The Chinese University of Hong Kong
Computer Science and Engineering Department

Spring Semester Report of

Final Year Project **LYU9903**

QoS Schemes in Wireless Networks

Written by **Ho Pun Mo** (97678721)
Supervised by Prof. Lyu Rung Tsong Michael
Marked by Prof. Ng Kam Wing

Printed on April 18, 2000

Contents

Chapter 1

Introduction

We have developed our QoS scheme in wireless network throughout the semesters. The stuffs related to our project will be described in this report.

We have chosen Linux as our development platform. The kernel version is 2.2.5-15CLE under the Redhat 6.0 distribution with CLE (Chinese Language Extensions). We have used two brands of wireless equipment, namely Proxim and WaveLAN. But we have sticked to WaveLAN at most of the time. The WaveLAN PCMCIA cards are in IEEE 802.11 standard, with about 2M bps bandwidth. There are other programmes needed for our system, say the iproute2+tc package, the RSVP daemon. If you are interested and would like to implement the system, you can feel free to ask us for help. We can be reached at pmho1@cse.cuhk.edu.hk and mng@cse.cuhk.edu.hk.

1.1 Review what we have done in last semester

Our main task in the last semester was to study the QoS scheme developed by the previous final year project team LYU9802, whose target aimed at QoS in wired LAN. Measurement was performed for their scheme too.

Their scheme combined bandwidth reservation and packet scheduling, which was, to some extent, a general method to achieve QoS. And it was also suggested and discussed in many papers. They were working in Linux too.

- **What is RSVP?**
  RSVP (Resource Reservation Protocol) is a protocol that allows channels or paths on the networks to be reserved for the transmission of video and other high-bandwidth messages. RSVP is part of the Internet Integrated Service (IIS) model, which ensures: best-effort service, real-time service, and controlled link-sharing, RSVP protocol is used by a host to request specific qualities of services from the network for particular application data streams or flows. It is also used by routers to deliver quality of service requests to all nodes along the path(s) of the flows and to establish and maintain state to provide the requested service.

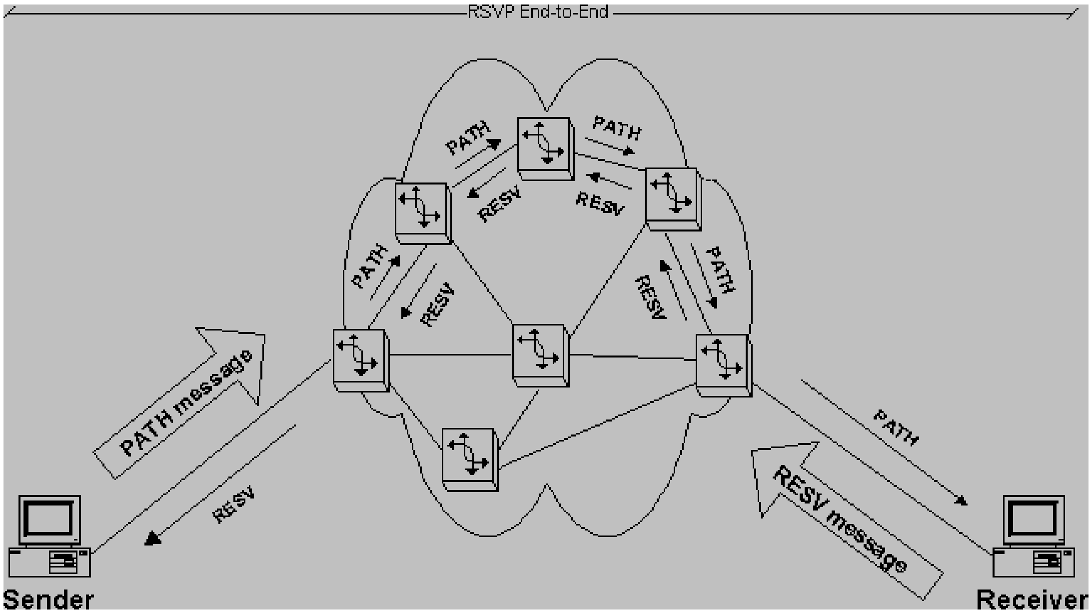

Figure 1.1: RSVP Overview.

Below is the flow of RSVP:

1. Senders characterize outgoing traffic in terms of the upper and lower bounds of bandwidth, delay, and jitter. RSVP sends PATH messages from the sender that contains this traffic specification (TSpec) information to the (unicast or multicast receiver(s)) destination address. Each RSVP-enabled router along the downstream route establishes a "path-state" that includes the previous source address of the PATH message(i.e. the next hop "upstream" towards the sender)
2. To make a resource reservation, receivers send a RESV (reservation request) message "upstream" to the source of the PATH message. In addition to the TSpec, the RESV message includes the QoS level required in an RSpec, and characterizes the packets for which the reservation is being made (e.g. the transport protocol and port number), called the "filter spec". Together, the RSpec and filter-spec represent "flow-descriptor" that routers use to identify reservations.
3. When an RSVP router receivers and RESV message, it uses the admission control process to authenticate the request and allocate the necessary resources. If the request accepted, the router sends the RESV upstream to the next router.
4. When the last router receives the RESV and accepts the request, it sends a confirmation message back to the receiver.
5. There is an explicit tear-down process for a reservation when sender or receiver ends an RSVP session.

- **What is CBQ?**
  Class based queueing (CBQ) is a packet scheduling algorithm proposed by Van

Jacobson. It is an approach of hierarchical link-sharing management [7]. It outlines a set of flexible, efficiently implemented gateway mechanisms that can meet a range of service and link-sharing requirements. One requirement for link-sharing is to share bandwidth on a link between multiple organizations, where each organization wants to receive a guaranteed share of the link bandwidth during congestion, but where bandwidth that is not being used by one organization should be available to other organizations sharing the link. Examples range from the multiple agencies that share the Trans-Altantic FAT pipe and each pay a fixed share of the costs to individuals who share a single ISDN line. Another requirement for link sharing is to share bandwidth on a link between different protocol families (e.g., IP and SNA), where controlled link sharing is desired because the different protocol families have different responses to congestion. A third example for link sharing is to share bandwidth on a link between different traffic types, such as telnet, ftp, or real-time audio and video.

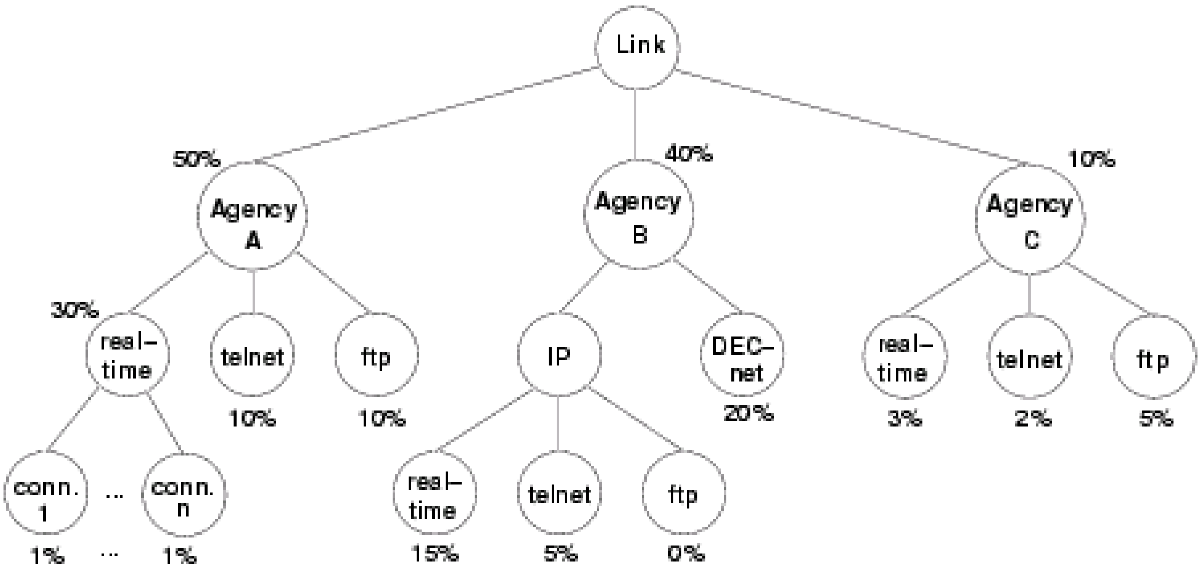

Figure 1.2: A hierarchical link-sharing structure.

CBQ is based on the link-sharing structure, an example is shown at figure 1.2. There are different constraints associated with different nodes of the structure. When the constraints are violated, the nodes that involved are needed to be regulated. [7] has a detail description on this.

We have measured the performance of their system in the last semester. The details can be refer to our report in the last term. Only one of the experiments will be reminded here as reference.

We have performed six sets of experiments (three for wired version while another three for the corresponding wireless version). The result can be said to be almost the same.

That is, the applications that needed QoS (MP3 player) could be run smoothly even with the interference of FTP applications. One of our experiments and the result is described below.

This is our third experiments for the wired LAN:

> In this experiment, the limit of bandwidth was 400kbits. The first measurement was done by using QoS while the second one was done without applying QoS. At the start of the experiment, the MP3 player was run. Sixty seconds later, the FTP application was run. And another sixty seconds then, another FTP application was run. All data and MP3 song are got from the same server via the wired network.
>
> It is shown from figure 1.3 and 1.4 that with QoS, MP3 player played smoothly and last for 3 minutes. However, without QoS, MP3 player last for about 5 minutes, and you could hear some click sounds when the song was playing.

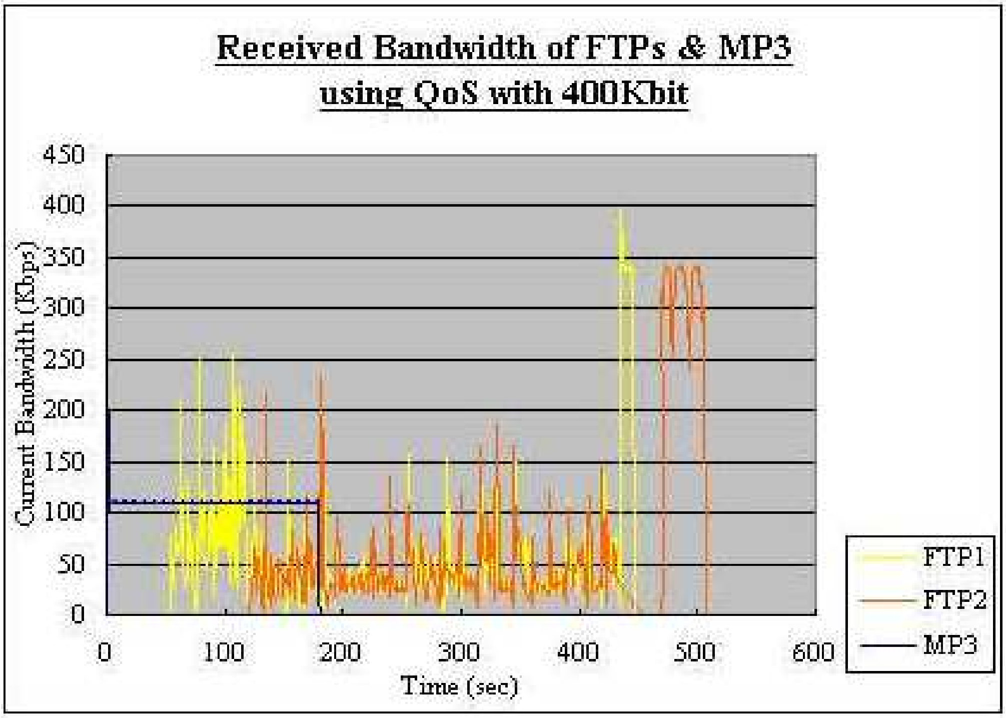

Figure 1.3: Result of experiment 3: with QoS.

In the last semester, we have also done other tasks, such as porting LYU9802's QoS scheme to the wireless environment, the initial version of our video application and etc. Details can be referred to our previous report.

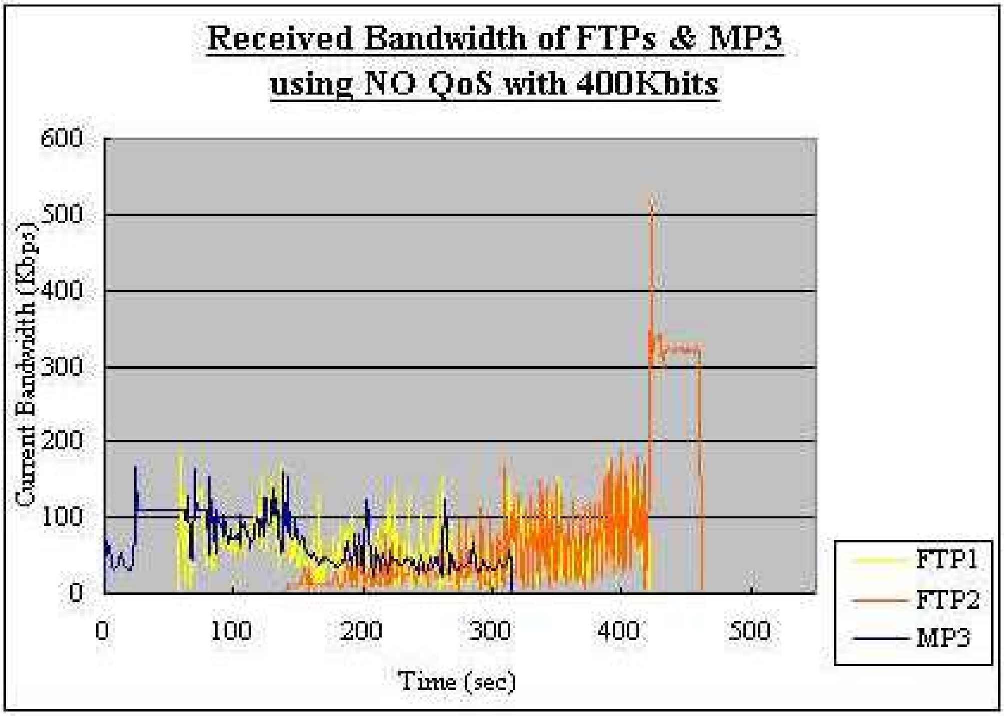

Figure 1.4: Result of experiment 3: without QoS.

