MASTER IN COMPUTER SCIENCE

Power characterization of network interfaces for low power wireless embedded systems

Briquet, Benjamin

Award date:
2003

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Power characterization of network interfaces For low power wireless embedded systems

Benjamin BRIQUET

Promoteur : Professeur Jean Ramaekers

Travail de fin d'études réalisé en vue de l'obtention
du titre de Maître en Informatique

Année Académique 2002-2003

UBS 10027073

Abstract

These last years, the portable devices such as PDAs are a sharp success. They are increasingly powerful at the point to be considered as genuine portable computers. Thanks to the rise of the wireless network, PDAs can benefit today from networks LAN. Unfortunately, this power and this portability affect to a significant degree their autonomy. This dissertation presents first the wireless technology to introduce the iPAQ H3600 provided with a Cisco wireless card. Various Dynamic Power Management (DPM) techniques are presented. Then, we show the interest to develop some techniques in the high-layers of the TCP/IP stack in order to reduce the energy consumption of the wireless card. We characterize after that the energy that can be saved by presenting a technique selected and implemented at the TCP layer, therefore well above the techniques implemented on the level of the wireless card (MAC layer). Lastly, we carry out experiments in order to measure the energy consumption of the card in its various operating modes and states because of the lack of data available on this subject and we indicate the impacts which these techniques have on the performance using a video application streaming.

Résumé

Ces dernières années, les appareils portables en tous genres tels que les PDAs connaissent un vif succès. Ils sont de plus en plus puissants au point d'êre considérés comme de véritables ordinateurs portables. Grâce à l'essor du réseau sans fil, les PDAs peuvent aujourd'hui profiter des réseaux LAN. Malheureusement, cette puissance et cette portabilité affectent de façon significative leur autonomie. Ce mémoire présente d'abord la technologie du sans fil pour présenter l'iPAQ H3600 muni d'une carte sans fil Cisco. Différentes techniques de gestion d'énergie (DPM) sont présentées. Ensuite, nous montrons l'intérêt d'en développer aux couches hautes de la pile TCP/IP afin de réduire la consommation d'énergie de la carte sans fil. Nous caractérisons par après l'énergie qui peut être épargnée en présentant une technique choisie et implémentée à la couche TCP, donc bien au-dessus des techniques implémentées au niveau de la carte sans fil (couche MAC). Enfin, nous effectuons des expérimentations afin de mesurer la consommation d'énergie de la carte dans ses différents modes de fonctionnement et états et ce, à cause du manque de données disponibles à ce sujet et indiquons l'impact qu'ont ces techniques sur la performance à l'aide d'une application vidéo streaming.

Acknowledgements

This thesis was the product of many hours of work, frustrating and stimulating in equal measure. It could not have been produced without the help of many people; the author therefore wishes to thank:

Professor Jean Ramaekers, to have given me the possibility to carry out my training period and for the reading of my successive drafts;

Professor Luca Benini, who facilitated my integration in his team in Bologna and supervised my training;

Assistant Professor Davide Bertozzi, who tolerated my frequent emails, and for his guidance and insight;

Assistant Professor Andrea Acquaviva, for its advices and encouragement during my stay in Urbino;

Jacques and Christine Briquet, my parents who have supported me every step of the way in my educational career. For the opportunities they have provided to their children and for teaching us all that our abilities were only limited by our imaginations. For pushing us, understanding us, and helping us to achieve our dreams;

Sophie Labeye, my girlfriend, for being wonderful who has supported my jumps of mood and whose encouragement has always been of great value. It is a debt I can never repay. I could not have done it without you;

Alain Decostre, my partner and a friend of mine, who has supported me during the many incredible adventures that Italy had held for us;

and last but not least, Jacqmotte, for making coffee that sustained me many a late night working on this thesis.

Contents

List of Figures

List of Tables

Introduction

Mobile computing is dramatically changing our day-to-day lives, especially with the popularity of small devices such as Personal Digital Assistants (PDAs), cellular phones and laptop computers. If these devices offered at the beginning only very limited functions or very poor performance, they quickly became very powerful at the point to be considered genuine multimedia devices like personal computers. The device that can be recorded its greater growth in term of sale is perhaps the PDA. Often provided with a fast processor, with a high definition color screen, with 32 or 64 Mb of read-write memory, the PDA, compact and light, became a convivial and powerful tool with wide capacities.

The problem of these devices is their autonomy. Power is a critical, limited, and shared system resource. The high-performance and the low-weight of PDAs impose enormous constraints in the design of battery. In order to extend the battery lifetime, managing efficiently the power consumption without too much compromising the performance has thus become a major concern. This problem can be addressed at various levels: in hardware by improving battery technology, by engineering more efficient electronics and components but also in software by implementing power consumption techniques in the drivers, in the operating system or even in applications. All the system can be designed in order to reduce the power consumption.

Dynamic Power Management (DPM) is a design methodology aiming at reducing power consumption of electronic systems, by performing selective shutdown of the idle system resources. The goal is thus to define policies that can be used to control the power consumption levels of the electronic system, by setting its components in different states, each characterized by a performance level and a power consumption. The effectiveness of a power management policy depends critically on an accurate modeling of the environment, and on the computation of the control policy.

In the Internet age, the network card forms integral part of the information processing systems, including PDAs. The network card most adapted for PDAs are without null doubts the wireless card that allows mobility while remaining connected to the world. Wireless networking is unquestionably a hot topic within the computing world, and with good reason. Wether it is video applications, web browsing, e-mail or simply the exchange of information, the ability to communicate without the encumbrance of a cable is a compelling prospect indeed. Connecting a wireless card to a PDA do not be thus marginal any more. DPM can thus also apply to wireless cards since these last drain fast the battery when they are used.

