue is not full (Figure 1). Otherwise, with no software queues defined above the hardware layer, the packet is lost before the reaching the medium.

It is the `hard_start_xmit` method's responsibility to check the hardware queues status. When the hardware queue is full, `netif_stop_queue` should be called and the `skb` has to be refused for transmission [4].

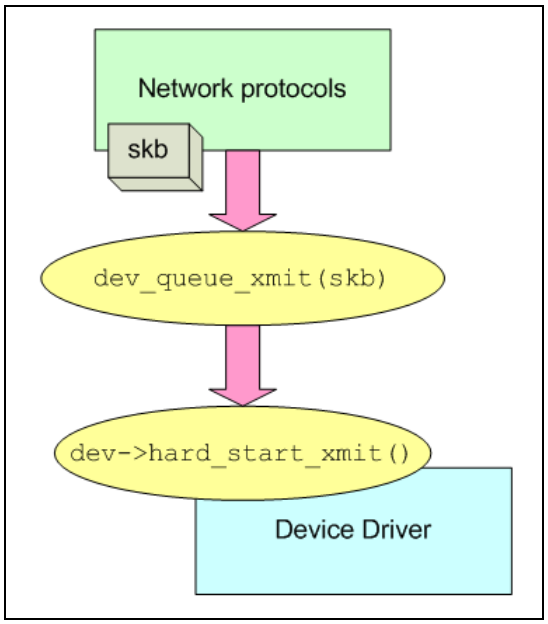

Figure 1. skb transmission using dev_queue_xmit

B. *Linux Traffic Control*

The basic building block of Linux traffic control is the queue discipline (`qdisc`). Some are simple and consist of just one queue with a simple FIFO scheduling algorithm, while other disciplines are much more complicated. A *classful qdisc* such as "priority queuing" (PRIO) contains classes in which traffic is sent according to the priority encoded in the packet or using traffic control filters. Each class can contain another class or a `qdisc`. As these internal disciplines can contain classes as well, a powerful hierarchy of traffic queues, schedulers and classifiers can be built.

Each network device has associated a queue management implementation, so that it does not rely solely on the device hardware queues. This offers not only an additional buffer space in the form of software

queues, but an entire QoS implementation inside the Linux kernel through the use of custom schedulers, configurable queues and detailed packet classification methods. This traffic control subsystem is placed between the upper network protocols and the `net_device` interface.

When the higher protocols send a packet for transmission, by calling `dev_queue_xmit(skb)`, the socket buffer is placed in one of the output queues of the network device. This is accomplished by use of the method `.enqueue` of the `qdisc` item associated with the device. This method will select the queue (if more than one) and the position of the packet inside the selected queue.

Handling of packets inside queues is done by `qdisc_run` and depends on the type of the packet scheduler associated with the queue discipline. In the case of a work-conserving-scheduler, `qdisc_run` will call `qdisc_restart` until either there are no more packets in the queue(s), or the network driver does not accept any more packets (through a `netif_stop_queue` call). Unless the device is stopped, `qdisc_restart` will use the `.dequeue` operation to extract a packet from the traffic control queue discipline(s) and call `hard_start_xmit` for the device [5].

If a non-work-conserving-scheduler is used (e.g. a *traffic shaper*), packets are serviced using a special Tx Soft IRQ triggered by a scheduling timer (Figure2).

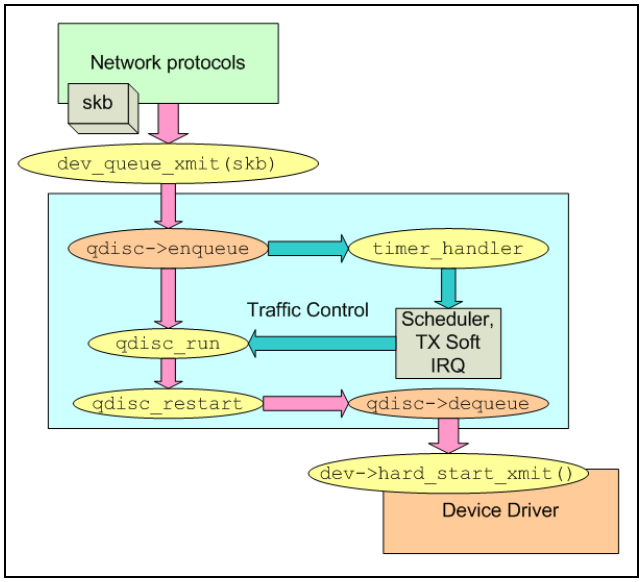

Figure 2. `skb` transmission with traffic control

C. Link Characteristics and Traffic Control Efficiency

Traditionally, network adapters used only one Tx hardware queue, associated with a FIFO scheduling policy. In this case, to implement more grained QoS policies, a different set of queues and schedulers is created and managed above the device driver by means of traffic control mechanisms. The intention is to keep the hardware queue empty as much as possible and rely on the customizable queue disciplines associated with the device by the router administrator.