1.2 What we have done in this semester

I will just outline here what we have done in this semester. Details are going to be presented later throughout the report.

We have developed our QoS scheme, which aims at providing QoS for wireless LAN. But one thing should be made clear first is that, there are many areas that QoS can be applied to. We have thought of many schemes, but what we can do is little. And QoS for wireless LAN is not necessary same as that for wired LAN. In order to avoid mixing up with wired LAN, we have chosen a scheme we have thought of that aims at providing QoS only for wireless LAN.

Besides our scheme, we have also implemented a video player, which is mainly for MPEG-1 format. RAPI (which will be introduced later, it is for RSVP) is used in it.

Moreover, as our project aims at providing QoS, and not at producing a product, number of lines of codes means nothing to us. And it is harder than just producing codes, since what we need to do is to control quality, which is a guarantee. And guarantee might not always be guaranteed!.

1.3 Division of labour

Although we have divided our project into two parts, we still work together as it is hard to divide the work sharply. In a few words, we have divided our project into two parts, one for QoS and one for video application.

I am responsible for the QoS part while my partner is responsible for the video application. Details of the video application can be referred to the report of my partner **Ng Maggie (mng@cse.cuhk.edu.hk)**. Nevertheless, I will briefly describe the video application also.

Chapter 2

QoS Scheme in Wireless LAN

QoS has been talked for many years, and the shape is gradually formed. However, QoS in wireless LAN is a hot and fresh topic, many papers are come out in year 1999. Because of this, what we can take as reference is not much, as most of the papers are not very practical. So, although it may not be general enough or not very grand and great, we have thought of the schemes ourselves, and have implemented it. And we have tried our best to avoid overlapping the QoS scheme with wired LAN. What I mean is that, our scheme is suitable for wireless LAN only, and it is mainly for video/audio applications. As our method is not general, some assumptions have been taken, which will be described later. And, actually, there are many areas that can be improved for the wireless environment, but we only focus on a specific aspect.

2.1 What we have thought of

I think we should tell about what we have thought of and the difficulties we meet first. In deed, we have used a few weeks to ponder what we can be done for QoS in wireless LAN. This is not an easy task, at least not as easy as thinking a product. And we must pay attention to the implementability of the scheme.

Decreasing error rate Since, according to many sources, we know that the error rate for wireless LAN is greater than that for wired LAN. So it is natural for us to think about decreasing the error rate in sending packets through the air. However, after we have searched for some papers that are talking about that, we found that what we needed to do was to alter the routine in hardware devices, which was impossible for us to do. So, we have abandoned this idea finally.

Improving the hand-off mechanism Roaming involves the communication between two access points. We have tested this on the 9th floor of Ho Sin Hang Engineering Building, and have found that the hand-off mechanism is very slow. And it is some how unacceptable. So, we have thought of improving this. Unfortunately, we have failed to find enough reference for us to do so. And access points must be involved in order to achieve this. But the problem is that, we can't alter anything about the access points, since they are hardware devices.

Using workstations to simulate access points As told above, we have thought of improving the hand-off mechanism, however access points must be involved. So, we have thought of using workstations to simulate access points. The idea is to connect an access point to a workstation. Every traffic must travel via the workstation to and from the access point. That is, using a workstation to simulate the "software" part of the hardware. The problem of this scheme is that the overhead is large, since every traffic must be filtered in the workstation.

We have thought of other schemes, but finally have been abandoned due to some reasons. In the next section, I will talk about our QoS scheme.

2.2 Introduction to our QoS scheme

Our QoS scheme aims at reducing the packet collision probability. The transmission medium of wireless LAN is unguided, so the packets are sent everywhere, and also, packets are come from everywhere. So, it is easy to get packets collided with other packets, at least, it is easier than that in wired LAN.

Our method is simple in speaking, that is reserving bandwidth before sending packets. One would like to reserve the bandwidth will broadcast a "signal" packet, saying that it is going to send something in the near future. When the others receive it, they will suspend sending their packets for a fixed period of time. Within this period, the one who reserved the bandwidth temporarily will send its packets. In this situation, we can ensure that the packets would not collide with others. Here, we have made an assumption, which is that, the environment is "clean". It means that there is no other packet or interference in the area, except the known ones. We have taken another assumption, that is, there is no wireless node entering or leaving the coverage of the access point and the two notebooks, and there should have no hidden nodes. Hidden nodes will be introduced later in the report. This method is suitable for video/audio streaming, since the packets are sent periodically, so we can predict the pattern of sending packets, and reserve the bandwidth accordingly.

The scenario of our system consists of at least two notebooks equipped with wireless devices, an access point connected to a wired network, where a server resides in. The system is in a closed area such that the environment is "clean" as to fulfill our assumption. Also, the access points and the two notebooks are within the coverages of each others.

Although the idea is simple in speaking, many prior knowledges must be known first before we can do something on this. Actually, this is the most time consuming task, that is the preparing state. In the following sections, those knowledges required will be introduced one by one.

2.3 IEEE 802.11

According to [10], IEEE 802.11 is gradually gaining its popularity. Wireless networks ranging from infrared within a single building to global network using low-orbit satellites. What we concern in our project is wireless local area network.

IEEE 802.11 was designed for three different media, two based on spread-spectrum radio, one based on infrared. All three has bandwidth between 1 to 2 Mbps. Spread-spectrum was designed to minimize the impact of interference from other devices. One common technique is frequency hopping. Signal is transmitted over a random sequence of frequencies. The sequence of jump is computed algorithmically (both at the sender and at the receiver). One advantage of frequency hopping is that signal appears as noise to any receiver that does not know the hopping sequence. Another common technique is direct sequence.

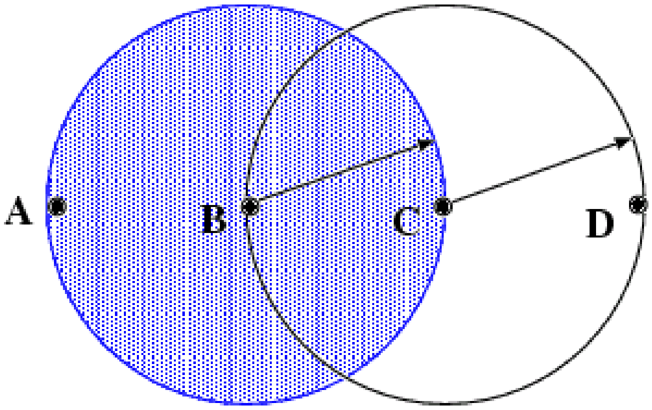

Figure 2.1: Example of Wireless Network.

To a first approximation, IEEE 802.11 operates just like Ethernet, wait until idle and backoff when a collision occurs. In reality, it is more complicated because nodes are not always within reach of each other.

In figure 2.1, B can exchange message with A and C but not D. C can reach B and D but not A. If both A and C want to send packet to B, A and C are unaware of the collision that will happen in B. A and C are said to be hidden nodes with respect to each other. Under the exposed node problem. If B sends a packet to A, it will be wrong for C to assume that it cannot transmit the information to D.

IEEE 802.11 addresses these two problems with an algorithm called Multiple Access with Collision Avoidance (MACA) or called Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA), it is different from the traditional one, which is called Carrier Sense Multiple Access/Collision Detection (CSMA/CD) and used in Ethernet. Before data transmission, sender and receiver need to exchange control frames with each other. This allows nearby nodes to know a data exchange is able to begin.

The protocol is as follow:

1. Sender sends request to send (RTS). The RTS frame includes a field to indicate how long the sender wants to hold the medium.

2. Receiver replies with a clear to send (CTS) to the sender.

3. Any node that sees CTS knows that it is closed to the receiver and therefore cannot transmit for the period specified in the CTS.

4. Any node that sees RTS but not CTS is not closed enough to the receiver to interfere with it, so it is free to transmit.

5. The receiver needs to send an ACK to the sender after successful receiving a frame. All nodes must wait for this ACK before trying to transmit.

6. Should two or more node detect an idle link and try to transmit a RTS at the same time, their RTS will collide. No support for collision detection, but sender knows of collision when there is no CTS after a period of time. Then it uses exponential backoff to delay from some random time and try again.

You may think that it is the same as our QoS scheme, and it has already been implemented. However, our scheme is just a supplement of the protocol described above. Any node send a RTS first before sending its packets, however this RTS is competing with other RTS's. So the node, that is the one plays video application in our system, cannot be guaranteed to reserve its bandwidth for the video stream. Our scheme just backups this by broadcasting a "signal" packet first. So the sender is not involved. And competition between sending RTS is reduced. it is still possible though. The difference is that, our "signal" packet is sent during no packet is required by the node, so, just if it cannot reserve at once, it still has chance. However, for only the protocol above, the node may miss its packets or the delay is too long.

2.4 Linux Wireless

According to [11], wireless LAN is a networking technology allowing the connection of computers without any wires and cables (apart from the mains, mostly using radio technology (and sometimes infrared). It's called LAN because the range targeted is small (with an office, a building, a store, a small campus, a house and etc.). This technology is slowly maturing, and despite a general lack of interest, Linux is able to take advantage of some of the wireless networks available.

We have used two brand of wireless devices in our project, namely Proxim RangeLAN2 and Lucent WaveLAN IEEE. There is information about these two for Linux users. It is kindly provided by [11].

Proxim RangeLAN2 (and some others)

Driver status:	stable
Version:	1.5.1
Where:	http://www.komacke.com/distribution.html
Creator:	Paul Chinn <loomer@1000klub.com>
Maintainer:	Dave Koberstein <davek@komacke.com>
Documentation:	Readme file
Configuration:	specific tool, partial implementation of Wireless Extensions
Statistics:	none
Multi-devices:	no ("insmod -o" multiple modules)
Other features:	Uses Proxim source code
Non implemented:	-
Bugs:	-
License:	Binary only for the core from Proxim + OpenSource Linux wrapper
Vendor web pages:	http://www.proxim.com/ http://www.wlif.com/ http://www.networks.digital.com/wireless/

According to [11], the RangeLAN2 is a classical product using the 2.4 GHz band, made by Proxim, a small Californian company. The products are certified and sold in approximately 50 countries. The RangeLAN2 is based on Proxim proprietary protocol, OpenAir, that Proxim is trying to push as an alternative to 802.11. Of course, many OEM version can be found (like the DEC and AMP versions). It comes as ISA cards, PCMCIA cards, design-in modules, and access point.

The RangeLAN2 implements a specific MAC protocol designed for radio (OpenAir, another pre 802.11) implemented on a generic microcontroller. It uses a 4 bits domain, 4 bits channel and 4 bits subchannel, and also a station type (primary, master, secondary, slave - this is used for network synchronization). There is no encryption, instead it uses a technique called Security ID (which is a simple password used to derive the network ID). The OpenAir protocol is heavily based on RTS/CTS, offer a good robustness but some overhead. It offers as well a modulable contention window size, contention free access for the master, packet fragmentation and power saving.

The Modem uses frequency hopping, and 2 levels of modulations (2FSK/4FSK): it runs a 1.6 Mbps signalling rate for good channel condition (short to medium distances) and falls back to 0.8 Mbps otherwise. Note that the Symphony line of product (home networking) offered by Proxim use the MAC protocol of the RangeLAN2 (OpenAir) with a lower cost radio, and the main difference is the software bundle and the price. On the other hand, the Proxim RangeLAN802 line is very different from OpenAir products, using the 802.11 protocol and a different interface, so the Linux driver won't work.

Dave (the maintainer) uses the Proxim driver source code to build a library (distributed as object only), so we should expect a good quality code. Paul (the creator) wrote the part to interface with the Linux Kernel and Dave maintains it. He has written as well a small utility to set the configuration in the driver (through ioctl). The driver supports the Proxim RangeLAN2, the Proxim Symphony, the DEC RoamAbout FH and the AMP Wireless products. The driver supports both ISA PnP and PCMCIA cards. The current driver doesn't support the RangeLAN802 line.

More detail about Proxim RangeLAN2 can be referred to our previous report, where the setup of it in Linux has been described also.

Lucent WaveLAN IEEE (and some others)

Driver status:	fairly stable
Version:	v1.0.1
Where:	http://www.fasta.fh-dortmund.de/users/andy/wvlan/
Maintainer:	Andreas Neuhaus <andy@fasta.fh-dortmund.de>
Documentation:	readme, headers
Configuration:	Wireless Extensions & module parameters
Statistics:	Wireless Extensions
Multi-devices:	yes, but the ISA to PCMCIA bridge must be reconfigured
Other features:	MTU selection, Wireless Extensions, private extensions
Non implemented:	some IEEE 802.11 support (power saving, authentication)
Bugs:	none reported
License:	GPL
Vendor web pages:	http://www.wvlan.com/ http://www.apple.com/airport/ http://www.cabletron.com/wireless/

According to [11], even if it uses the same name, the WaveLAN IEEE product is completely different from the old WaveLAN, and totally incompatible in term of protocol and hardware interface. It is still built by Lucent, and it still operates in the 2.4 GHz ISM band (Direct Sequence), but the new hardware fully supports the IEEE 802.11 protocol and is no longer based on a Ethernet chip. There is only a PCMCIA version (the ISA version uses a ISA to PCMCIA bridge) and the different access points.