Several power management techniques are currently being studied in order to decrease the power consumption of these components. They relate to all the layers of the network, mainly of TCP/IP. IEEE 802.11, the standard of the wireless protocol, succeeds in standardizing the use of a mode of low consumption of energy for all the wireless network cards which respects this standard. The interest today is to define techniques at a higher level to profit from the visibility on the network than have the high-layers of the protocol stack.

This present dissertation focuses first on the need of developing power management techniques at higher layers of the protocol stack in order to reduce energy. However, the majority of power management techniques developed are specific to a traffic profile or a layer and rely on the in-built power policies implemented in the wireless card. However, the majority of commercial datasheets do not provide sufficiently data to characterize the power consumption of the in-built power management techniques that they implement in hardware. Taking power measurements of the three power management policies implemented in the Cisco Aironet 350 series wireless card has been performed during my training period at the Department of Electronics, Computer Science and Systems (DEIS) of the University of Bologna (Italy), from September 2002 to January 2003.

This dissertation is structured as follows. The first chapter presents the Wireless Local Area Network (WLAN) and compares it with the wired network. We terminate the chapter by introducing the problem of the power consumption and the solution proposed by the IEEE 802.11 standard.

The second chapter gives an overview of the iPAQ H3600 by describing each of its main components. The iPAQ H3600 is a powerful and multimedia PDA to which a wireless card can be easily connected. We present at the end of this chapter the Cisco Aironet 350 wireless client adapter and describe how the power issue is addressed in commercially available network interfaces and especially in the Cisco card.

In the third chapter, we describe the basic concepts of Dynamic Power Management. Three DPM levels are considered: the component level, the system level and the network level. Examples of power reduction techniques are also included. Each example concerns one of the main elements of a classical electronic system. This chapter has been written in cooperation with Alain Decostre.

In the fourth chapter, we give a simple overview of the Linux operating system that has been installed on the iPAQ and introduce the networking under Linux and in particular, TCP/IP. TCP/IP is presented in the form of a candidate to integrate power management techniques relating to the network interfaces.

In the fifth chapter, we recall the main mechanisms of TCP/IP in order to identify where it is worth working to implement some power management technique. We continue by giving the implementation of the memory buffers and we see how data is sent through the network stack. Finally, we conclude this chapter by reviewing the power management opportunities.

In the sixth chapter, we present a power management technique taken from the literature[1] and implemented in the transport layer, and especially in TCP. We present also how much power can be saved with this technique above the in-built power management techniques implemented in the Cisco Aironet 350 wireless card.

The last chapter characterizes the power consumption of the Cisco Aironet 350 wireless Client Adapter by giving the energy consumed by the card in its different states and modes when the iPAQ runs a video streaming application. We indicate that communicating over a wireless medium consumes significant energy. We present also the impacts of the CAM, PSP and PSPCAM power policies of the Cisco card on the performance. Based on the experimental results, we draw a conclusion.

We terminate this dissertation by a summary and by foreseeing the future possibilities of power management.

[1]This technique has been proposed by Davide Bertozzi from the DEIS lab.

Chapter 1

Local Area Network (LAN) and Wireless Local Area Network (WLAN)

1.1 Introduction

The main goal of this chapter is to give an overview of the well-known IEEE 802.3 network and the reference in the wireless network: IEEE 802.11b. These networks are very similar in the services that they must offer and a comparison seems natural.

First, we overview the IEEE 802.3 standard layer by layer following the Open Systems Interconnection Reference model (OSI model). Then we describe the 802.11b by comparing its mechanisms with these of IEEE 802.3 standard.

1.2 The OSI reference model

To describe the Wired Local Area Network and the Wireless Local Area Network, we will use the reference OSI model. The Wired Local Area Network, here after called LAN, and the Wireless Local Area Network (WLAN) differ only at the physical and the data link layer. The principle is that each layer of the OSI model exposes services to the layer above, the latter not having to worry about the implementation of these services. The figure 1.1 depicts the IEEE 802.3 and IEEE 802.11 standards compared to the OSI model. Description of these two standards are given in this chapter.

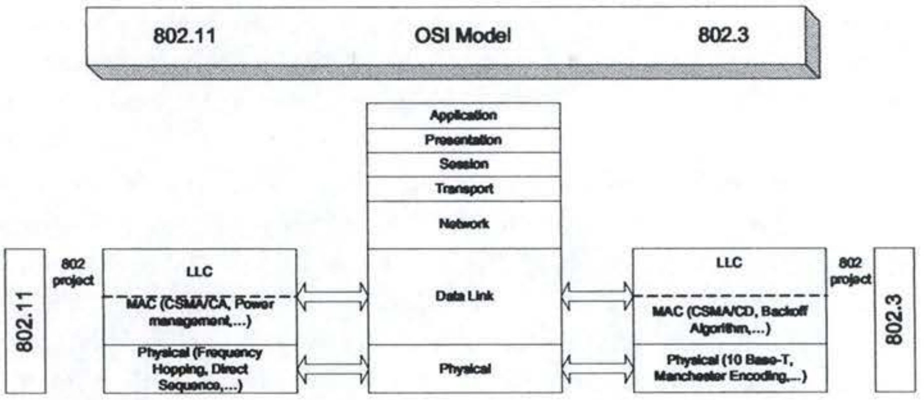
802.11
OSI Model
802.3
Application
Presentation
Session
Transport
Network
Data Link
Physical
802.11
802 project
LLC
MAC (CSMA/CA, Power management,...)
Physical (Frequency Hopping, Direct Sequence,...)
LLC
802 project
MAC (CSMA/CD, Backoff Algorithm,...)
Physical (10 Base-T, Manchester Encoding,...)
802.3

FIGURE 1.1: 802.3, 802.11 versus OSI Model

- The physical layer relates to the medium of transmission: it can be with physical guide (electrical cables and optical fibres) or without any physical guide (infrared waves, radio waves).
- The data link layer is responsible for regrouping bits from the physical layer into frames and handles transmission errors.