When the service rate of the outgoing link is relatively constant (e.g. for wired transmission), the hardware queue of the device is kept empty by means of a shaper configured at the traffic control level. A queuing discipline with multiple classes can be configured so that its total shaping rate is slightly lower than the ongoing link rate. The limitation ensures that, when the arrival rate is bigger than the service rate, the internal queues configured for each class will fill up, and not the hardware queue (as the device transmission buffer cannot be modified). Thus, link sharing between different flows is enabled in software according to the values of the traffic control parameters (queues from each class can have different lengths, types and internal service rates).

When the outgoing link has a variable capacity (e.g. the case of wireless transmission), the situation is more complex. The shaping rate of the queue discipline should be set at every moment at a value slightly lower than the actual link rate, which in Linux is not possible without special modifications (as Linux uses *static* QoS rules). If the shaping rate is statically configured at a value close to the maximum link capacity, when the actual link rate is lower than this value, the packets will pass into the hardware queue without filling the software created buffers, and thus bypassing any QoS configuration. As the network card is not QoS-aware, all the flows will share the same FIFO hardware queue. On the other hand, if the shaping rate of the queue discipline is configured at a lower rate, when the actual link rate goes higher than this value, a part of the link capacity is wasted.

A QoS mechanism that works well even when the quality of the outgoing link is variable is *traffic prioritization*. Employment of a classful queue discipline such as PRIO, without any traffic shaping, can ensure that important packets reach the hardware queue(s) before low priority packets leave the PRIO queues. In this case, there can be no fixed service rate guarantees.

D. Devices with Multiple Hardware Queues

With the advent of network devices with multiple hardware queues, two interrelated problems arise. The first one is caused by a design limitation in the Linux network subsystem and drivers, while the second involves cross-layer QoS issues.

1. Linux Network Design Limitation

The general rules that need to be implemented by a traditional network driver in order to place packets on the Tx queue are:

- When the hardware queue is full, the driver signals the event by calling `netif_stop_queue`. The queue discipline attached to this network interface will stop sending packets to the network card.
- If the `hard_start_xmit` driver function is still called, the packet is not accepted and thus it must be returned to the above layer (by calling `.requeue`).

- When the actual transmission of packets takes place, there will be room for new packets in the hardware queue, and thus the driver will signal this event through a `netif_start_queue` call.

If a driver does not call `netif_stop_queue` when the internal queue is full, CPU usage will rise drastically as the queue discipline set up for the device will dequeue and requeue packets continuously without any knowledge of the state of the hardware queue. A well-behaved device driver uses `netif_stop_queue` and `netif_start_queue` to provide feedback to the network scheduler.

If the network adapter has multiple hardware queues, with QoS support at the hardware level, the above signaling method is not sufficient. Consider a device with two Tx hardware queues, intended for packets with different priorities (Figure3). When the low priority hardware queue is full, if the driver calls the `netif_stop_queue` method, the high priority packets will also be blocked, as the Tx queue is stopped. If the driver does not call `netif_stop_queue`, each time a low priority packet will arrive, the driver has to refuse the packet, returning it to the queue discipline only to start over again – and thus greatly consuming CPU resources without a useful reason.

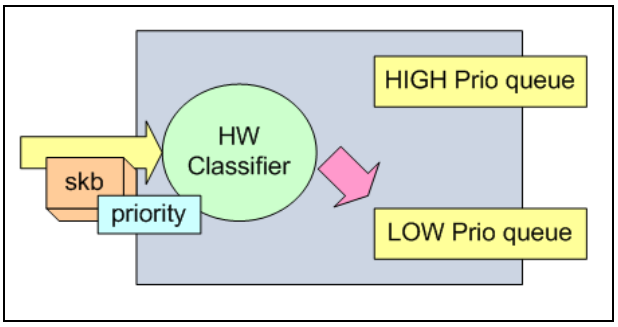

Figure 3. Network adapter with two Tx hardware queues

2. *Cross-Layer QoS Issues*

If the network adapter implements any form of QoS at the hardware level, traditional traffic control disciplines (implemented in software) must take this into account, or risk loosing some important benefits. For example, a wireless network card with IEEE 802.11e support offers higher chances of transmission for packets placed in the *Voice Access Category* hardware queue [3]. In this case, a decision should be taken about which QoS settings have to be enabled in the traffic control subsystem (that can be correlated with the hardware QoS features). It may not be feasible to completely give up the QoS facilities provided by the traffic control subsystem: for example, for certain flows, additional software queues can be configured on top of the hardware buffers to avoid packet loss.

In [4], the authors proposed a generic solution which involves adding support for multiqueue features to the Linux network schedulers implemented in various queue disciplines and also to the multiqueue devices drivers. One additional requirement is that the stack has to handle traffic from multiqueue and non-multiqueue devices in a transparent manner, with little or no additional overhead. The first part of the implementation exposes the number of hardware queues present in the device to the `net_device` structure, and enables manipulation of *each* queue state by using a new set of methods (`netif_{start|stop|wake}_subqueue`).