The recently announced Apple AirPort is in fact the Lucent WaveLAN IEEE repackaged (the second generation, 802.11 HR compliant). The AirPort card for the iBook is the OEM version of the WaveLAN, with the antennas being pre-integrated in the laptop. Note that the interface doesn't seem to be PCMCIA compatible even is the connector is the same, so this card can't be used in the normal PC-Card slot of other laptops. Cabletron is also selling the WaveLAN IEEE as RoamAbout 802.11.

The WaveLAN appears to the PC as a standard network card and interfaces naturally with the networking stack. The configuration includes only setting the network name (ESSID), the rest is automatic (finding the equivalent BSSID and channel).

As usual for Lucent, the documentation and website are rich.

As with all IEEE 802.11 products, the WaveLAN offers a fully featured MAC protocol, including MAC level acknowledgement, optional RTS/CTS, fragmentation, automatic rate selection, roaming. This seems exhaustive, but is mandatory for IEEE 802.11 compliance. Different versions of the card include different levels of security (bronze is basic, which we are using, silver is with WEP (RC4-40 bits) and gold is with proprietary 128 bit encryption).

The 2.4 GHz modem is an enhanced version of the previous generation, Direct Sequence Spread Spectrum (11 chips encoding), using both 1 and 2 Mbps signalling rate (using effectively 22 MHz of bandwidth), diversity antennas and with 13 different frequencies (depending on the regulations). The modem sensitivity seems especially good.

Lucent also sells a "turbo" version of the card, which in addition to the standard IEEE 802.11 bit rate (1 and 2 Mbps) adds 5 and 10 Mbps bit-rates for shorter range using Lucent proprietary modulations, and so, it is not compatible with 802.11 HR and other 11 Mbps products.

Lucent offers as well the second generation of the WaveLAN IEEE, which is much cheaper and fully compliant with the new 802.11 HR standard, supporting 5.5 and 11 Mbps bit-rate (compatible with other 11 Mbps products). This is the version used in the Apple AirPort.

The driver for Linux is based on Lucent source code, but doesn't contain any of the code necessary for the higher level 802.11 functionality (authentication, encryption and power saving), however the driver fully interoperable with other WaveLAN nodes and access points.

Note that Lucent has released a binary library driver which is more complete than the driver of Andreas (the maintain). However, since we need the source code, so we are using the one provided by Andreas.

2.5 Linux Networks

According to [12], networking and Linux are terms that are almost synonymous. In a very real sense Linux is a product of the Internet or World Wide Web (WWW). Its developers and users use the web to exchange information ideas, code, and Linux itself is often used to support the networking needs of organization.

The TCP/IP protocols were designed to support communications between computers connected to the ARPANET, an American research network funded by the US government. The ARPANET pioneered networking concepts such as packet switching and protocol layering where one protocol uses the services of another. ARPANET was retired in 1988 but its successors have grown even larger. What is now known as the World Wide Web grew from the ARPANET and is itself supported by the TCP/IP protocols. Unix was extensively used on the ARPANET and the first released networking version of Unix was 4.3 BSD. Linux's networking implementation is modeled on 4.3 BSD in that it supports BSD sockets (with some extensions) and the full range of TCP/IP networking. This programming interface was chosen because of its popularity and to help applications be portable between Linux and other Unix platforms.

2.5.1 An Overview of TCP/IP Networking

In an IP network every machine is assigned an IP address, this is a 32 bit number that uniquely identifies the machine. The WWW is a very large, and growing, IP network and every machine that is connected to it has to have a unique IP address assigned to it. IP addresses are represented by four numbers separated by dots, for example, 16.42.0.9. This IP address is actually in two parts, the network address and the host address. The sizes of these parts may vary (there are several classes of IP addresses) but using 16.42.0.9 as an example, the network address would be 16.42 and the host address 0.9. The host address is further subdivided into a subnetwork and a host address. Again, using 16.42.0.9 as an example, the subnetwork address would be 16.42.0 and the host address 16.42.0.9. This subdivision of the IP address allows organizations to subdivide their networks. For example, 16.42 could be the network address of the ACME Computer Company; 16.42.0 would be subnet 0 and 16.42.1 would be subnet 1. These subnets might be in separate buildings, perhaps connected by leased telephone lines or even microwave links. IP addresses are assigned by the network administrator and having IP subnetworks is a good way of distributing the administration of the network. IP subnet administrators are free to allocate IP addresses within their IP subnetworks.

Generally though, IP addresses are somewhat hard to remember. Names are much easier. lyu9903wireless.cse.cuhk.edu.hk is much easier to remember than 137.189.89.94 but there must be some mechanism to convert the network names into an IP address. These names can be statically specified in the /etc/hosts file or Linux can ask a Distributed Name Server (DNS server) to resolve the name for it. In this case the local host must know the IP address of one or more DNS servers and these are specified in /etc/resolv.conf.

Whenever you connect to another machine, say when reading a web page, its IP address is used to exchange data with that machine. This data is contained in IP packets each of which have an IP header containing the IP addresses of the source and destination machine's IP addresses, a checksum and other useful information. The checksum is derived from the data in the IP packet and allows the receiver of IP packets to tell if the IP packet was corrupted during transmission, perhaps by a noisy telephone line. The data transmitted by an application may have been broken down into smaller packets which are easier to handle. The size of the IP data packets varies depending on the connection media; Ethernet packets are generally bigger than PPP packets. The destination host must reassemble the data packets before giving the data to the receiving application. You can see this fragmentation and reassembly of data graphically if you access a web page containing a lot of graphical images via a moderately slow serial link.

Hosts connected to the same IP subnet can send IP packets directly to each other, all other IP packets will be sent to a special host, a gateway. Gateways (or routers) are connected to more than one IP subnet and they will resend IP packets received on one subnet, but destined for another onwards. For example, if subnets 16.42.1.0 and 16.42.0.0 are connected together by a gateway then any packets sent from subnet 0 to subnet 1 would have to be directed to the gateway so that it could route them. The local host builds up routing tables which allow it to route IP packets to the correct machine. For every IP destination there is an entry in the routing tables which tells Linux which host to send IP packets to in order that they reach their destination. These routing tables are dynamic and change over time as applications use the network and as the network topology changes.

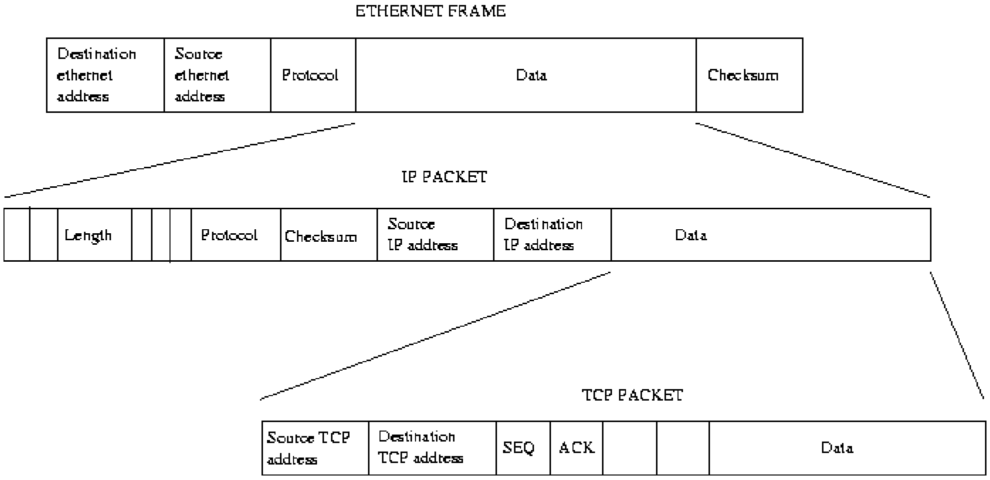

Figure 2.2: TCP/IP Protocol Layers.

The IP protocol is a transport layer that is used by other protocols to carry their data. The Transmission Control Protocol (TCP) is a reliable end to end protocol that uses IP to transmit and receive its own packets. Just as IP packets have their own header, TCP has its own header. TCP is a connection based protocol where two networking applications are connected by a single, virtual connection even though there may be many subnetworks, gateways and routers between them. TCP reliably transmits and receives data between the two applications and guarantees that there will be no lost or duplicated data. When TCP transmits its packet using IP, the data contained within the IP packet is the TCP packet itself. The IP layer on each communicating host is responsible for transmitting and receiving IP packets. User Datagram Protocol (UDP) also uses the IP layer to transport its packets, unlike TCP, UDP is not a reliable protocol but offers a datagram service. This use of IP by other protocols means that when IP packets are received the receiving IP layer must know which upper protocol layer to give the data contained in this IP packet to. To facilitate this every IP packet header has a byte containing a protocol identifier. When TCP asks the IP layer to transmit an IP packet, that IP packet's header states that it contains a TCP packet. The receiving IP layer uses that protocol identifier to decide which layer to pass the received data up to, in this case the TCP layer. When applications communicate via TCP/IP they must specify not only the target's IP address but also the port address of the application. A port address uniquely identifies an application and standard network applications use standard port addresses; for example, web servers use port 80. These registered port addresses can be seen in /etc/services.

This layering of protocols does not stop with TCP, UDP and IP. The IP protocol layer itself uses many different physical media to transport IP packets to other IP hosts. These media may themselves add their own protocol headers. One such example is the Ethernet layer, but PPP and SLIP are others. An Ethernet network allows many hosts to be simultaneously connected to a single physical cable. Every transmitted Ethernet frame can be seen by all connected hosts and so every Ethernet device has a unique address. Any Ethernet frame transmitted to that address will be received by the addressed host but ignored by all the other hosts connected to the network. These unique addresses are built into each Ethernet device when they are manufactured and it is usually kept in an SROM on the Ethernet card. Ethernet addresses are 6 bytes long, an example would be 08-00-2b-00-49-A4. Some Ethernet addresses are reserved for multicast purposes and Ethernet frames sent with these destination addresses will be received by all hosts on the network. As Ethernet frames can carry many different protocols (as data) they, like IP packets, contain a protocol identifier in their headers. This allows the Ethernet layer to correctly receive IP packets and to pass them onto the IP layer.

In order to send an IP packet via a multi-connection protocol such as Ethernet, the IP layer must find the Ethernet address of the IP host. This is because IP addresses are simply an addressing concept, the Ethernet devices themselves have their own physical addresses. IP addresses on the other hand can be assigned and reassigned by network administrators at will but the network hardware responds only to Ethernet frames with its own physical address or to special multicast addresses which all machines must receive. Linux uses the Address Resolution Protocol (or ARP) to allow machines to translate IP addresses into real hardware addresses such as Ethernet addresses. A host wishing to know the hardware address associated with an IP address sends an ARP request packet containing the IP address that it wishes translating to all nodes on the network by sending it to a multicast address. The target host that owns the IP address, responds with an ARP reply that contains its physical hardware address. ARP is not just restricted to Ethernet devices, it can resolve IP addresses for other physical media, for example FDDI. Those network devices that cannot ARP are marked so that Linux does not attempt to ARP. There is also the reverse function, Reverse ARP or RARP, which translates physical network addresses into IP addresses. This is used by gateways, which respond to ARP requests on behalf of IP addresses that are in the remote network.

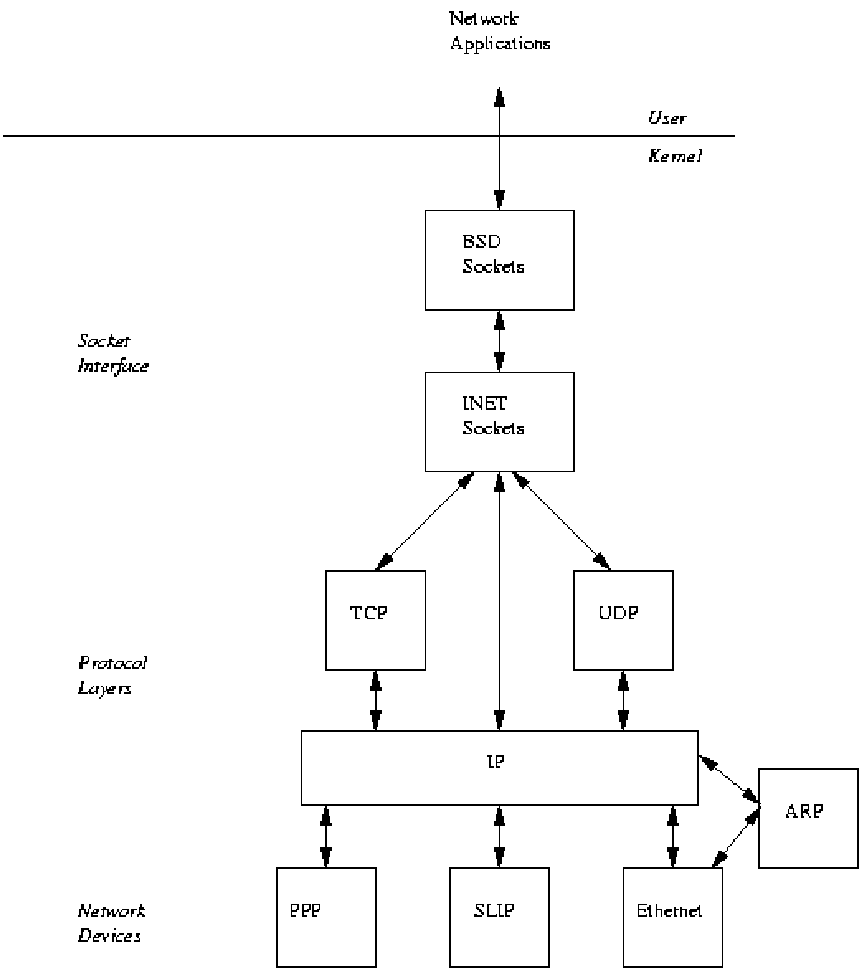

Figure 2.3: Linux Networking Layers.