1.3 Wired Local Area Network

When there are only two computers, the communication between them is very easy. We can connect them by a simple cable and we have protocols like PPP or SLIP that manage the transmission of bits from one host to another.

When we have several computers that share a medium, they are said to contend for the channel. The problem comes from the fact that if computers send information at the same time, the different transmissions collide, are affected and no transmission is successful. The first solution is to define independent sub-channels, one for each transmission, which turns the channel (the medium) into a set of point-to-point links. Another solution is to define a controlled access (also called a deterministic access) to the medium to avoid collisions or on the contrary, a random access (also called a non-deterministic access) to the medium that defines host behavior when collisions occur. It is worth noting that when a host transmits information on the medium all hosts can hear this information but only one host, the destination, receives the information that is addressed to it. That is the reason why Local Area Networks are also called broadcast networks.

IEEE proposed three different implementations for wired Local Area Networks: IEEE 802.3, IEEE 802.4 and IEEE 802.5. In fact, these projects belong to a greater project IEEE 802 whose goal was to define standards for broadcast Local Area Networks to enable communications between equipments from different vendors.
IEEE 802.4 (Token Bus) and IEEE 802.5 (Token Ring) use a controlled access: one host can

communicate at a time on the medium thanks to a mechanism where a host must obtain permission before transmitting. On the contrary, IEEE 802.3 offers a random access based on the protocol CSMA/CD 1-persistent. We will see later in detail this kind of protocol.

Nevertheless, we will not particularly describe in this section IEEE 802.4 and IEEE 802.5 for mainly three reasons. First, because the Wireless Local Area Network implements another form of random access than one used by IEEE 802.4 and IEEE 802.5. Second, the wireless networking standards are compatible with the IEEE 802.3. Third IEEE 802.3 is the most used Local Area Network. In fact, Wireless Local Area Network is superficially similar to IEEE 802.3. Understanding the background of IEEE 802.3 helps slightly with Wireless Local Area Network. We will thus describe more the IEEE 802.3 network in comparison with the different Wireless Local Area Networks even if the latter ones can easily be opposed to other cable networks. Sometimes people call IEEE 802.3 networks Ethernet networks. Ethernet network mainly refers to the first version of LAN invented by three companies in the beginning of eighties and is slightly different from IEEE 802.3.

1.3.1 The physical layer of IEEE 802.3

The medium that we encounter in the world of IEEE 802.3 networks is composed of cables like 10Base5, 10Base2, 10Base-T and 10Base-F.

The first number specifies the speed of the network, the string "Base" specifies that it uses a baseband transmission scheme and finally the last number or letter indicates the maximal length of cable in the case of a number or the type of cable used in the case of a letter (T stands for Twisted pair, F stands for optical Fiber). For example, 10Base5 means that the network works at 10Mbit/s, that it uses a baseband transmission technique and that it accepts a cable of maximal 500 meters length by "section".

Technical transmission

Baseband transmission consists of sending directly the bits on the medium by an electrical current that can take two values. Because the electrical current can take two values, this technique is also called digital encoding. Either each value represents directly one bit (1 or 0), either it is a transition from a value to another that represents a particular bit (1 or 0). The IEEE 802.3 uses the Manchester encoding to transmit the bits across the medium.

What is the Manchester encoding?

A time interval is attached to each bit. The signal changes value in the medium of the interval. To code a "0" the current will be negative (example V0) on first half of the interval and positive (example V1) on second half, while to code a "1", the opposite holds. In other words, in the medium there is a transition from bottom to top for one "0" and from top to bottom for one "1". The figure 1.2 depicts the Manchester encoding.

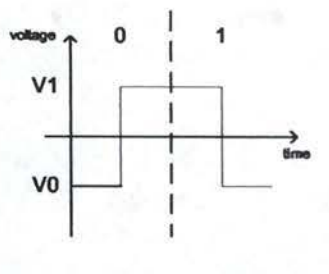

FIGURE 1.2: The Manchester encoding

- As we can see, Manchester encoding needs 2 values of current (V0 and V1) to encode only one bit. The throughput will thus be divided by two compared to an encoding where 1 bit is coded by only one value of the signal (one bit for one symbol).
- The advantage of using Manchester is that it allows synchronization between sender and receiver.

The maximal length of the cable

We have talked above about the maximal length of a 10Base5 cable. In fact, beyond 500 meters for this type of cable, we need a repeater that receives, amplifies and then retransmits signals in both directions because the power of signal decreases with the distance (attenuation phenomena). In the case of cables, the decrease of the signal is a logarithmic function of the distance.
In fact, signals sent across the medium are subject to three physical perturbations: attenuation, temporal distortion and noises.