The cross-layer QoS coordination was implemented by the authors through the modification of the PRIO queue discipline, where the `prio_dequeue` operation checks that the hardware queue mapped to the `skb` priority is running or full, each time a packet has to be dequeued from the `qdisc`.

Both the above enhancements were recently included in the latest versions of the Linux kernel. Although very necessary, they are not sufficient: when using a wireless card, its driver also has to be modified in order to monitor and signal the state of its hardware buffers at *subqueue* level.

III. QoS LINUX ISSUES (RECEIVING SIDE)

Throughput and latency of received traffic as seen by the application layer are greatly affected by the input rate, especially if the node is CPU-bound. For fast network interfaces (e.g. Gigabit), embedded nodes and devices (e.g. routers) with a large number of network cards, this situation becomes more and more common.

The description and analysis from this section are relevant for any Linux-based node that plays one of the following roles:

- Clients that receive mixed traffic (video and elastic data)
- Servers that update their content dynamically (receiving data either on the same network interface that is used for transmission, or on another one – e.g. a Gigabit card)
- Multimedia nodes in a peer-to-peer network
- Routers in a QoS-enabled network

A. *Frame Reception*

When a network card receives either a packet or a burst of packets, it will generate an interrupt in order to inform the kernel about the arrival of data. The interrupt is handled by the *interrupt service routine* (ISR) of the network driver.

Traditionally, drivers directly push the packets to the upper layers for processing by calling the `netif_rx` routine from their ISR. The packets are enqueued to the backlog queue associated with the current CPU (`input_pkt_queue`) and a soft IRQ is scheduled, responsible for further advance of the packets in the protocol stack (Figure 4).

Before 2.4 kernels, the network subsystem provided a single backlog queue (with a default size of 300 packets). Only one packet at a time could enter the system, regardless of the number of processors or network interfaces available. The 2.4 kernels introduced the

concept of "per CPU queues" together with round-robin interrupt scheduling. These changes enabled concurrent packet processing. To avoid packet reordering, input processing for each network interface has to be assigned to a single CPU via IRQ affinity.

In order to alleviate the problem of system congestion collapse under heavy load, mainly caused by *interrupt livelock* [6], the 2.6 kernels introduced a new API (NAPI).

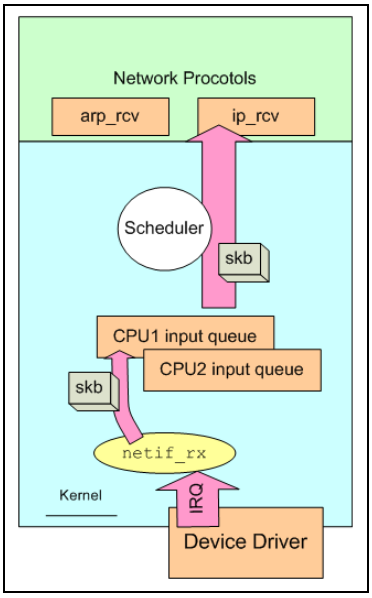

Figure 4. Packet reception (old API)

This solution combines interrupt processing with polling in order to reduce the number of interrupt requests at high input traffic rates. Under heavy load, only the first packet in a burst will generate an interrupt, and all subsequent packets will be picked up by polling. Without NAPI, the interrupt rate can grow so much for a fast interface that packets will be continuously pushed at high priority to the full backlog queue, while no time slices are left for other useful processing. Therefore, packets will be dropped without reaching the upper layers.

The NAPI implementation has two components: a generic infrastructure integrated into the kernel, and several requirements for network drivers.

Instead of calling `netif_rx` to push a frame on the backlog queue, a NAPI compliant driver will schedule a soft IRQ (NET_RX_SOFTIRQ) for frame reception through a call to `netif_rx_schedule`. This will register the input device into a CPU polling list (`poll_list`) and will temporarily turn off the hardware RX interrupts for that interface. When the software IRQ executes, `net_rx_action` is called, the list of devices in polling state is examined and the virtual `poll` function is called for each device. The polling routine has to be implemented by the network driver and its role is to fetch the packets from its internal queues. The devices in the polling list are serviced in a round-robin fashion. For each interface driver, there are a maximum number of packets that can be processed during a `poll` call. If the queue is not cleared during a given limit of time or after processing its frame quota, the device is pushed to the end of the list for further processing in the next slot. When the queue eventually becomes empty, the interrupt notifications are enabled for the device, and the interface is removed from the polling list (Figure 5).

With the generic NAPI infrastructure in place, non-NAPI device drivers will push their packets through the `netif_rx` call to the CPU backlog queue associated with a special device called `backlog_dev`. The `netif_rx_schedule` is further called for this pseudo-device acting as a wrapper for all non-NAPI drivers. The `backlog_dev` is enqueued to the CPU polling list, thus enabling round-robin fairness between NAPI and non-NAPI compliant device drivers. If a CPU backlog queue fills up, the network subsystem will enter the *throttled* state: no packets will be accepted until the queue becomes empty again. Unless the non-NAPI driver implements some form of internal flow control to disable interrupt generation under congestion, the system is still exposed to the interrupt livelock issues.