2.5.2 The Linux TCP/IP Networking Layers

Just like the network protocols themselves, Figure 2.3 shows that Linux implements the internet protocol address family as a series of connected layers of software. BSD sockets are supported by a generic socket management software concerned only with BSD sockets. Supporting this is the INET socket layer, this manages the communication end points for the IP based protocols TCP and UDP. UDP (User Datagram Protocol) is a connectionless protocol whereas TCP (Transmission Control Protocol) is a reliable end to end protocol. When UDP packets are transmitted, Linux neither knows nor cares if they arrive safely at their destination. TCP packets are numbered and both ends of the TCP connection make sure that transmitted data is received correctly. The IP layer contains code implementing the Internet Protocol. This code prepends IP headers to transmitted data and understands how to route incoming IP packets to either the TCP or UDP layers. Underneath the IP layer, supporting all of Linux's networking are the network devices, for example PPP and Ethernet. Network devices do not always represent physical devices; some like the loopback device are purely software devices. Unlike standard Linux devices that are created via the mknod command, network devices appear only if the underlying software has found and initialized them. You will only see /dev/eth0 when you have built a kernel with the appropriate Ethernet device driver in it. The ARP protocol sits between the IP layer and the protocols that support ARPing for addresses.

2.5.3 The BSD Socket Interface

This is a general interface which not only supports various forms of networking but is also an inter-process communications mechanism. A socket describes one end of a communications link, two communicating processes would each have a socket describing their end of the communication link between them. Sockets could be thought of as a special case of pipes but, unlike pipes, sockets have no limit on the amount of data that they can contain. Linux supports several classes of socket and these are known as address families. This is because each class has its own method of addressing its communications. Linux supports the following socket address families or domains:

UNIX	Unix domain sockets
INET	The Internet address family supports communications via TCP/IP protocols
AX25	Amateur radio X25
IPX	Novell IPX
APPLETALK	Appletalk DDP
X25	X25

There are several socket types and these represent the type of service that supports the connection. Not all address families support all types of service. Linux BSD sockets support a number of socket types:

Stream These sockets provide reliable two way sequenced data streams with a guarantee that data cannot be lost, corrupted or duplicated in transit. Stream sockets are supported by the TCP protocol of the Internet (INET) address family.

Datagram These sockets also provide two way data transfer but, unlike stream sockets, there is no guarantee that the messages will arrive. Even if they do arrive there is no guarantee that they will arrive in order or even not be duplicated or corrupted. This type of socket is supported by the UDP protocol of the Internet address family.

Raw This allows processes direct (hence "raw") access to the underlying protocols. It is, for example, possible to open a raw socket to an Ethernet device and see raw IP data traffic.

Reliable Delivered Messages These are very like datagram sockets but the data is guaranteed to arrive.

Sequenced Packets These are like stream sockets except that the data packet sizes are fixed.

Packet This is not a standard BSD socket type, it is a Linux specific extension that allows processes to access packets directly at the device level.

Processes that communicate using sockets use a client server model. A server provides a service and clients make use of that service. One example would be a Web Server, which provides web pages and a web client, or browser, which reads those pages. A server using sockets, first creates a socket and then binds a name to it. The format of this name is dependent on the socket's address family and it is, in effect, the local address of the server. The socket's name or address is specified using the sockaddr data structure. An INET socket would have an IP port address bound to it. The registered port numbers can be seen in /etc/services; for example, the port number for a web server is 80. Having bound an address to the socket, the server then listens for incoming connection requests specifying the bound address. The originator of the request, the client, creates a socket and makes a connection request on it, specifying the target address of the server. For an INET socket the address of the server is its IP address and its port number. These incoming requests must find their way up through the various protocol layers and then wait on the server's listening socket. Once the server has received the incoming request it either accepts or rejects it. If the incoming request is to be accepted, the server must create a new socket to accept it on. Once a socket has been used for listening for incoming connection requests it cannot be used to support a connection. With the connection established both ends are free to send and receive data. Finally, when the connection is no longer needed it can be shutdown. Care is taken to ensure that data packets in transit are correctly dealt with.

The exact meaning of operations on a BSD socket depends on its underlying address family. Setting up TCP/IP connections is very different from setting up an amateur radio X.25 connection. Like the virtual filesystem, Linux abstracts the socket interface with the BSD socket layer being concerned with the BSD socket interface to the application programs which is in turn supported by independent address family specific software. At kernel initialization time, the address families built into the kernel register themselves with the BSD socket interface. Later on, as applications create and use BSD sockets, an association is made between the BSD socket and its supporting address family. This association is made via cross-linking data structures and tables of address family specific support routines. For example there is an address family specific socket creation routine which the BSD socket interface uses when an application creates a new socket.

When the kernel is configured, a number of address families and protocols are built into the protocols vector. Each is represented by its name, for example "INET" and the address of its initialization routine. When the socket interface is initialized at boot time each protocol's initialization routine is called. For the socket address families this results in them registering a set of protocol operations. This is a set of routines, each of which performs a a particular operation specific to that address family. The registered protocol operations are kept in the pops vector, a vector of pointers to proto_ops data structures.

The proto_ops data structure consists of the address family type and a set of pointers to socket operation routines specific to a particular address family. The pops vector is indexed by the address family identifier, for example the Internet address family identifier (AF_INET is 2).

2.5.4 The INET Socket Layer

The INET socket layer supports the internet address family which contains the TCP/IP protocols. As discussed above, these protocols are layered, one protocol using the services of another. Linux's TCP/IP code and data structures reflect this layering. Its interface with the BSD socket layer is through the set of Internet address family socket operations which it registers with the BSD socket layer during network initialization. These are kept in the pops vector along with the other registered address families. The BSD socket layer calls the INET layer socket support routines from the registered INET proto_ops data structure to perform work for it. For example a BSD socket create request that gives the address family as INET will use the underlying INET socket create function. The BSD socket layer passes the socket data structure representing the BSD socket to the INET layer in each of these operations. Rather than clutter the BSD socket with TCP/IP specific information, the INET socket layer uses its own data structure, the sock which it links to the BSD socket data structure. This linkage can be seen in figure 2.4. It links the sock data structure to the BSD socket data structure using the data pointer in the BSD socket. This means that subsequent INET socket calls can easily retrieve the sock data structure. The sock data structure's protocol operations pointer is also set up at creation time and it depends on the protocol requested. If TCP is requested, then the sock data structure's protocol operations pointer will point to the set of TCP protocol operations needed for a TCP connection.

Details can be referred to [12].

2.5.5 The IP Layer

One of the problems of having many layers of network protocols, each one using the services of another, is that each protocol needs to add protocol headers and tails to data as it is transmitted and to remove them as it processes received data. This make passing data buffers between the protocols difficult as each layer needs to find where its particular protocol headers and tails are. One solution is to copy buffers at each layer but that would be inefficient. Instead, Linux uses socket buffers or sk_buffs to pass data between the protocol layers and the network device drivers. sk_buffs contain pointer and length fields that allow each protocol layer to manipulate the application data via standard functions or "methods".

Figure 2.5 shows the sk_buff data structure; each sk_buff has a block of data associated with it. The sk_buff has four data pointers, which are used to manipulate and manage the socket buffer's data:

head points to the start of the data area in memory. This is fixed when the sk_buff and its associated data block is allocated,

data points at the current start of the protocol data. This pointer varies depending on the protocol layer that currently owns the sk_buff,

tail points at the current end of the protocol data. Again, this pointer varies depending on the owning protocol layer,

end points at the end of the data area in memory. This is fixed when the sk_buff is allocated.

There are two length fields len and truesize, which describe the length of the current protocol packet and the total size of the data buffer respectively. The sk_buff handling code provides standard mechanisms for adding and removing protocol headers and tails to the application data. These safely manipulate the data, tail and len fields in the sk_buff:

push This moves the data pointer towards the start of the data area and increments the len field. This is used when adding data or protocol headers to the start of the data to be transmitted,

pull This moves the data pointer away from the start, towards the end of the data area and decrements the len field. This is used when removing data or protocol headers from the start of the data that has been received,

put This moves the tail pointer towards the end of the data area and increments the len field. This is used when adding data or protocol information to the end of the data to be transmitted,

trim This moves the tail pointer towards the start of the data area and decrements the len field. This is used when removing data or protocol tails from the received packet.

The sk_buff data structure also contains pointers that are used as it is stored in doubly linked circular lists of sk_buff's during processing. There are generic sk_buff routines for adding sk_buffs to the front and back of these lists and for removing them.
Details can be referred to [12].

2.6 Linux Module

Linux is a monolithic kernel; that is, it is one, single, large program where all the functional components of the kernel have access to all of its internal data structures and routines. The alternative is to have a micro-kernel structure where the functional pieces of the kernel are broken out into separate units with strict communication mechanisms between them. This makes adding new components into the kernel via the configuration process rather time consuming. Say you wanted to use a SCSI driver for an NCR 810 SCSI and you had not built it into the kernel. You would have to configure and then build a new kernel before you could use the NCR 810. There is an alternative, Linux allows you to dynamically load and unload components of the operating system as you need them. Linux modules are lumps of code that can be dynamically linked into the kernel at any point after the system has booted. They can be unlinked from the kernel and removed when they are no longer needed. Mostly Linux kernel modules are device drivers, pseudo-device drivers such as network drivers, the WaveLAN driver we are using, or file-systems.

You can either load and unload Linux kernel modules explicitly using the insmod and rmmod commands or the kernel itself can demand that the kernel daemon (kerneld) loads and unloads the modules as they are needed.

Dynamically loading code as it is needed is attractive as it keeps the kernel size to a minimum and makes the kernel very flexible. My current Intel kernel uses modules extensively and is only 406Kbytes long. I only occasionally use VFAT file systems and so I build my Linux kernel to automatically load the VFAT file system module as I mount a VFAT partition. When I have unmounted the VFAT partition the system detects that I no longer need the VFAT file system module and removes it from the system. Modules can also be useful for trying out new kernel code without having to rebuild and reboot the kernel every time you try it out. Nothing, though, is for free and there is a slight performance and memory penalty associated with kernel modules. There is a little more code that a loadable module must provide and this and the extra data structures take a little more memory. There is also a level of indirection introduced that makes accesses of kernel resources slightly less efficient for modules.

Once a Linux module has been loaded it is as much a part of the kernel as any normal kernel code. It has the same rights and responsibilities as any kernel code; in other words, Linux kernel modules can crash the kernel just like all kernel code or device drivers can.

So that modules can use the kernel resources that they need, they must be able to find them. Say a module needs to call kmalloc(), the kernel memory allocation routine. At the time that it is built, a module does not know where in memory kmalloc() is, so when the module is loaded, the kernel must fix up all of the module's references to kmalloc() before the module can work. The kernel keeps a list of all of the kernel's resources in the kernel symbol table so that it can resolve references to those resources from the modules as they are loaded. Linux allows module stacking, this is where one module requires the services of another module. For example, the VFAT file system module requires the services of the FAT file system module as the VFAT file system is more or less a set of extensions to the FAT file system. One module requiring services or resources from another module is very similar to the situation where a module requires services and resources from the kernel itself. Only here the required services are in another, previously loaded module. As each module is loaded, the kernel modifies the kernel symbol table, adding to it all of the resources or symbols exported by the newly loaded module. This means that, when the next module is loaded, it has access to the services of the already loaded modules.

When an attempt is made to unload a module, the kernel needs to know that the module is unused and it needs some way of notifying the module that it is about to be unloaded. That way the module will be able to free up any system resources that it has allocated, for example kernel memory or interrupts, before it is removed from the kernel. When the module is unloaded, the kernel removes any symbols that that module exported into the kernel symbol table.

Apart from the ability of a loaded module to crash the operating system by being badly written, it presents another danger. What happens if you load a module built for an earlier or later kernel than the one that you are now running? This may cause a problem if, say, the module makes a call to a kernel routine and supplies the wrong arguments. The kernel can optionally protect against this by making rigorous version checks on the module as it is loaded.

Other information is available at [12].