- The attenuation corresponds to a loss of signal energy during its propagation across the medium. It is expressed in decibels per kilometer. It depends very closely of the signal frequency and of the bandwidth of the medium. The attenuation in a wired network is generally an logarithmic function of the distance. To understand why the attenuation is relative to the bandwidth of a cable, we must refer to the Fourrier formula[1] where a signal can be broken up into infinity of harmonics. In fact, the bandwidth of a cable refers to the width of a frequency band. Beyond this width, the different harmonics of a signal cannot be fit into the band and undergo some weakenings. The receive cannot thus rebuild these harmonics. Therefore, if there are many of these harmonics in the original signal, the receiver cannot rebuild entirely the original signal from the received signal. Bandwidth limits thus the number of harmonics that "can be fit" in a particular medium. A medium affects thus always a signal when it passes trough it. The narrower the bandwidth, the less numerous the harmonics cannot be affected and thus more important will be the effect of attenuation on the signal.

[1]A signal can be handled by a periodic *g(t)* function that has T as period: $g(t) = \frac{1}{2}c + \sum_{n=1}^{\infty} a_n sin(2\pi nft) + \sum_{n=1}^{\infty} b_n cos(2\pi nft)$ where $f = \frac{1}{T}$ is the fundamental frequency, a_n, b_n are the amplitudes respectively of sinus and cosine of row n. Such a decomposition is called a Fourrier series.

- The temporal distortion is caused by the difference of speed that each component of a signal has compared to others components of this signal. This distortion can lead to errors of transmission.

- The last perturbation that we can meet in cable networks is one that is caused by the noises. The noises are all undesirable signals which are present on the medium. Noises can be produced by the sender, by interfering the medium itself or for example by electromagnetic fields that affect the signal on the medium.

In fact, each type of medium offers a maximal bandwidth and it is more or less strongly prone to the noises, to the attenuations or to the temporal distortions. That is why we can see differences of "performance" between cables.

We will see in section 1.3.2 that the IEEE 802.3 protocol, at data link layer, restricts even more the total length of IEEE 802.3 network. We can already say that the total length cannot exceed 2.5 km and any part of the network cannot traverse more than 4 repeaters.

1.3.2 The data link layer of IEEE 802.3

As we can see in OSI model section, two sub-layers compose the data link layer in the 802.3 protocol: Logical Link Control (LLC) and the Medium Access Control (MAC). LLC is a common layer to all broadcast LANs while MAC layer is specific to the LAN technology. In other words, the IEEE 802.3 MAC is different from IEEE 802.4 MAC and from IEEE 802.5 MAC.

In fact, MAC layer is responsible for the management of the multiple access to the medium. As we have already seen in the introduction of this chapter, LANs are built around a single medium that is shared by several computers. This MAC layer must determine which computer is allowed to transfer data. This layer plays thus an important role in the LANs.

If we transmit bits on the channel at physical layer, we say from the MAC layer that we transmit frames. A frame is a logical group of bits. We will overview later the IEEE 802.3 MAC frame format.

The Medium Access Control (MAC) for IEEE 802.3

What is the access scheme of IEEE 802.3?

The access scheme is based on the principle that a behavior of a host is determined by "what's going on" on the medium. The station listens to the medium before transmitting. All the protocols that respect this principle are known under the name "carrier sense protocols" (CS protocols).

The IEEE 802.3 networks use the **CSMA/CD 1-persistent protocol** (CSMA/CD stands for Carrier Sense Multiple Access with Collision Detection).

CSMA: a station wishing to transmit must first listen to the channel (e.g. by measuring the channel's voltage level). If the channel is busy, another station is transmitting and our station must wait until it detects that the channel is idle. When the channel is sensed idle, our station can transmit. 1-persistent means that when a station decides to transmit a frame, the probability that it finds the medium free is equal to 1.

CD: during transmission, the station listens to the channel and if another transmission is detected (e.g. higher voltage level than expected for one transmission), all stations involved in the collision stop transmitting immediately. Each station generates a 32-bit jam sequence to inform other stations of this collision and then computes a randomly-sized time interval, waits for that amount of time, and begins the transmission attempt again.

If CSMA/CD can detect collisions when they occur, a capacity drawback remains because collisions waste valuable transmission capacity.

In addition, we can note another problem. Even if the station listens to the medium before transmitting, collisions can still occur because it takes non-zero time for signal to propagate along the channel. If we denote by t the propagation time, the worst-case collision detection time is $2t$. Indeed, the signal must return to the transmitting host.
However, the problem comes from the fact that the station listens before transmitting (CSMA), during transmission (CD) but not after the transmission. Therefore, if the station wants to be sure there is no collision with another frame, it must transmit during $2t$ time. This forced the IEEE 802.3 protocol to define a minimum frame length in CSMA/CD systems. For example, in a LAN of 10 Mbit/s with a cable of 2500 meters including 4 repeaters, the minimum amount of time required for transmitting a frame is 51.2 μs. This time enables sending a frame of 64 bytes.

That is why the IEEE 802.3 protocol imposes a maximal cable length too. The maximal length is 2500 meters with 4 repeaters at maximum with a LAN of 10 Mbit/s. If a frame has not a length of 64 bytes, the station must add zeros in the frame (padding).

There is also another reason why the protocol defines a minimum length frame. That allows stations to distinguish between valid frames and non-valid frames (due to collisions).

The Exponential Backoff algorithm

We have seen that a station must wait a randomly-sized interval when it detects a collision after transmitting a frame. The algorithm used in IEEE 802.3 is called Exponential Backoff. We will not describe deeply this algorithm because it is a complicated algorithm. The basic idea is that the more numerous the collisions, the longer the station may have to wait before transmitting again.