B. *Ingress Traffic Control*

Linux traffic control mechanisms have been designed to implement QoS on the egress path. On packet reception, traffic can only be classified using *policing filters* in order to drop, reclassify or accept incoming frames.

Before passing the packets to the upper protocol handlers, the polling scheduler checks if the device has any policing filters set up. Actions configured for these filters can include for example the dropping of low priority packets when their input rate is higher than a certain threshold. In this case, the policed frames will be discarded before reaching the upper protocol handlers.

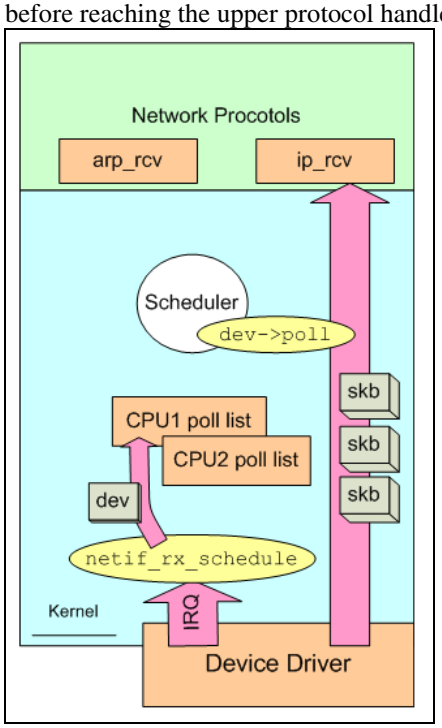

Figure 5. Packet reception (NAPI)

A different solution is the Intermediate Queuing Device (IMQ) [7]. The IMQ kernel patch provides a special input device which enables the use of egress queuing disciplines for ingress shaping. In addition, as IMQ receives traffic from all interfaces, QoS rules configured by the `qdiscs` attached to the IMQ are applied globally, and not on a per-interface basis. Because IMQ is implemented at the networking layer, it benefits from the packet classification syntax provided by `netfilter` [8]. At the same time, employing IMQ has the disadvantage that ingress traffic shaping takes place on already processed packets. Therefore IMQ should be moved to the device layer (where the policing filters are also defined), so that packets received by the network interfaces will enter the IMQ as soon as possible.

C. *QoS Issues*

The scheduling process for incoming packets at the data link layer is illustrated in an abstracted form in Figure 6. Packets are serviced using a round-robin scheduler with quotas and per-queue time limits. Packet differentiation at this point is reduced to the device queue identifier. There is no support for recognition and better treatment of high priority packets. One form of (limited) prioritization can be obtained on a per-interface basis by increasing the quota (weight) of important devices and decreasing the quota of all other interfaces.

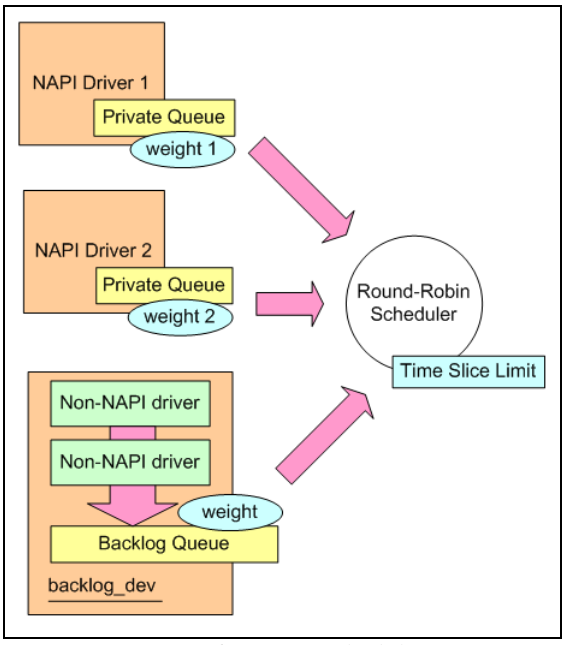

Figure 6. Ingress scheduling

When a packet burst is enqueued into the internal queue of a device, it is the driver's responsibility to recognize high priority frames. For a NAPI device, the scheduler will call its `poll` method in order to service (some of) the packets. Unless the driver is QoS-enabled and sends the high priority frames first, these packets will experience an additional latency caused by all unimportant traffic placed before them in the interface queue. For non-NAPI devices, an additional problem is the backlog buffer where packets are pushed. Because more than one interface share the same FIFO queue, QoS prioritization is easily negated. In addition, there can be several backlog queues, one for each CPU present in the system.

The round-robin scheduler also affects the handling of high priority frames. The fixed order in which queues are serviced causes additional latencies. In the worst case, important packets have to wait in their buffer before being serviced until all quota (or time slice) of every other queue is consumed. Low priority packets can be serviced, sent to upper layers and processed before an important video frame gets to be dequeued by the scheduler.