Examples are always better than thousands of words. Below is an example module given by [13], which will jot down who has opened a file. The technique in altering the system call table has been adopted in our system in order to provide a system call for sending our “signal” packets. Adding new system call in Linux will be introduced later in this report.

```
/* syscall.c
*
* System call “stealing” sample
*/


/* Copyright (C) 1998-99 by Ori Pomerantz */


/* The necessary header files */

/* Standard in kernel modules */
#include <linux/kernel.h> /* We're doing kernel work */
#include <linux/module.h> /* Specifically, a module */

/* Deal with CONFIG_MODVERSIONS */
#if CONFIG_MODVERSIONS==1
#define MODVERSIONS
#include <linux/modversions.h>
#endif

#include <sys/syscall.h> /* The list of system calls */

/* For the current (process) structure, we need
* this to know who the current user is. */
#include <linux/sched.h>




/* In 2.2.3 /usr/include/linux/version.h includes a
* macro for this, but 2.0.35 doesn't - so I add it
* here if necessary. */
#ifndef KERNEL_VERSION
#define KERNEL_VERSION(a,b,c) ((a)*65536+(b)*256+(c))
#endif
```

```
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
#include <asm/uaccess.h>
#endif


/* The system call table (a table of functions). We
* just define this as external, and the kernel will
* fill it up for us when we are insmod'ed
*/
extern void *sys_call_table[];


/* UID we want to spy on - will be filled from the
* command line */
int uid;

#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
MODULE_PARM(uid, "i");
#endif

/* A pointer to the original system call. The reason
* we keep this, rather than call the original function
* (sys_open), is because somebody else might have
* replaced the system call before us. Note that this
* is not 100% safe, because if another module
* replaced sys_open before us, then when we're inserted
* we'll call the function in that module - and it
* might be removed before we are.
*
* Another reason for this is that we can't get sys_open.
* It's a static variable, so it is not exported. */
asmlinkage int (*original_call)(const char *, int, int);



/* For some reason, in 2.2.3 current->uid gave me
* zero, not the real user ID. I tried to find what went
* wrong, but I couldn't do it in a short time, and
* I'm lazy - so I'll just use the system call to get the
* uid, the way a process would.
*
* For some reason, after I recompiled the kernel this
* problem went away.
*/
asmlinkage int (*getuid_call)();



/* The function we'll replace sys_open (the function
```

```
* called when you call the open system call) with. To
* find the exact prototype, with the number and type
* of arguments, we find the original function first
* (it's at fs/open.c).
*
* In theory, this means that we're tied to the
* current version of the kernel. In practice, the
* system calls almost never change (it would wreck havoc
* and require programs to be recompiled, since the system
* calls are the interface between the kernel and the
* processes).
*/
asmlinkage int our_sys_open(const char *filename, int flags, int mode)
{
int i = 0;
char ch;

/* Check if this is the user we're spying on */
if (uid == getuid_call()) {
/* getuid_call is the getuid system call,
* which gives the uid of the user who
* ran the process which called the system
* call we got */

/* Report the file, if relevant */
printk("Opened file by %d: ", uid);
do {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
get_user(ch, filename+i);
#else
ch = get_user(filename+i);
#endif
i++;
printk("%c", ch);
} while (ch != 0);
printk("\n");
}

/* Call the original sys_open - otherwise, we lose
* the ability to open files */
return original_call(filename, flags, mode);
}



/* Initialize the module - replace the system call */
int init_module()
{
/* Warning - too late for it now, but maybe for
* next time... */
```

```
printk("I'm dangerous. I hope you did a ");
printk("sync before you insmod'ed me.\n");
printk("My counterpart, cleanup_module(), is even");
printk("more dangerous. If\n");
printk("you value your file system, it will ");
printk("be `sync; rmmod' \n");
printk("when you remove this module.\n");

/* Keep a pointer to the original function in
* original_call, and then replace the system call
* in the system call table with our_sys_open */
original_call = sys_call_table[__NR_open];
sys_call_table[__NR_open] = our_sys_open;

/* To get the address of the function for system
* call foo, go to sys_call_table[__NR_foo]. */

printk("Spying on UID:%d\n", uid);

/* Get the system call for getuid */
getuid_call = sys_call_table[__NR_getuid];

return 0;
}


/* Cleanup - unregister the appropriate file from /proc */
void cleanup_module()
{
/* Return the system call back to normal */
if (sys_call_table[__NR_open] != our_sys_open) {
printk("Somebody else also played with the ");
printk("open system call\n");
printk("The system may be left in ");
printk("an unstable state.\n");
}

sys_call_table[__NR_open] = original_call;
}
```

2.7 Linux PCMCIA

What we are using is the WaveLAN PCMCIA card, so it is better to have a general knowledge about Linux PCMCIA. However this is not very critical for our project, so if you are interested, please refer to [14].

2.8 Linux System Call

The 386 recognizes two event classes: exceptions and interrupts. Both cause a forced context switch to new a procedure or task. Interrupts can occur at unexpected times during the execution of a program and are used to respond to signals from hardware. Exceptions are caused by the execution of instructions.

Two sources of interrupts are recognized by the 386: Maskable interrupts and Non-maskable interrupts. Two sources of exceptions are recognized by the 386: Processor detected exceptions and programmed exceptions.

Each interrupt or exception has a number, which is referred to by the 386 literature as the vector. The NMI interrupt and the processor detected exceptions have been assigned vectors in the range 0 through 31, inclusive. The vectors for maskable interrupts are determined by the hardware. External interrupt controllers put the vector on the bus during the interrupt-acknowledge cycle. Any vector in the range 32 through 255, inclusive, can be used for maskable interrupts or programmed exceptions. Here is a listing of all the possible interrupts and exceptions:

0	divide error
1	debug exception
2	NMI interrupt
3	Breakpoint
4	INTO-detected Overflow
5	BOUND range exceeded
6	Invalid opcode
7	coprocessor not available
8	double fault
9	coprocessor segment overrun
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	reserved
16	coprocessor error
17-31	reserved
32-255	maskable interrupts

The priority of simultaneous interrupts and exceptions is:

HIGHEST	Faults except debug faults
.	Trap instructions INTO, INT n, INT 3
.	Debug traps for this instruction
.	Debug traps for next instruction
.	NMI interrupt
INTR	interrupt

Under Linux the execution of a system call is invoked by a maskable interrupt or exception class transfer, caused by the instruction int 0x80. We use vector 0x80 to transfer control to the kernel. This interrupt vector is initialized during system startup, along with other important vectors like the system clock vector.

iBCS2 requires an lcall 0,7 instruction, which Linux can send to the iBCS2 compatibility module appropriate if an iBCS2-compliant binary is being executed. In fact, Linux will assume that an iBCS2-compliant binary is being executed if an lcall 0,7 call is executed, and will automatically switch modes.

As of version 0.99.2 of Linux, there are 116 system calls. Documentation for these can be found in the man (2) pages. When a user invokes a system call, execution flow is as follows:

- Each call is vectored through a stub in libc. Each call within the libc library is generally a syscallX() macro, where X is the number of parameters used by the actual routine. Some system calls are more complex then others because of variable length argument lists, but even these complex system calls must use the same entry point: they just have more parameter setup overhead. Examples of a complex system call include open() and ioctl().

- Each syscall macro expands to an assembly routine which sets up the calling stack frame and calls _system_call() through an interrupt, via the instruction int $0x80

  For example, the setuid system call is coded as _syscall1(int,setuid,uid_t,uid); which will expand to:

```
_setuid:
    subl $4,%exp
    pushl %ebx
    movzwl 12(%esp),%eax
    movl %eax,4(%esp)
    movl $23,%eax
    movl 4(%esp),%ebx
    int $0x80
    movl %eax,%edx
    testl %edx,%edx
    jge L2
    negl %edx
    movl %edx,_errno
    movl $-1,%eax
    popl %ebx
    addl $4,%esp
    ret
L2:
    movl %edx,%eax
    popl %ebx
    addl $4,%esp
```

ret

The macro definition for the syscallX() macros can be found in /usr/include/linux/unistd.h, and the user-space system call library code can be found in /usr/src/libc/syscall/

- At this point no system code for the call has been executed. Not until the int $0x80 is executed does the call transfer to the kernel entry point _system_call(). This entry point is the same for all system calls. It is responsible for saving all registers, checking to make sure a valid system call was invoked and then ultimately transferring control to the actual system call code via the offsets in the _sys_call_table. It is also responsible for calling _ret_from_sys_call() when the system call has been completed, but before returning to user space.

  Actual code for system_call entry point can be found in /usr/src/linux/kernel/sys_call.S. Actual code for many of the system calls can be found in /usr/src/linux/kernel/sys.c, and the rest are found elsewhere.

- After the system call has executed, _ret_from_sys_call() is called. It checks to see if the scheduler should be run, and if so, calls it.

- Upon return from the system call, the syscallX() macro code checks for a negative return value, and if there is one, puts a positive copy of the return value in the global variable _errno, so that it can be accessed by code like perror().

The startup_32() code found in /usr/src/linux/boot/head.S starts everything off by calling setup_idt(). This routine sets up an IDT (Interrupt Descriptor Table) with 256 entries. No interrupt entry points are actually loaded by this routine, as that is done only after paging has been enabled and the kernel has been moved to 0xC0000000. An IDT has 256 entries, each 4 bytes long, for a total of 1024 bytes.

When start_kernel() (found in /usr/src/linux/init/main.c) is called it invokes trap_init() (found in /usr/src/linux/kernel/traps.c). trap_init() sets up the IDT via the macro set_trap_gate() (found in /usr/include/asm/system.h). trap_init() initializes the interrupt descriptor table as shown here:

0	divide_error
1	debug
2	nmi
3	int3
4	overflow
5	bounds
6	invalid_op
7	device_not_available
8	double_fault
9	coprocessor_segment_overrun
10	invalid_TSS
11	segment_not_present
12	stack_segment
13	general_protection
14	page_fault
15	reserved
16	coprocessor_error
17	alignment_check
18-48	reserve

At this point the interrupt vector for the system calls is not set up. It is initialized by sched_init() (found in /usr/src/linux/kernel/sched.c). A call to set_system_gate (0x80, &system_call) sets interrupt 0x80 to be a vector to the system_call() entry point.

The following is the procedure to add new system call in Linux:

1. Create a directory under the /usr/src/linux/ directory to hold your code.

2. Put any include files in /usr/include/sys/ and /usr/include/linux/.

3. Add the relocatable module produced by the link of your new kernel code to the ARCHIVES and the subdirectory to the SUBDIRS lines of the top level Makefile. See fs/Makefile, target fs.o for an example.

4. Add a #define __NR_xx to unistd.h to assign a call number for your system call, where xx, the index, is something descriptive relating to your system call. It will be used to set up the vector through sys_call_table to invoke you code.

5. Add an entry point for your system call to the sys_call_table in sys.h. It should match the index (xx) that you assigned in the previous step. The NR_syscalls variable will be recalculated automatically.

6. Modify any kernel code in kernel/fs/mm/, etc. to take into account the environment needed to support your new code.

7. Run make from the top level to produce the new kernel incorporating your new code.

For our QoS scheme, we also need to add new system call in Linux. However, there is a problem. The system call we intended to adding is for calling the WaveLAN driver to send "signal" packet. However, the WaveLAN driver is a module, that means that we can't compile it with the kernel, as it is a module which can be loaded and unloaded dynamically. However, the system call is needed to be resided in the kernel. That is, our new system call is not part of the kernel! We have used a tricky solution to solve this problem.

We have added a new system call following the above procedure. However, the function body is empty except printing out a message requesting the user to insert a WaveLAN/IEEE PCMCIA card. And for the WaveLAN driver, we have modified it so that when it is loaded, it will change the kernel's system call table, as described in the previous section (Linux Module). Therefore, when someone calls the system call, it will call the function in the driver if the driver is loaded while it will print out the request if the driver is unloaded. This is the tricky method we adopt.

2.9 WaveLAN/IEEE Frame Structures

This is the most crucial thing we need for our project. Unfortunately, reference is not enough for us. So, we have used up lots of time in finding out how to send our "signal" packets. And we always need to take the risk with trial and error. In this section, I will introduce the receive and transmit frame structures for WaveLAN/IEEE.

The Receive and Transmit Frame Structures are exchanged between the HCF (please refer to [16]) and the WMAC Controller, and consist of of three segments:

1. Control fields for the WMAC Controller
2. Header information
3. Data

For transmit, the control fields are used by the HCF to control how the frame is handled by the WMAC Controller. The MSF-supplied header information and data are used to build the MAC frame for data transfer.

After reception, the control fields are used by the WMAC Controller to inform the HCF/MSF about frame related status information.

Figure 2.6 shows the formats of these three segments. The light dashed arrows on the right indicate the 802.3 fields. The heavy dashed arrows on the right indicate the Ethernet-II encapsulated fields. The light solid arrows on the right indicate the 802.11 fields.

These structures are accessed by HCF functions of the Data Transfer Group. The gray column of Byte offset valuesin the Frames as seen on the Host-NIC I/F. These offsets are not usable at the WCI I/F. They are only specified to ease debugging on the HCF-level.

The non-gray column of Byte offset values reflect the values corresponding with the current implementation of the HCF.

Successive hcf_put_data() calls (functions provided by HCF source code) will fill sequentially the fields reflected by the arrows in the right border. In case of 802.11, the solid arrow, in case of 802.3, the light dashed arrow and in case of encapsulation, the heavy dashed arrow. A single hcf_get_date() call (also provided by HCF source code) behaves analogous to hcf_put_data(). The important difference is that each hcf_get_data() starts at the specified offset. For example, specifying 0x0000 in case of 802.11 (or more correctly HFS_ADDR_1), starts at absolute offset 0x0012, specifying 0x000 or HFS_DAT_LEN in case of 802.3 or encapsulated frames starts at absolute offset 0x002E, specifying HFS_DAT in case of 802.3 or 802.11 frames starts at absolute offset 0x003C, and in case of encapsulated at absolute offset 0x0042. In case of 802.3, access of the HFS_LEN field yields the length value (absolute offset 0x003A). In case of encapsulated Ethernet-II, access of the HFS_LEN field yields the type value (absolute offset 0x0042). The length and SNAP-header are not accessible.

There are two functions that are very useful for our system, namely hcf_get_data() and hcf_put_data(). Followings are the definitions of these two functions. Details can be referred to [16].

Get data function (hcf_get_data) The purpose of this function is to obtain received message data parts or control information from the WaveLAN PCMCIA card. The prototype of it is *int hcf_get_data(IFBP ibp, int offset, wci_bufp *datbuf, int datlen);* The return value can be HCF_SUCCESS (success) or HCF_ERR_NO_NIC (card removed during data retrieval).

ifbp	Value indicating the address of I/F Block.
offset	Value specifying the beginning of the data part or the control information to be obtained.
datbuf	Pointer to the Data Buffer location.
datlen	Value specifying the number of message bytes to be obtained.
Return Values	Enumerated value indicating the Read Status.

Detail description of this function can be referred to [16].

Put data function (hcf_put_data) The purpose of this function is to transfer transmit message data parts or control information to the WaveLAN PCMCIA Card. The prototype of it is *void hcf_put_data(IFBP ifbp, wci_bufp *datbuf, int datlen, hcf_16 port);* There is no return value for this function.

ifbp	Value indicating the address of I/F Block.
datbuf	Pointer to the Data Buffer location.
datlen	Value specifying the number of message bytes to be transferred.
port	Number indicating the MAC port. MAC Port 0 => HCF_PORT_0 ============= HCF only section (start) ========== MAC Port 1 => HCF_PORT_1 MAC Port 2 => HCF_PORT_2 MAC Port 3 => HCF_PORT_3 MAC Port 4 => HCF_PORT_4 MAC Port 5 => HCF_PORT_5 MAC Port 6 => HCF_PORT_6 Discard preceding data => HCF_PUT_DATA_RESET ============= HCF only section (end) ===========
Return Values	N/A

Detail description of this function can be referred to [16].