In fact, we can divide the time into three different periods: transmission period, contention period and idle period. The randomly-size interval corresponds to the contention period while idle period corresponds to the period where no station desires to send a frame and the transmission period corresponds to the period where at least one station transmits a frame. The contention period is also called “contention window”or “backoff window”. A contention window is divided into slots. The slot length is medium-dependent; higher-speed physical layers use shorter slot times. Stations pick a random slot and wait for that slot before attempting to access the medium; all slots are equally likely selected. When several stations are attempting to transmit, the station that picks the first slot (the station with the lowest random number) wins. The "backoff" principle is that the backoff time is selected from a larger range each time a transmission fails. The more numerous the transmissions increase, the more numerous the growth of contention window is important (the length of contention window increases more rapidly each time: in fact, the contention window moves to the next greatest power of two ($2^i - 1$), e.g., 31, 63, 127, 255.). Hence, the contention window is used to manage the access to the medium to avoid collisions. We will see later that this contention windows is also used in WLAN.
The slot length for IEEE 802.3 is fixed to 51.2 μs, which corresponds to the time needed by a station to ensure itself that the medium is free. The contention window is limited to 1023 transmission slots. After this length elapsed, the attempt at sending the frame is canceled. The contention period is on average (under the hypothesis of heavy traffic conditions: e.g. no idle periods) $5.4t$.

Finally, if the length of the cable increases, the transmission period must be larger too (e.g., the length of the frame must increase). The efficiency decreases because we must pad more frames on average. If the channel bandwidth increases, the transmission period must be larger too or the length of the cable must decrease. In the case of a bandwidth decreases, the efficiency increases because we spent more time to transmit useful bits on the channel (we pad less frames).

IEEE 802.3 Frame

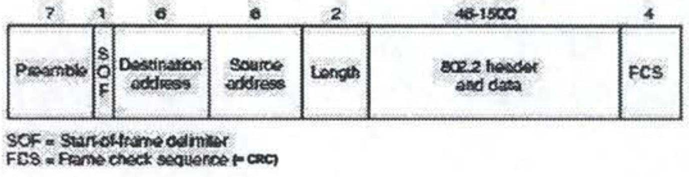

FIGURE 1.3: The 802.3 frame

Referring to the figure 1.3, the preamble allows receiver to synchronize with incoming transmission. SOF is the start flag in IEEE 802.3. Physical addresses have each a unique 6-bytes address. The maximum length of data field is 1500 bytes.

Logical Link Protocol

It offers a homogeneous format of frame and a homogeneous interface to the higher network layer regardless the underlying network LAN. This makes it possible to mask the differences between networks. As we have already said above, LLC is non-architecture specific: it is a common layer to all 802 broadcast LANs. IEEE 802.2 LLC defines therefore the encapsulation of higher-level protocol like IP or IPX for example.
It offers three types of services: non-guaranteed service in non-connected mode (LLC type 1), reliable service in connected mode (LLC type 2) and service with acknowledgments in non-connected mode (LLC type 3).

1.4 Wireless Local Area Network

In 1997, after seven years of work, the IEEE published 802.11, the first recognized international standard for wireless LANs. Mainly because of limited throughput of this first version of protocol that can only reach 1 and 2 Mbps data rates and because of limited wireless flexibility, in September 1999 they published the 802.11b. It uses the DSSS method access to the 2.4 GHz band frequency and allows two higher speeds (5.5 and 11 Mbps in addition to the 1 and 2 Mbps). 802.11b protocol is not really completely different from 802.11 but it is rather a better-defined protocol that has allowed launching the WLAN into the market and competing seriously with other LANs. There are currently a lot of Working Groups and Test Groups (WG and TG) who work actively in the WiFI world:

- 802.11a is the second wireless project using the 5GHz band. It can theoretically achieve a bandwidth of 54Mb/s.
- 802.11g is an adaptation of 802.11b in the same band (2.4GHz) and allows to achieve a bandwidth of 54Mb/s.
- ...

In this section, we see the general working of IEEE 802.11 (which is also the same one as 802.11b), comparing with IEEE 802.3 network. For that, we follow the same OSI presentation structure. Even if we can report a lot of differences between IEEE 802.11 and 802.3 networks, they have a shared heritage at such point that IEEE 802.11 is sometimes called "wireless Ethernet".

The WLAN network is wireless and promote therefore mobility. That is one of the mainly reasons the WLAN was built and a lot of differences between wired LAN and WLAN come directly or indirectly from that particularity. Stations can move while connected to the network and transmit frames while in motion. The stations that are used in WLAN are often indeed battery-operated portable devices. That is why the IEEE 802.11 considers mechanisms to reduce the power consumption of these battery-operated portable devices. We will introduce the problem of power consumption at the end of this section.

We will see that the major problem is dealing with the difficulties transmitting over the air. Both physical and data link layers try to reduce interference with signal by implementing several mechanisms. That is mainly what we will see now, in the next sections.

1.4.1 The physical layer of IEEE 802.11

The reliability of wireless transmission is really not comparable to the reliability of a wired network. The air is more subject to interference than a shielded cable. The physical layer in WLAN is thus more complex than the physical layer of other LANs and particularly of IEEE 802.3. That is the reason why the physical layer in WLAN is split into two sub-layers: Physical Layer Convergence Procedure (PLCP) and Physical Medium Dependent (PMD). PLCP adds its own header to the MAC frame. This header is in particular useful for synchronization. Indeed, the preamble is added in this layer and no more in MAC layer like in 802.3 protocol. The PMD is responsible for transmitting any bit it receives from the PLCP into the air. We can see the PLCP frame on the figure 1.4.