D. *Ingress Prioritization Solutions*

Following a bottom-up approach, both the device drivers and the generic network subsystem could benefit from better ingress QoS mechanisms. At the network interface layer, each driver should implement packet differentiation and prioritization of important frames in order to reduce latency. Packet recognition also proves very useful when queues are full, because low priority packets can be dropped before important frames.

For a non-NAPI device, both `dequeue` and `drop` policies of the backlog queue can be changed to favor important packets. A generic solution implies early frame classification using the same filtering rules as those used by the traffic control policing filters.

To reduce processing time for low priority packets, policing filters can be configured on top of the scheduler in order to drop excessive, unimportant traffic before it would reach the upper protocol handlers. Limiting the bandwidth of low priority packets can decrease the overall latency experienced by multimedia flows. This method is easy to implement because there is no need to modify the current kernel traffic control subsystem.

Another method of blocking low priority packets to reach the application layer before important frames is implemented using IMQ. When enabled, the IMQ is set up on top of the round robin scheduler (actually, at the network layer). A strict priority queuing discipline can be attached to the IMQ, with multiple classes of traffic. Each traffic class includes an additional queue to store incoming frames serviced by the round-robin scheduler, but in this case the packets are placed into queues according to their priorities. While these additional buffers can also cause latency, the PRIO scheduler will always send the important frames to be processed in the upper layers strictly before low priority packets.

A better and more difficult solution is to modify the round-robin scheduler in order to integrate strict queue and packet prioritization. If early frame classification support is already in place, in order to enable packet differentiation at driver layer, important packets can be recognized easily by the scheduler to provide better treatment. In an ideal case, each interface and device driver would provide separate queues for different classes

of traffic, so that the scheduler could service important packets based only on the queue priority advertised by the device driver. Therefore, our proposed generic solution requires the following changes (Figure 7):

1. Modification of the round robin scheduler to implement classes of traffic with strict priority service.
2. Methods that enable early packet classification, so that they can be called even from the device driver layer.
3. Placement of important packets into separate queues as early as possible in the device driver. The priority of each queue should be advertised to the scheduler.
4. If no separate queues are available, the driver should place the important packets in front of its internal queue. In this case, the priority of the queue is dynamic: each time the buffer contains important frames, the driver should advertise the queue as high priority. Frame recognition is enabled by the methods proposed at 2. It is important to have customizable classification rules set up at higher layers because the importance of packets can not be decided solely by the device driver.
5. For non-NAPI devices, CPU backlog queues should modify their `dequeue` policy to service important packets first.
6. Different queues that belong to the same class of traffic should be serviced in a round-robin manner.

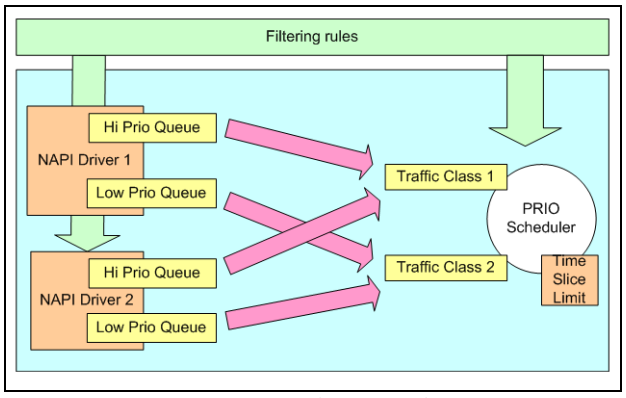

Figure 7. A custom solution for better traffic prioritization at reception

Enabling classes of traffic with strict prioritization and a customized `dequeue` operation for single queues aims to reduce latency and, together with the `drop` policy, increases the throughput of high priority traffic.

IV. IMPLEMENTATION

The CL-enabled Linux solution was implemented using several open-source software components (Figure 8):

- *User-space:* a modified Linux build of the VideoLan Client (VLC) [9], capable of marking packets either at flow level, or individually (by taking into account their importance in the video stream).
- *Kernel (traffic control):* a recent kernel (2.6.24.3) with multiqueue support in the network subsystem [10]. The queue discipline attached to the wireless interface is PRIO (also multiqueue aware).
- *Kernel (driver):* a modified version of the MadWifi driver for WLAN network cards based on the Atheros chipset. The driver can classify the incoming packets according to their marking, in order to select the required MAC access class for transmission [11]. The driver was modified so that it can signal the state of each device hardware queue to the attached `qdisc`.

A practical issue that had to be addressed was caused by the way in which the multiqueue support in the PRIO queuing discipline was implemented. When the PRIO `qdisc` creates its internal classes, it maps them to the hardware queues in the same order in which the hardware buffers are reported by the driver. Thus, the first hardware queue is mapped to the first PRIO class (with the highest priority), the second hardware buffer is mapped to the next priority, and so on. For wireless cards, the IEEE 802.11e standard specifies the access categories in a reversed order: AC0 corresponds to the lowest priority and AC3 to the highest. Due to this fact, the priorities of the hardware queues as seen and used by the driver are completely reversed compared with the mapping performed by the PRIO queuing discipline. For experiments, the priorities of the PRIO `qdisc` can be reversed by changing the order in which the queues are serviced, but a generic mapping solution is still needed.