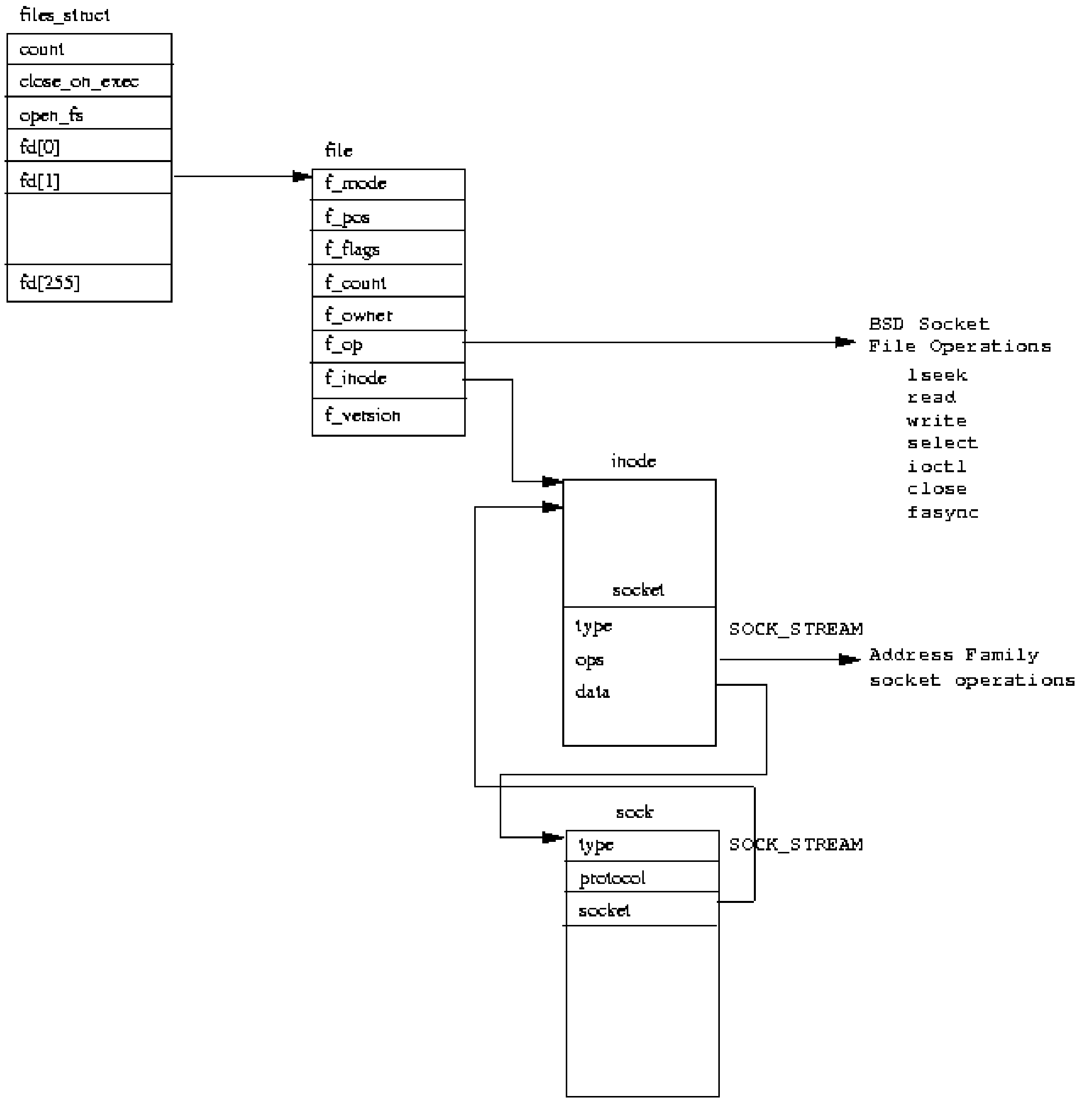

Figure 2.4: Linux BSD Socket Data Structures.

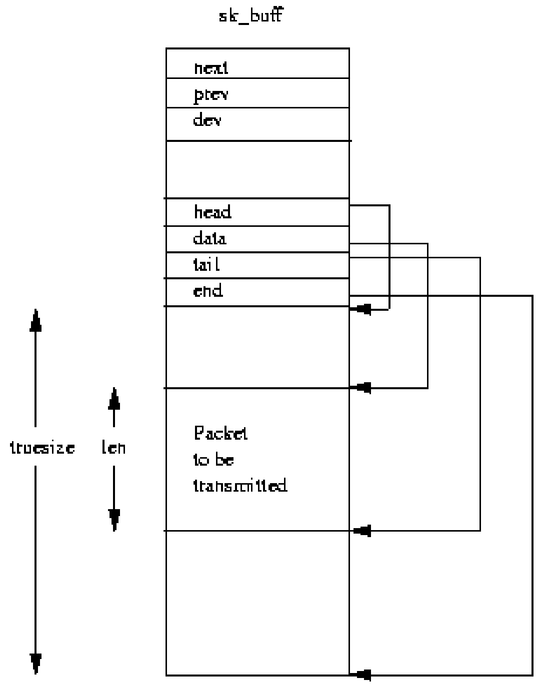

Figure 2.5: The Socket Buffer (sk_buff).

Byte offset		HCF Mnemonic	Transmit Frame Structure	Structure segment	Receive Frame Structure	
00	−2E	HFS_STAT	Status	**Control**	Status	
02	−2C		0	**Fields**		
			0	Common for		
06	−28	HFS_Q_INFO	SwSupport	802.11	Signal / Silence	
				and	Rate	
0A	−24		0	802.3		
0C	−22	HFS_TX_CNTL	TxControl			
0E	−20	HFS_FRAME_CNTL	FrameControl	**Header Info**	FrameControl	
10	−1E	HFS_ID	Duration/ID		Duration/ID	
12	00	HFS_ADDR_1	Address1	802.11 Format	Address1	
18	06	HFS_ADDR_2	Address2		Address2	
1E	0C	HFS_ADDR_3	Address3		Address3	
24	12	HFS_SEQ_CNTL	SequenceControl		SequenceControl	
26	14	HFS_ADDR_4	Address4		Address4	
2C	1A	HFS_DAT_LEN	DataLen **(Little Endian)**		DataLen **(Little Endian)**	
2E	00	HFS_ADDR_DEST	DestAddr	**Header Info**	DestAddr	
				802.3		
34	06	HFS_ADDR_SRC	SrcAddr	Format	SrcAddr	
3A	0C	HFS_LEN	Length **(Big Endian)**		Length **(Big Endian)**	(ref)
3C	0E	HFS_DAT	Data / SNAP header	**Data Info**	Data / SNAP header	
3E			Data / SNAP header		Data / SNAP header	
40			Data / SNAP header		Data / SNAP header	
42		HFS_TYPE	Data / E-II Type		Data / E-II Type	(ref)
				Common for 802.11 and 802.3		
			(Max. 2304 bytes)		(Max. 2304 bytes)	

Figure 2.6: Frame structures formats.

Chapter 3

Video Application

We would like to implement a real time video player that can get file through both wired and wireless networks with QoS enabled. One of the reasons of implementing real time video player is that the demand of multimedia access is increasing rapidly, it is mentioned earlier. Another reason is that real time audio needs only about 150 Kbps bandwidth, which is not enough to see the performance of multimedia on wireless LANs with 2 Mbps throughputs. In order to show the effect of performance of multimedia on wireless LANs and show the significance of using QoS, implementing real time video is a good choice. We have choose MPEG-1 video format as our target video format, since MPEG-1 video requires a bandwidth of about 1.3 Mbps, which can be compared to 2 Mbps throughput of wireless LANs.

3.1 Implementation of the Video Player (for the last semester)

The source of our video player is from mpeg_play, which is implemented by Berkeley Multimedia Research Center. We have hacked the source code in order to enable real time access of video files through the network. The idea of this implementation is using RPC, we have provided some services for the client to handle the video files via network. The services are:

fopen Provided for client to open a file.
fclose Provided for client to close an opened file.
fread Provided for client to retrieve data from an opened file.
fseek Provided for client to set the position of the file pointer of an opened file.
ftell Provided for client to get the current position of the file pointer of an opened file.
fgetc Provided for client to get a character from an opened file.
rewind Provided for client to reset the position of the file pointer to the start position.

By setting the environment variable LYU_MPEG_SVR to the address of the server, the modified mpeg_play will automatically retrieve the file from the server specified. When the environment variable is not set, it will retrieve the file as normal from the local host. In last semester, the modified mpeg_play could work properly in both local and network (wired and wireless) mode. However, QoS was not currently supported. In order to enable

QoS in our video application, there were still lots needed to be done.

However, we find that this version of video streaming is not good. The video cannot be played smoothly. So we have abandoned this version of video player, and implement a new one for a better result, and have made it QoS enabled. I am going to describe it, however, details should better be referred to my partner's report.

3.2 Implementation of the Video Player (for this semester)

Figure 3.1 shows the overall design of the real-time video application.

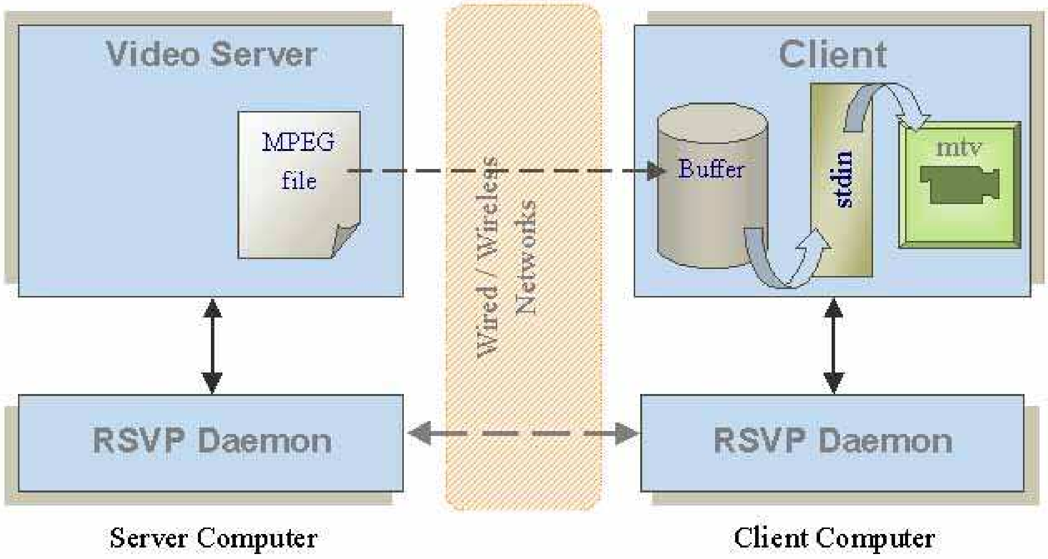

Figure 3.1: Overall Design of Real-time Video Application

In the video application, we use MPEG-1 as the video format. MPEG-1 video needs bandwidth of about 1.2Mbps. The decoder used in decoding MPEG video is mtv[17]. We choose mtv as the decoder because it can play video from stdin, which is easier for the implementation of real-time player.

We have implemented a client/server model to handle the transmission of packets through the networks. In higher view, the server will send the video file to client, which can be a wired client or wireless node, by using TCP connection. Then, the client will store the received packets into a circular buffer. On the other hand, it will put the packets into stdout, and through the pipe, the mtv player will consume the packets from stdin. By using this design, we can store the video file in server, and play the video through the networks in real-time. In our implementation, we must use TCP connection to ensure the reliable connection. It is because in wireless LANs, the error rate is high, and so it is easy to lose packets, but the decoding of MPEG file cannot has lost packets. To ensure all video packets can be transmitted to the client, TCP connection is needed.

The video application is QoS embedded. That is, by using the qos functions in server side, we can control the QoS level of the applications. The QoS used in real-time video application is RSVP and traffic control (packet scheduling). RSVP is for reservation of bandwidth. By using RAPI (API for RSVP), we can control the RSVP parameters. The traffic control is to control the priority of packets sent through the networks. Details of RAPI and traffic control will be discussed in following sections.

3.3 Running Video Application

To run the video application with QoS enabled, traffic control also needed. But this will be talked in later chapter. You can now just assume that when a command is run, the QoS can be enabled.

Server Side

Before running any application, RSVP should be set in both server side and client side. You can run the following command in root to open RSVP:

> rsvpd -R <client_address>

Rsvpd is a daemon program that uses the RSVP resource reservation protocol to set up reservation state in hosts and routers. It supports an API that allows applications to make reservation requests. It includes an adaptation module to the admission control and traffic control mechanisms provided by the output device drivers in the kernel.

<client_address> is the IP address of the client host. It can be a hostname or a dotted-decimal number. -R parameter will make rsvpd sending a unicast copy of any UDP-encapsulated RSVP message directly to <client_address>, in addition to copies it normally sends. It ensure that the RSVP message can send to client.

Other than -R parameter, there are other parameter can be set, depends on the needs of users. Here is the usage of rsvpd:

rsvpd [-D] [-d debug_bits] [-l logging_level] [-R router_addr] [-t mcast_TTL]

-D Execute rsvpd in debugging mode.

-d Set the debugging mask to the integer debug_bits.

-l Set the logging level, which controls logging of data for debugging and diagnosis, to the integer logging_level.

-t Set the IP TTL value for multicasting diagnostic information (summary of the state) to mcast_TTL. This information is multicast when the DEBUG_MCAST_STATE bit is set in the debugging bit mask.

After running rsvpd, user can run the video server as follows:

> ./TCPsendto <port>

<port> is the port number.

To control the quality of service, user in the server side should login as root and then set the following commands:
To enable QoS:

> ./qos.wireless

To disable QoS:

> ./noqos.wireless

To delete all traffic control, you can type:

> ./del.all

To show the classification of traffic, you can type:

> ./tc.show

qos.wireless, noqos.wireless, del.all, and tc.show are the scripts that access the parameters of traffic control. In qos.wireless and noqos.wireless, the limited bandwidth is set to 2Mbits. To limited bandwidth can be changed, in wired network, it should be set lower in order to see the effect of QoS.