The PLCP layer and its frame

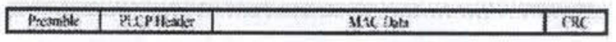

FIGURE 1.4: PLCP frame

PLCP has two structures, a long and a short preamble. All compliant 802.11b systems have to support the long preamble. The short preamble option is provided in the standard to improve the efficiency of a network when transmitting special data such as voice, VoIP (Voice-over IP) and streaming video.

PMD layer

Most of WLAN today use the 2.4GHz ISM band. ISM stands for Industrial, Scientific and Medical. It is indeed a frequency band reserved for individual users. Nobody must pay the right (a license) to use a device working in this band. The drawback of this is that this band is very crowded compared to other licensed bands: a lot of devices use this band. It adds thus a lot of noises and interference in IEEE 802.11 networks.

Spread spectrum versus traditional radio communication

Traditional radio technique tries to put as much signal as possible into a band as narrow as possible. Spread spectrum technique is different because the modem spreads the transmission over a wide band and uses more bandwidth than the system really needs for transmission to reduce the impact of local interferences (bad frequencies) on the system. For that, mathematical functions are used to spread the signal power over a large range of frequencies. The receiver performs the inverse operation, the signal is reconstructed as a narrow-band signal, and, more importantly, any narrow-band noise is smeared out so the signal shines through clearly. Spread spectrum does a better job of dealing with interference compared with traditional radio technique.

WLAN uses Frequency-hopping (FH) spread spectrum radio technology or Direct sequence (DS) spread spectrum radio technology to transmit bits in the air. In 1999 with IEEE 802.11b, another transmission technology was used: High-Rate Direct Sequence (HR/DS or HR/DSSS).

Direct sequence spread spectrum

"The principle is to spread the signal on a larger band (spread spectrum) by multiplexing it with a signature (the code) to minimize interference and background noise. The system works over a fixed large channel. To spread the signal, each bit is sur-modulated by a code (a fast repetitive pattern). In the receiver, the original signal is recovered by receiving the whole spread channel and demodulating by the same code"[2].
Because the interference is at a particular frequency, it has less bad impact on the Direct Sequence system than other systems like traditional radio system because the former uses a larger bandwidth so the interference represents a smaller part. Moreover, the receiver must use the code to demodulate the received signal (signal processing), which has the effect to decrease signals not modulated by the code. It allows "to identify"(recover) the useful signal relatively to other signals (noises,...).

Because it uses a large channel, direct sequence systems dispose only a few channels in the bandwidth. In general, there are 3 separate channels that do not generate interferences on each other.

Frequency hopping

Frequency hopping systems jump from one frequency to another. The 2.4GHz ISM band is divided in 79 channels of 1MHz. Periodically, the system hops to a new channel following a cyclic random pattern. The idea is that travelling through a lot of channels, it limits the use of a channel disturbed by interference.

Frequency hopping systems are cheaper than direct sequence and moreover sophisticated signal processing is not needed to extract the bit stream from the radio signal like in direct sequence systems.

However, even if direct sequence systems are more complicated and need more electrical power consumption than frequency hopping systems, it allows a higher data rate. That is the reason why it is more used today in wireless LAN. It is a turbo version of this direct se-

[2]TOURRILHES J., A bit more about the technologies involved..., Hewlett Packard Laboratories, Palo Alto, 3 August 2000.

quence technology that we find now in WLAN devices that respect the IEEE 802.11b standard.

High-Rate Direct Sequence (HR/DS or HR/DSSS)

To increase the speed of transmission, the key idea is to play with the modulation technique. Unlike in IEEE 802.3 networks where the transceiver must change from a signal value to another to represent only one bit (remember that Manchester encoding "divides" by two the bandwidth: e.g., two symbols for only one bit), here high-rate direct sequence system represents more than one bit per symbol. In fact, 802.11b physical layer uses the Complementary Code Keying (CCK) that represents 8 bits by symbol and achieves 11 and 5.5Mb/s rate.

Comparisons with IEEE 802.3

- Medium: IEEE 802.3 uses cables while IEEE 802.11 uses the air.
- Encoding and transmission techniques: IEEE 802.3 uses Manchester encoding while IEEE 802.11 uses Complementary Code Keying (CCK) with Quadrature Phase Shift Keying (QPSK) modulation and Direct-Sequence Spread-Spectrum (DSSS) technology.

Transmission over the air

As we have already mentioned above, the reliability of a wireless network is completely different from a wired network. In addition to the three perturbations that we have seen in IEEE 802.3 section, there are other phenomena that affect the radio waves. Multipath and fading are the major disturbing effects of radio waves in addition to noises caused by all devices using the 2.4GHz band.

Multipath and fading

When waves are transmitted into the air, they can reflect, scatter or diffract in buildings, walls, ceiling and so on. Radio signals travel thus more than one route between the transmitter and the receiver. This phenomenon is called multipath propagation. Some signals are thus delayed compared to other signals that take more direct route and so, they do not reach the receiver at the same time. This can result in what is called frequency selective fades (fading), meaning that, at certain frequencies, the signal may attenuate or disappear. This can happen because the different delayed signals are summed at the receiver. The result signal can be smaller in amplitude than either of the individual signals. The figure 1.5 plots this case with two signals.