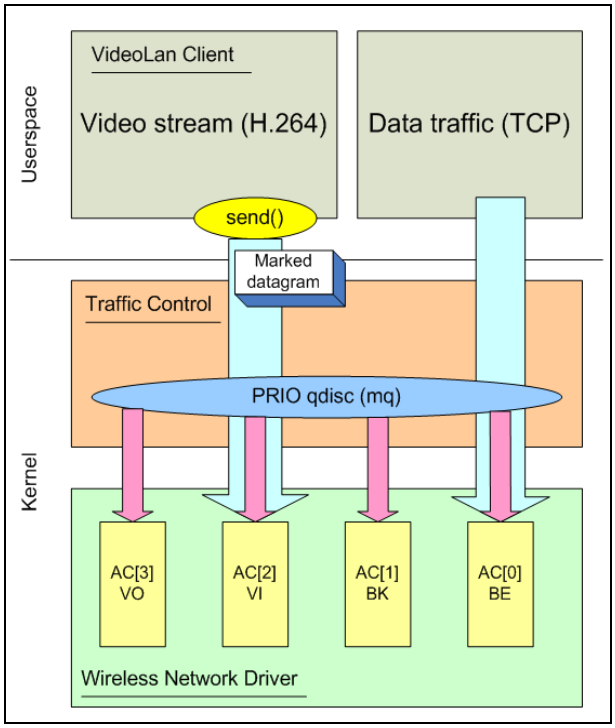

Figure 8. CL-enabled Linux video streaming server

V. VIDEO STREAMING SCENARIOS

The testbed necessary for experiments was setup by connecting the CL-enabled station with a Linux machine – the receiver, through an access point (Figure 9).

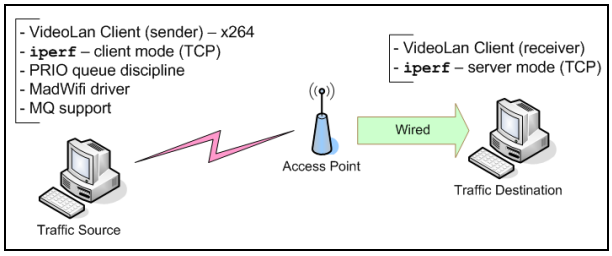

Figure 9. Testbed for video and data streaming

In each scenario, two flows were sent between the source and the destination: an H.264 video stream and a TCP stream (as background traffic). The VLC software can play the role of the sender and the receiver for video streaming, and thus it was used at both transmission ends. The H.264 codec integrated into VLC is x264 [12]. For data traffic, the iperf tool [13] was also installed on both stations, in sender mode and receiver mode respectively. On the wireless adapter (a SMC device based on the Atheros 5212 chipset), the physical rate was set to a fixed value of 9Mbps. The PRIO queuing discipline was configured for the wireless interface, so that packets marked with a Type-of-Service (TOS) value of 0xB8 were serviced with a higher priority than Best-Effort packets. In addition, the modified MadWifi driver will also send the important packets to the AC2 (Video) hardware queue. For each scenario, a TCP connection was first established, 30 seconds before starting the video flow. A 10-seconds Foreman sequence, encoded on the fly by the VLC player, was then sent to the destination. Depending on the scenario, none, some or all of the datagrams in the video stream were marked for the Video AC. In each case, the background TCP flow was sent as Best-Effort traffic (i.e. no marking). Default IEEE 802.11e values for the two ACs (configured by the MadWifi driver) are shown in Table I.

Table I. Default IEEE 802.11e driver values

Access Category	CW_{min}	CW_{max}	AIFS multiplier	TXOP limit [bytes]
Best-Effort	16	1024	2	2048
Video	8	16	2	3008

The H.264 encoded stream was saved at each communication end, and the two resulting files were compared in order to determine the quality of the transmission. The difference in quality was measured by computing the Structural Similarity (SSIM) index between each frame of the two video files [14]. An average SSIM for the entire h.264 flow was determined as the mean value of the SSIMs of all frames.

A. *Scenario 1*

The first scenario compares the performance of the following experiments:

1. Original MadWifi driver (i.e. no multiqueue support in the driver): all packets are transmitted through the Best-Effort Access Category hardware queue. When the buffer is full, the packets are returned to the PRIO discipline, but the driver still accepts packets (the full-queue event is not signaled).
2. Multiqueue support enabled, and all video packets are marked as high priority (TOS = 0xB8), and sent through the Video Access Category hardware queue.
3. Multiqueue support enabled, but no marking for video packets. The difference between this case and the first one is that, when the hardware queue is full, the driver will signal the event so that PRIO will not service packets until there is room in the hardware buffer.
4. Multiqueue support enabled, with only some of the video packets marked for high priority transmission. Based on the importance assigned to the H.264 NAL units by the x264 encoder, a priority value is determined (either Video or Best-Effort class) for each packet.