In file noqos.wireless, there is no priority of any type of services. That is any applications in client requesting for the packets from the same server will compete for the bandwidth. However, in file qos.wireless, there is a priority for real-time services. We have set our real-time video application having higher priority by setting the "Type of Service" (ToS). Therefore, the video application can get more bandwidth in the transmission of packets.

The details of these scripts and traffic control is discussed in Integration Chapter.

Client Side

Same as server side, the client should first open RSVP by the following command:

> rsvpd -R <server_address>

After that, if the server has already run the video server, then the client can run the video application by the following command:

> ./TCPrecv <server_address> <port>

And the client can see the mtv control box pop out. Then, just click the <File> option in the control box, and choose <Read From Stdin> is okay. The video will be play if there is enough packets in the buffer. (It is usually start when the buffer is 10% full.)

Chapter 4

Integration

This part is written together with my partner, as this is the common work we have done.

In this chapter, we will attempt to tell you about our idea of integrating the QoS scheme we have thought of for the wireless LAN and the QoS scheme that LYU9802 has used for the wired LAN. The traffic control in Linux will also be introduced too.

4.1 Integration of the whole system

Figure 4.1: Integration of the whole system.

Figure 4.1 shows the overall idea of our system. There are two notebooks, one being the video streaming client and the other one being the traffic source, that is, it will provide the traffic which will congest the network. The server is resided in a wired network, which will provide the video stream for the client. Access point is involved for the communication between the client, which is resided in the wireless LAN and the server, which is resided in the wired LAN. We assume that there is no other traffic across the air, and there is no wireless node leaving or entering the coverage of the access point. Moreover, the two notebooks and the access point are with the coverages of each others. LYU9802's QoS scheme will be applied for the wired part, that is from the server to the access point (actually, it is up to the client also!) while our QoS scheme will be applied for the wireless part (precisely, it is for the two notebooks).

At the very start, both the client and the server should run the RSVP daemon as introduced in the previous section in order to establish a RSVP path from the client to the server and vice versa. Then the server should start up in order to wait for the client to request video streams. After then, the client can ask the server for the video it wants. At the same time other traffic will be generated, either from the client itself or from the other notebook. Now, if no QoS scheme is applied, the client cannot see the video smoothly, and would be annoyed.

At this time, our QoS schemes are useful. First of all, for the path from the server to the client, RSVP has reserved the bandwidth for us, and the traffic control and packet schedule algorithm, that is CBQ, will give higher priority for the video stream such that the stream can reach the client punctually. And other traffic will be blocked up to some extent. However this is not enough, since the bandwidth in wireless LAN is not reliable and the collision rate is high. So, when there is no traffic for the client, it will broadcast a "signal" packet to tell the others, that is the other notebook, that it is going to receive packets soon, asking others to suspend its traffic for a while. After the other nodes receive that packet, they will be silent for a while, and the environment is cleaned up. Now, the stream from the server can reach the client safely. And the mechanism above is running over and over again, until the video is end.

The video application, RSVP, CBQ, our QoS scheme and etc. have been introduced in the previous chapters, however, one of the most important issues has not been mentioned yet, it is the traffic control in Linux, available from the iproute2+tc package. Now, we are going to introduce it.

4.2 Traffic Control

There is a package, iproute2+tc, provided for Linux to control the traffic in Linux. The installation can be referred to our previous report and [3].

According to [3], there are some problems comes along with the iproute2+tc package that the writer of it does not know. The package does not work or gives wrong result in some situation if the bugs are not fixed. In order to fix the bug, first of all, one should edit the file q_prio.c under the directory iproute2/tc/ and change the codes

```
if (pmap_mode) {
 for (idx=0; idx < TC_PRIO_MAX; idx++)
  opt.priomap[idx] = opt.priomap[TC_PRIO_BESTEFFORT];
```

```
}
```

to

```
if (pmap_mode) {
 for (; idx <=TC_PRIO_MAX; idx++)
  opt.priomap[idx] = opt.priomap[TC_PRIO_BESTEFFORT];
} else
 for (idx=0; idx <=TC_PRIO_MAX; idx++)
  opt.priomap[idx] = opt.priomap[TC_PRIO_BESTEFFORT];
```

After changing the codes, one should recompile the package following the procedure given by the package itself. It order to ensure it has compiled, one should first of all, clean up the object files such that the compiler will compile the package from scratch.

Setup of a CBQ queue can be done by tc, which comes with the iproute2+tc package. However, the package is lack of documentation, the usage and the effect of the scripts are not clear. We have tested it with trial and error, and have read the references written by [3]. Now, we can have a little idea of what it is doing about, and have written two scripts for our QoS system. One of the scripts is to restrict the bandwidth to about 2Mbps without any QoS parameter, and the other one is to restrict the bandwidth to about 2Mbps too but with QoS applied for the video application we have implemented. The following script is the one given by the original iproute2+tc package, which is enclosed in the directory iproute2+tc/example with name cbqinit.eth1.

1. #! /bin/sh
   (Identification of shell script)

2. TC=/usr/local/src/iproute2/iproute2/tc/tc

3. IP=/usr/local/src/iproute2/iproute2/ip/ip
   (Define path of the package)

4. $TC qdisc add root dev eth0 handle 1: cbq bandwidth 10Mbit cell 8 avpkt 1000 mpu 64
   (Setup a queue using CBQ at root)

5. $TC class add dev eth0 parent 1:0 classid :1 est 1sec 8sec cbq bandwidth 10Mbit rate 10Mbit allot 1514 cell 8 weight 1Mbit prio 8 maxburst 50 avpkt 1000
   (Setup a CBQ class at root)

6. $TC class add dev eth0 parent 1:1 classid :2 est 1sec 8sec cbq bandwidth 10Mbit rate 4Mbit allot 1514 cell 8 weight 500Kbit prio 6 maxburst 50

avpkt 1000 split 1:0 defmap ff3d
(Setup a CBQ class under root occupying the full bandwidth but will 4Mbps rate of transfer only)

7. $TC qdisc add dev eth0 parent 1:2 sfq quantum 1514b perturb 15
(Setup a SFQ queue for the class defined in line 6)

8. $TC class add dev eth0 parent 1:1 classid :3 est 2sec 16sec cbq bandwidth 10Mbit rate 1Mbit allot 1514 cell 8 weight 100Kbit prio 2 maxburst 100 avpkt 1000 split 1:0 defmap c0
(Setup a CBQ class under the class defined in line 6 with rate of transfer 1Mbps and full bandwidth)

9. $TC qdisc add dev eth0 parent 1:3 sfq quantum 1514b perturb 15
(Setup a SFQ queue for the class defined in line 8)

10. $TC class add dev eth0 parent 1:1 classid :4 est 1sec 8sec cbq bandwidth 10Mbit rate 100Kbit allot 1514 cell 8 weight 10Mbit prio 8 maxburst 10 avpkt 1000 split 1:0 defmap 2
(Setup a CBQ class under the class defined in line 9 with rate of transfer 100Kbps and full bandwidth, this is a class defined for interactive application.)

11. $TC qdisc add dev eth0 parent 1:4 sfq quantum 1514b perturb 15
(Setup a SFQ queue for the class defined in line 10)

12. $TC filter add dev eth0 parent 1:0 protocol ip priority 100 rsvp
(Setup a packet classifier with RSVP as its protocol at the root)

13. $IP route add 137.189.230.44 dev eth0 proto static flowid 1:4
(Add a route entry to route table)

14. $TC class add dev eth0 parent 1:1 classid 1:7FFE cbq rate 5Mbit bandwidth 10Mbit allot 1514b avpkt 1000 weight 50Kbit prio 8 maxburst 20 cell 8
(Setup a CBQ class under the class defined in line 6 for real time traffic)

15. $TC class add dev eth0 parent 1:7FFE classid 1:7FFF est 4sec 32sec cbq rate 1Mbit bandwidth 10Mbit allot 1514b avpkt 1000 weight 10Kbit prio 6 maxburst 10 cell 8 split 1:7FFE defmap ffff
(Setup a CBQ class under the class defined in line 14 to redefine the real time traffic)

The explanation of the script above is given by [3].

Now, we are going to describe our scripts, which are aimed to control the traffic in our system. In order to make it narrower such that the effect of congestion can be easier demonstrated, both of the scripts restrict the bandwidths to about 2 Mbps, it is because the bandwidth for the wired LAN (Ethernet) is 10 Mbps.

noqos.wireless

```
#! /bin/sh

TC=tc
IP=ip
DEVICE=eth0
BW="10Mbit"
LIMIT="2Mbit"
REMAIN="8Mbit"

#Ethernet
$TC qdisc \
add dev $DEVICE root handle 1: \
cbq bandwidth $BW avpkt 1000 mpu 64

#Others
$TC class \
add dev $DEVICE parent 1: classid :1 est 1msec 8msec \
cbq bandwidth $BW rate $REMAIN allot 1514 maxburst 50 avpkt 1000 \
bounded isolated prio 1 split 1:0 defmap 0xffaa

#FTP and Video Stream
$TC class \
add dev $DEVICE parent 1: classid :4 est 1msec 8msec \
cbq bandwidth $LIMIT rate $LIMIT allot 1514 maxburst 50 avpkt 1000 \
bounded isolated prio 7 split 1:0 defmap 0x0001
```

qos.wireless

```
#! /bin/sh

TC=tc
IP=ip
DEVICE=eth0
BW="10Mbit"
LIMIT="2Mbit"
REMAIN="8Mbit"
VIDEO="2Mbit"
FTP="1Mbit"
```

```
#Ethernet
$TC qdisc \
add dev $DEVICE root handle 1: \
cbq bandwidth $BW avpkt 1000 mpu 64

#Others
$TC class \
add dev $DEVICE parent 1: classid :1 est 1sec 8sec \
cbq bandwidth $BW rate $REMAIN allot 1514 maxburst 50 avpkt 1000 \
bounded isolated prio 1 split 1:0 defmap 0xffaa

#IEEE 802.11
$TC class \
add dev $DEVICE parent 1: classid :2 est 1sec 8sec \
cbq bandwidth $LIMIT rate $LIMIT allot 1514 maxburst 50 avpkt 1000 \
bounded isolated prio 4 split 1:0 defmap 0x0005

#IEEE 802.11 (to be divided, declare as parent node)
$TC qdisc \
add dev $DEVICE parent 1:2 handle 4: \
cbq bandwidth $LIMIT avpkt 1000 mpu 64

#FTP
$TC class \
add dev $DEVICE parent 4: classid :3 est 1sec 8sec \
cbq bandwidth $FTP rate $FTP weight $FTP \
allot 1514 maxburst 50 avpkt 1000 \
bounded isolated prio 8 split 4:0 defmap 0x0004

#Video Stream
$TC class \
add dev $DEVICE parent 4: classid :4 est 1sec 8sec \
cbq bandwidth $VIDEO rate $VIDEO weight $VIDEO \
allot 1514 maxburst 50 avpkt 1000 \
bounded isolated prio 1 split 4:0 defmap 0x0001
```

Figure 4.2 shows the link structure used in noqos.wireless. The bandwidth we have is 10M, we must narrow it down in order to show the congestion. So, the 10M link is divided into two half, one with 8M and the other one is 2M. Now, the 2M bandwidth is shared by the FTP application and the video player. 2M is chosen since the bandwidth for the WaveLAN/IEEE is about 2M. They will compete for bandwidth with each other. The result for the competition has been plotted as graph. You can see the graph in the next chapter. The "defmap" field in the script is a mask for the TOS field of the applications. The value for FTP is 5, while that for our video application is 1. So after doing a logical AND, that is 1 AND 0x0001 and 5 AND 0x0001, the result is 0x0001 for both case, therefore they will fall into the node with class id 4, as shown figure 4.2.

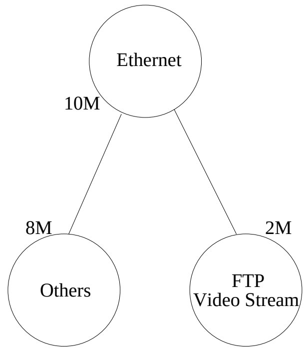

Figure 4.2: Link structure used in noqos.wireless.

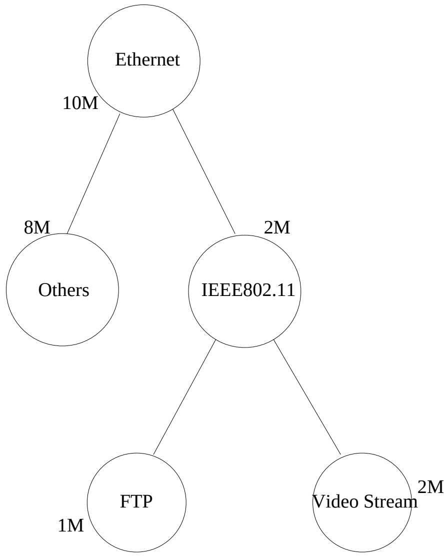

Figure 4.3: Link structure used in qos.wireless.

Figure 4.3 shows the link structure used in qos.wireless. It is based on the one in noqos.wireless. However, the IEEE 802.11 node is further subdivided so that the FTP application and video stream are treated separately. FTP application will be assigned 1M bandwidth while video stream will be assigned 2M bandwidth. You will wonder why the bandwidth is larger when summed up, that is 1M plus 2M is more than the parent node's 2M. It is because the priority is higher for the video stream to get the bandwidth (prio 1 and prio 8 in the script). So, when there is an application requesting video stream, it can get the full 2M while the FTP application will get the remaining, which is less than 1M. However, when there is no video stream sending, the FTP application can get 1M bandwidth for its transfer.