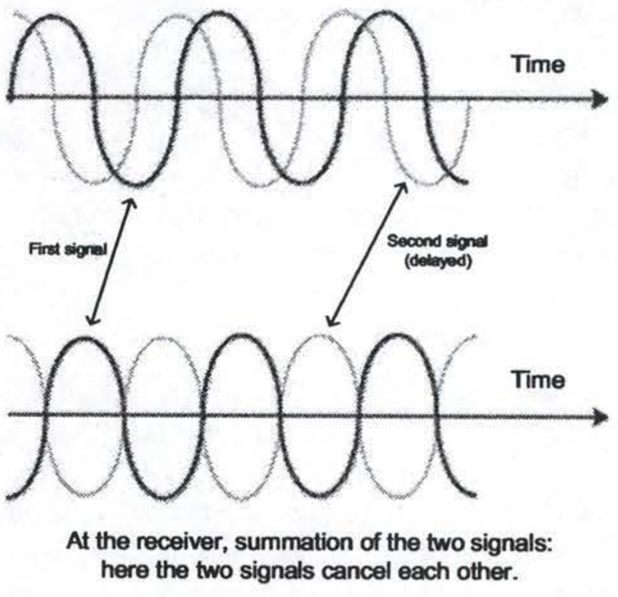
Time
First signal
Second signal (delayed)
Time
At the receiver, summation of the two signals: here the two signals cancel each other.

FIGURE 1.5: Multipath and fading in the receive signal

We do not forget that the receiver can also move. This results in a variation of the different path lengths. The phase in relation to each signal coming from the different paths will thus also vary. The phenomenon described just above appears and the received signal vary in amplitude. For narrow band signals (using only a very narrow section of the frequency spectrum) these power fades can result in a total loss of signal. Of course, that depends also on the degree of fade and the sensitivity of the receiver. For wide band signals, multipath can result in periodic frequency fade and intersymbol interference (delay spread). Because information is spread over a wider bandwidth, multipath generally results in distortion of the signal but not the total loss. The main problem is the fact that there are overlaying copies of the signal each time having a different delay. The symbols used to represent the digital information show thus also some overlapping. If this overlapping is important, that can lead to interference, which makes difficult to demodulate the signal without bit errors.

Comparisons with IEEE 802.3

As we can see, WLAN is more subject to interference than a wired network. Even attenuation is stronger in WLAN than the attenuation in a wired network. We have said that the attenuation in a wired network is generally a logarithmic function of the distance while the attenuation in WLAN is proportional to the square of the distance. We can also easily appreciate that waves into air are more subject to noises than waves in a cable. In addition to the well-know physical perturbations that we have encountered in wired networks, we have new type of perturbations (delay spread, multipath fading) that can be different from an environment to another. These "dynamic" perturbations make the wireless communications less robust and need a deeper study of the environment before deploying a wireless network.

Due to its "overture to the open air", it is worth noting now that a wireless network is also less protected from the hackers than a wired network. If the protection cannot be assured by the physical layer, the MAC layer must define a mechanism to reduce the danger of attacks. Wired Equivalent Privacy optional service was defined in 802.11 to protect link-layer from active and passive attacks. Nevertheless, on this point too, WLAN is less robust than a wired network because several flaws have already been discovered in WEP. IEEE 802.11i targets the MAC layer to reinforce the security in order to replace WEP.

Design and topology of IEEE 802.11

IEEE 802.11 networks are often composed of 4 components: distribution system, access points, wireless medium (the 2.4 GHz frequency band) and stations.

A station is a computing device with a wireless interface. Stations of a wireless network are often portable and can communicate each other either directly (independent mode or ad-hoc mode) either through an access point (infrastructure mode). An access point is a device composed of two interfaces: a wireless interface and often an IEEE 802.3 (or Ethernet) interface. The main function of an access point is to deliver frames outside its range using the Ethernet interface. It contains what we call the wireless-to-wired bridging function. The distribution system is the logical component responsible for forwarding frames to their destination when several access points are connected to form a large area. The distribution system is composed of the bridging function and of the distribution system medium, which is often an Ethernet or IEEE 802.3 network.

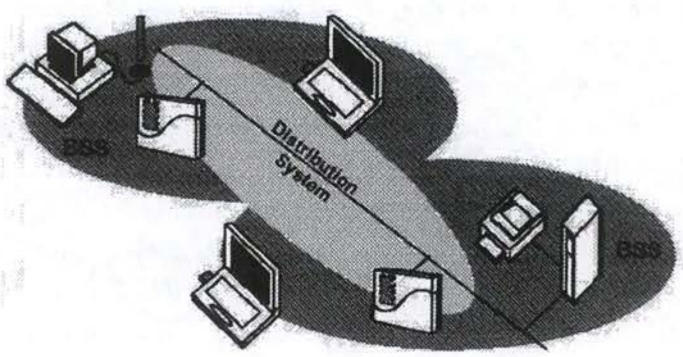

FIGURE 1.6: an example of an Extended Service Set (ESS)

This configuration (figure 1.6) defines a Basic Service Set (BSS) and can be assembled with another BSS to form an Extended Service Set (ESS).

1.4.2 The data link layer of IEEE 802.11

Like in IEEE 802.3 network, the data link layer is composed of two sub-layers: Logical Link Control (LLC) and Medium Access Control (MAC). Remember that LLC is a common layer to all broadcast LANs while MAC layer is specific to the LAN technology. Consequently, there is only the IEEE 802.11 MAC layer that can be compared with the IEEE 802.3 MAC layer. IEEE 802.11 is thus another type of network LAN and thanks to that "fits correctly" in the OSI stack and particularly in the IEEE 802 projects. LLC layer ensures the interface with the higher network layers (IP, IPXĚ). We will hence not talk about LLC layer in this WLAN chapter.