The average SSIM values obtained for the four cases are shown in Table II, together with the average peak signal-to-noise (Average PSNR) quality measure. PSNR is a classical measure for image quality, but it does not match very well to human perception. On the other hands, SSIM supplies predictions which are much closer to the results obtained through subjective testing.

Table II. Scenario I - Average SSIM and PSNR

Test No.	Average SSIM index	Average PSNR [dB]
1	0.60	18.52
2	0.89	71.66
3	0.58	18.06
4	0.44	16.41

The comparative evolution of the SSIM index in each case is illustrated in Figure 10. Results of the second test, from Table II and Figure 10, show that prioritized video packets clearly benefit from the multiqueue addition, and the gain in quality at the receiver is significant. If we compare the first case with the third, results seem very similar. It may appear that, if only one hardware queue is used (i.e. when all packets have the same priority), nothing is gained if the driver signals the full-queue event to the attached queue discipline. In reality, there is a difference in behavior, that can be observed in the VLC send() routine. In the first case, because the driver never

acknowledges its full-queue situation, the PRIO discipline in turn will always accept packets from the protocol stack. This means that a send() call issued by an application will always succeed, even if the hardware queue is full.

If the driver *does* signal when a subqueue is full, PRIO will stop servicing and also will not accept new packets from the above protocol stack. In this case, a `send()` call will return an error such as "resource temporarily not available", so that the issuing application can react accordingly (e.g. the `send()` call can be repeated with the same input buffer, or, if too much time has elapsed, it can be skipped). The error can be seen as cross-layer signaling, or feedback, between the driver and application. The worst received quality was obtained during the fourth test. In this case, a number of datagrams from the H.264 stream were prioritized, and were sent (earlier) through the *Video Access Category* hardware queue, while the remaining datagrams were transmitted through the *Best-Effort* queue. As a result, many packets were reordered. At destination, as the current version of VLC cannot establish the correct packet order, each out-of-sequence datagram was ignored.

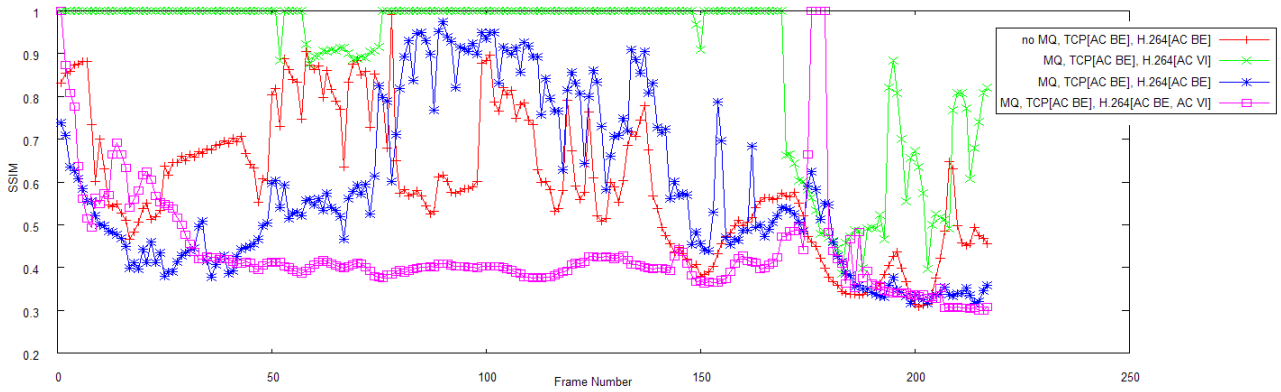

Figure 10. Scenario 1 - Per frame SSIM evolution

B. Scenario 2

The second scenario measures the effect on the prioritized video transmission at different values of the CW_{min} parameter for the *Best-Effort* AC hardware queue. Previous studies have demonstrated that CW_{min} has a greater effect than CW_{max} in most situations [12]. As the difference between the CW_{min} values for AC0 and AC2 increases, video packets obtain a higher priority for channel access compared with the TCP traffic. Therefore, the quality of the received video improves when CW_{min} for AC0 increases (Figure 11).

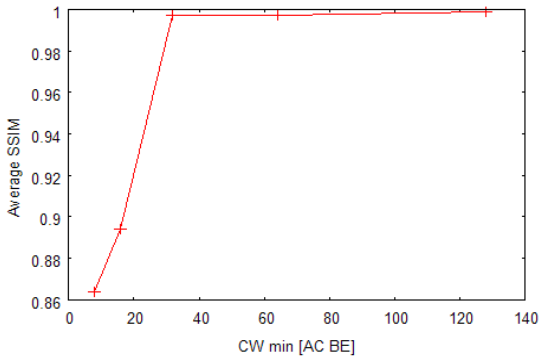

Figure 11. Scenario 2 – Average SSIM at different AC0 CW min values

C. Scenario 3

The third scenario evaluates the effect of the different H.264 encoding parameters (modifying the number of I, P and B frames) on the quality of the video transmission and also on the rate of the encoded flow (Table III). Similar with the previous experiments, video data was transmitted at a higher priority (AC2) than the TCP background traffic (AC0). Even so, it can be seen that a bigger number of B-frames increases the degradation of the received flow, as each error caused by a lost packet has a bigger effect at the decoder. The results also suggest a possible CL optimization that takes into account both the available rate on the wireless link, and the effect of different values for the H.264 encoding parameters. If the physical rate provided by the wireless card is larger than the video stream transmission rate, the number of I and P frames could be increased by the encoding software, in order to keep a high video quality at the receiver. Increasing directly the target encoding rate, without modification of the percentages of I, P and B frames may provide a better overall quality, but will not protect against lost packets.