[3] has given a more detail explanation of iproute2+tc, you can refer to it if you are interested. Although they have tried out the package (which has nearly no documentation) and given some documentations, we still need to try ourselves to guess the meaning of the commands. This can be said as one of the difficult parts in our project.

Chapter 5

Performance Evaluation

5.1 Introduction of the Experiment

We have done the performance evaluation on the real-time video application. As time is limited, we have only done the performance evaluation in wireless networks. Since wireless networks has narrower bandwidth, it is much more important to have Quality of Services in wireless networks, and thus it is much more meaningful to demonstrate and evaluate the performance of real-time video application in wireless networks.

The evaluation procedure is similar to the one that was done in the last semester. The scenario of the experiment is shown in figure 5.1.

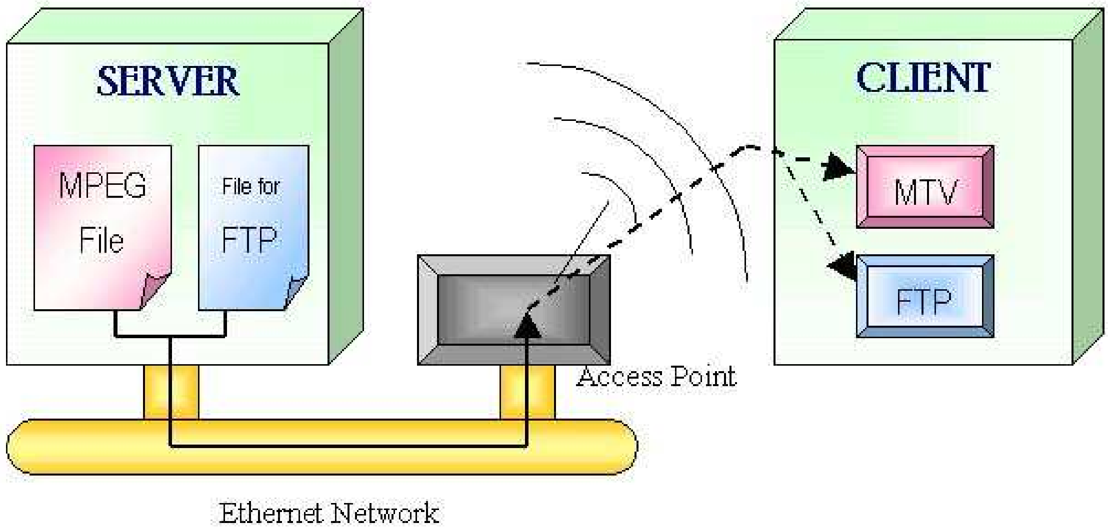

Figure 5.1: Scenario of the Experiment

In the experiment, we have a server which is resided in the wired network. And the wired network is connected to the access point. The client, which is a wireless node, is connected to the server through the access point. We used WaveLAN wireless devices in this experiment. RSVP is used to reserve bandwidth through the path from server to client. And traffic control is done in the server side.

To evaluate the performance, we have run the real-time video application and the FTP application simultaneously. The real-time video application has higher priority than FTP to get more bandwidth. At the start of the experiment, we have run the FTP application. It requested the file transfer from a server. After some time, the real-time video application is run in the client, and it also requested the MPEG video file from the same server. During the experiment, the statistics of both FTP and video applications are record.

5.2 Experiment Results

In the experiment, we set the limit of bandwidth to 2Mbps, which is the bandwidth for WaveLAN Turbo (Bronze) PC card. QoS was disabled in the first measurement and QoS was enabled in the second measurement. At the start of experiment, FTP was run. After 30 seconds, the real-time video player was started to run. For both measurements, the files used for FTP and video player were the same and all gotten from the same server. Figure 5.2 and figure 5.3 shows the current received bandwidth without QoS and with QoS respectively.

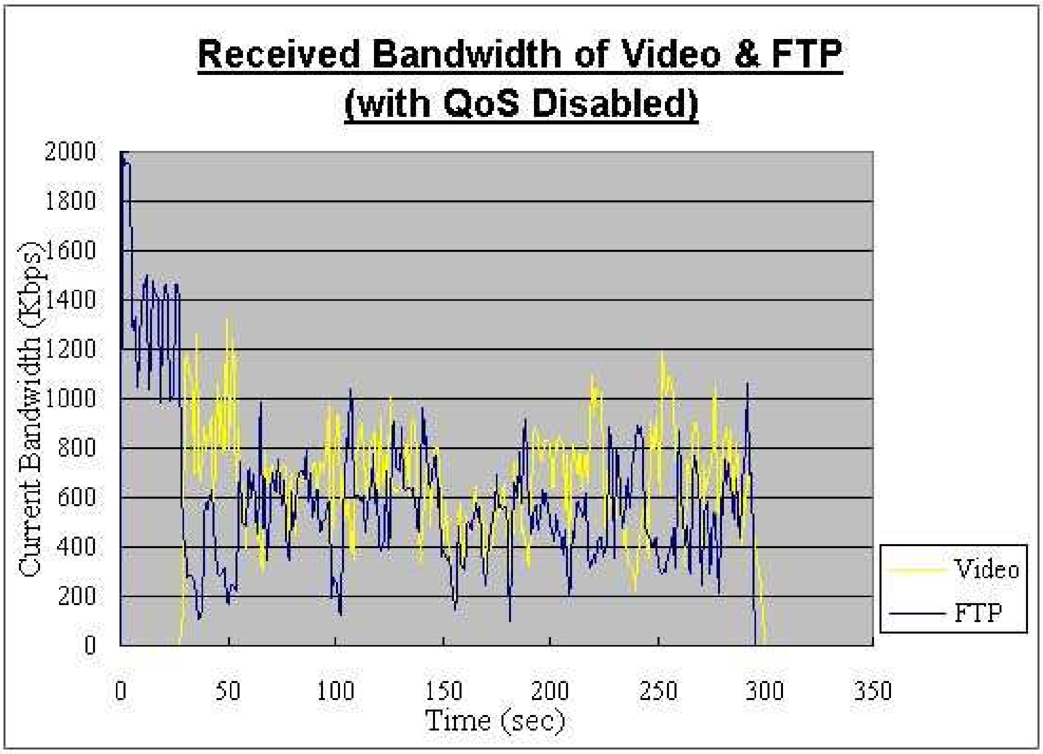

Figure 5.2: Current Received Bandwidth of Video & FTP Without QoS

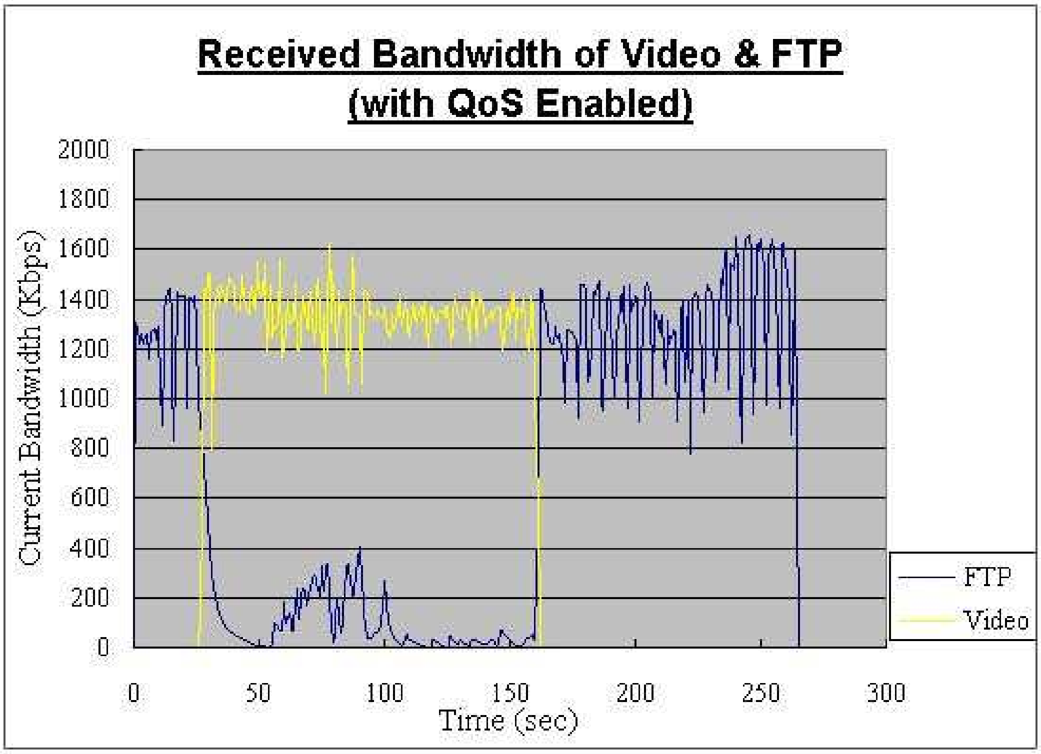

Figure 5.3: Current Received Bandwidth of Video & FTP With QoS

It is shown in the figures that by using QoS, the video played for about 2.27 minutes, and the FTP used up about 4.42 minutes. But without QoS, the video played for about 4.55 minutes, and the FTP used up about 4.91 minutes. This shows that under QoS, both applications can use up less time and the video played more smoothly than without QoS.

5.3 Analysis of The Results

In the above experiment, the effect of using QoS is shown. You can see that during using QoS, the video can play more smoothly, and it is also shown from the graphs that, the received bandwidth of video is much more steady than without QoS. This is because when QoS is enabled, real-time video application has higher priority and so bandwidth is reserved for video application. You can see from figure 5.3 that when the video is playing, the FTP application can get a little bandwidth, as most of the bandwidth is used by video application. This can ensure that the video can play smoothly in through the networks. However, when QoS is disabled, no application has higher priority and so both video and FTP applications are competing for the bandwidth. You can see from figure 5.2 that both applications have large fluctuation in received bandwidth. In the demonstration, you can see that the video application's buffer is always empty, and the video is always stopped due to not enough packets in the buffer.

Conclusion can be made from the performance evaluation that QoS is applied to wireless network successfully. The results is satisfied in both audio and video applications. And so, by using the QoS scheme, wireless network can transfer real-time multimedia data smoothly.

5.4 Improvement

Improvement can be made in addition to what we have already done. Actually there are many other ways to provide QoS. We are going to give two improvements that can be made now.

5.4.1 Frame order

There are different kinds of frames in MPEG format, namely I-frame, B-frame, P-frame and D-frame. D-frame is no longer used now (and seldom if used). There usually ordering of the frames are IBBPBBPBBIBBPBBP... . I-frame is the independent frame, which does not need other frames in order to decompress itself. P-frame depends on the previous I-frame in order to decompress while B-frame depends on both the backward and foreward frames in order to decompress. So, we can improvement the streaming but sending I-frame and P-frame first and finally the B-frames. So the client can first decompress the I-frame and P-frame and at the same time receiving the B-frames. After they are decompress, the B-frames can then be decompressed without waiting for the P-frame. This is a tricky method, however, we have not yet implemented this in our system.

5.4.2 RAID

In server side, the video files can be stored in RAID such that the retrieval of them can be done in parallel, and therefore reducing the overhead needed to read the video streams. We can store different frames in different disks such that we can read the I-frame, P-frame and B-frames in parallel. However, synchronization should be made for the reading of frames, since I-frame and P-frame should be sent before the B-frames are sent.

Bibliography

[1] Wan Yung Chun, Lam Shu Yin. *Computer Science and Engineering Department (CUHK) Final Year Project (LYU9802) Quality of Service in Communications Networks Report version 4.0.*

[2] Wan Yung Chun. *Computer Science and Engineering Department (CUHK) Final Year Project (LYU9802) Quality of Service in Communication Networks: Techniques and Evaluation Final Report.*

[3] Lam Shu Yin. *Computer Science and Engineering Department (CUHK) Final Year Project (LYU9802) QoS in Linux: Traffic Control Implementation, Installation and Evaluation Final Report.*

[4] Daniel L. Lough, T. Keith Blankenship, Kevin J. Krizman. *A Short Tutorial on Wireless LANs and IEEE 802.11.*

[5] Hui Zhang. *QoS Control And Resource Management in the Internet, Too Much or Too Little.* Cornell University Computer Science Colloquium, Nov 1998.

[6] Kate Gerwig. *Netting Quality Is No Simple Catch – Glut Of Protocols Could Delay Differentiation By Five Years.* News Analysis at http://www.tele.com, October 04, 1999.

[7] Sally Floyd and Van Jacobson. *Link-sharing and Resource Management Models for Packet Networks.* IEEE/ACM Transactions on Networking, Vol.3, No.4, 1995.

[8] David G. Messerschmitt. *QoS and Wireless Access.* Dept. of EECS University of California at Berkeley.

[9] David Kidston. *Wireless QoS Issues: An Overview.* University of Waterloo, Sept. 10, 1997.

[10] Prof. John C.S. Lui. *CEG4430 Lecture Notes.* The Chinese University of Hong Kong.

[11] Jean Tourrilhes. *Linux Wireless LAN Howto.* 8 September 1999.

[12] David A Rusling. *The Linux Kernel.* 1996-1999.

[13] Ori Pomerantz. *Linux Kernel Module Programming Guide.* 1999.

[14] David Hinds. *Linux PCMCIA Programmer's Guide.* 6 January 2000.

[15] Michael K. Johnson. *Linux Kernel Hackers' Guide.* 1996, 1997.

[16] WaveLAN/IEEE System Team. *DRAFT Software Interface Specification for Wireless Connection Interface for WaveLAN/IEEE (HCF-light).*

[17] MpegTV. *http://www.mpegtv.com*

[18] *http://www.proxim.com*

[19] *http://www.wavelan.com*

[20] *http://www.networkcomputing.com*

[21] *http://www.cl.cam.ac.uk/Research/SRG/netos/diskqos*

[22] *http://www.qosforum.com*

[23] *http://www.isi.edu/div7/rsvp/rsvp.html*