TABLE III: SCENARIO 3 - EFFECTS OF H.264 ENCODING

I [%]	P [%]	B [%]	H.264 rate [Mbps]	Avg. SSIM
4.30	22.97	72.73	3.43	0.73
4.24	25.85	69.91	4.63	0.75
4.07	33.33	62.60	4.81	0.76
4.00	96.00	0.00	4.95	0.89
4.80	95.2	0.00	5.00	0.92

VI. CONCLUSIONS

The paper demonstrated that a practical evaluation of H.264 video streaming over WLAN devices with multiple hardware queues is feasible. Furthermore, cross-layer signaling is also possible, by packet prioritization at both user-space and kernel (traffic control subsystem and driver). Bottom-up signaling could help video-streaming applications to adapt to the current hardware buffer availability. Video flows can avoid competing with background traffic (e.g. TCP) because they benefit of better access to the channel, due to the implemented mechanism. Practical evaluation showed that it is recommended to send the datagrams that belong to the same flow through a single hardware queue, in order to avoid packet reordering (at least for legacy applications). Measurements illustrated that, by changing the H.264 parameters (percentage of I, P and B frames), the average Structural Similarity index could be increased to a high quality level (greater than 0.9). From IEEE 802.11e point of view, the difference between minimum contention windows of Video Access Category and Best-Effort Access Category can also improve the quality of the perceived video. On the receive-side endpoint, the current state of Linux network subsystem is analyzed and a solution to improve the latency and throughput of high priority traffic is proposed. Future testbeds will involve a larger number of nodes (senders and receivers), including also UDP background traffic. In addition, experiments will measure the performance of high traffic input for nodes that use a combination of high speed and low speed interfaces with both NAPI and non-NAPI device drivers. Results will be compared with a similar set of experiments performed on a Linux node that implements the customized traffic handling scheduler.

REFERENCES

[1] J.Soldatos, E.Vayias and G. Kormenzatas, "On the Building Blocks of Quality of Service in Heterogeneous IP Networks", *IEEE CST,* vol. 7, issue 1, pp. 69-88, First Quarter 2005.

[2] S.Khan, Y.Peng, E.Steinbach, M. Sgroi and W.Keller, "Application-Driven Cross-Layer Optimization for Video Streaming over Wireless Networks", *IEEE Communication Magazine,* vol. 44, no.1, pp. 122-130, January 2006.

[3] IEEE 802.11e, "Wireless LAN Medium Access Control (MAC) Enhancements for Quality of Service (QoS)", 802.11e Standard, 2005

[4] Z.Yi and P.P.Waskiewicz, Jr., "Enabling Linux Network Support of Hardware Multiqueue Devices", *Proceedings of the Linux Symposium,* pp. 305-310, June 2007.

[5] K. Wehrle, F. Pählke, H. Ritter, D. Müller and M. Bechler, "The Linux® Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel", August 2004.

[6] J.C. Mogul and K.K. Ramakrishnan, "Eliminating Receive Livelock in an Interrupt-Driven Kernel", ACM Transactions on Computer Systems, vol. 15, no. 3, pp. 217-252, August 1997.

[7] Linux IMQ – Intermediate Queueing Device. Available: http://www.linuximq.net

[8] Netfilter – Linux Packet Filtering Framework. Available: http://www.netfilter.org

[9] VideoLan – VLC media player. Available: http://www.videolan.org

[10] The Linux Kernel Archives. Available: http://kernel.org

[11] MadWifi –Linux driver for WLAN devices with Atheros chipsets. Available: http://madwifi.org

[12] x264 – Free H264/AVC decoder. Available: http://www.videolan.org/developers/x264.html

[13] iperf – The TCP/UDP bandwidth measurement tool. Available: http://dast.nlanr.net/Projects/Iperf

[14] SSIM – The SSIM Index for Image Quality Assessment. Available: http://www.ece.uwaterloo.ca/~z70wang/research/ssim

[15] Y.Cheng *et al.*, "A Cross-Layer Approach for WLAN Voice Capacity Planning", *IEEE JSAC*, vol.25, no.4, pp. 678-88, May 2007

[16] G.Lazar, V.Dobrota and T.Blaga, "Practical Evaluation of H.264 Video Streaming over IEEE 802.11e Devices by Cross-Layer Signaling", *Proceedings of the 16th IEEE Workshop on Local and Metropolitan Area Networks LANMAN 2008*, Cluj-Napoca, Romania, September 3-6, 2008, pp.13-18, ISBN: 978-1-4244-2027-8