The Medium Access Control (MAC) for IEEE 802.11

What is the access scheme of IEEE 802.11?

There are two modes defined in IEEE 802.11. First, the Distributed Coordination Functions (DCF) present inside the specification provide a distributed access with no centralized controller, like IEEE 802.3. We will see how it works under the hood just afterwards. Even so, as mentioned above (see the introduction of Wired Local Area Network), Wireless Local Area Networks do include a certain form of a controlled access like one defined in IEEE 802.5. They are the point coordination functions (PCF) that provide this mode. In this mode, the Access point, the point coordinator, polls each station to say them that they may transmit or not. This polling resembles to the token in IEEE 802.5 networks where a station must have a token to have the authorization of transmitting. This mode is also called "contention-free services" because we have not any contention period to manage the access to the medium (see "Exponential backoff"under the section 1.3.2). We will not describe this access mode because it is an optional service in IEEE 802.11 and so it is not widely implemented yet.

The problems of CSMA/CD:

Even the WLAN looks like an Ethernet or an IEEE 802.3 network, the CSMA/CD cannot be applied here mainly because it's impossible to listen during transmission (CD) under penalty of needing two transceivers, which would be too expensive. These transceivers are in reality half-duplex: either they send a frame either they listen to the medium but not both at the same time. The solution consists of avoiding collisions instead of being capable to detect them. IEEE 802.11 uses CSMA/CA (CA stands for Collision Avoidance).

How can we avoid collisions?

802.11 uses interframe spacing to avoid collision. There are four different fixed interframe spaces:

- Short InterFrame Space (SIFS): The SIFS is used for the highest-priority transmissions. High-priority transmissions can begin once the SIFS has elapsed. Once these high-priority transmissions begin, the medium becomes busy, so frames transmitted after the SIFS has elapsed have priority over frames that can be transmitted only after longer intervals.
- Point coordination functions InterFrame Space (PIFS) frame is used during contention-

free operation. We will not consider them here.

- Distributed coordination functions Interframe Space (DIFS) is the minimum medium idle time for DCF.
- Extended InterFrame Space (EIFS): that is the only interframe space that is not fixed. It is used when there is an error in frame transmission.

The relation SIFS < PIFS < DIFS between the different interframe spaces must be respected.

The principle is that, before transmitting, a station must listen to the air during a period equivalent to one of these periods. The period "chosen"by the station depends of the priority of the transmission. If the medium remains free during this period, the transmission can begin immediately. Hence, if a station waits only a period equivalent to the SIFS after a frame was sent on the medium, it will have the priority compared to stations that must wait a DIFS period.

Unfortunately, this not the only problem that we face and CSMA/CA does not solve all the problems because even if a station hears the medium before transmitting (CSMA), it may happen that this station may not be able to hear all stations. There can be indeed stations that are out of its range. This is the "hidden node problem". Actually, the problem appears when the station transmits a frame to another station while this latter is receiving a frame from a station, which is out of range of the first. The station that is receiving the two frames cannot make sense of anything. The figure 1.7 depicts this situation.

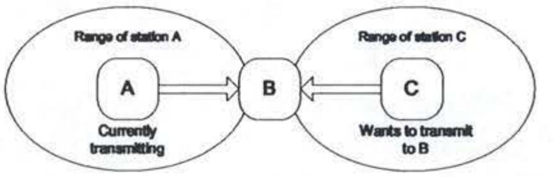

FIGURE 1.7: The "hidden node problem"

The RTS/CTS mechanism

To solve this problem, stations use an RTS/CTS mechanism. RTS stands for Request To Send and CTS for Clear To Send. If a station wants to transmit a frame, this station send a RTS frame after listening to the medium during a DIFS period. This special control frame will reserve on the one hand the medium for the station that has transmitted it and in addition silence any station that hears it. It allows silencing any station near the sender. Though it is not enough to resolve the hidden node problem because the stations in the vicinity of the receiver that did not hear the RTS of the sender may send a frame and disturb the receiver. That is why the receiver uses the special control frames CTS that, on the one hand, advertises the sender that it is "Clear To Send" and that, in addition, silences the stations in the range of the receiver. To speed up this exchange, SIFS periods are used after the sender of RTS frame obtained the medium.

Positive acknowledgment

Unlike IEEE 802.3, WLAN incorporates furthermore positive acknowledgment. Positive acknowledgment means that each transmitted frame must be acknowledged by the receiver only when the received frame has been correctly transmitted. If a frame is not acknowledged, the sender supposes, after the expiration of a timer, that the frame has not reached correctly the destination and so tries to retransmit the frame. It is worth noting that a positive acknowledgment is send by the receiver after a SIFS period only and not after a DIFS period. It allows the receiver to have priority compared to other stations. It avoids that the medium is reserved by another station and defers the acknowledgment. In truth, it could cause the expiration of the sender timer whereas the frame was correctly received. It is worth noting that only unicast frames are acknowledged, the broadcast and multicast frames are not acknowledged.

How does the control frames RTS/CTS reserve the medium?

The Network Allocation Vector (NAV) provides another "carrier sensing" (CSMA), this time virtually. In fact, most 802.11 frames carry a duration field, which can be used to reserve the medium for a fixed period. The NAV is a timer that indicates the amount of time the medium will be reserved. Stations set the NAV to the time for which they expect to use the medium, including any frames necessary to complete the current operation. Other stations count down from the NAV to 0. When the NAV is nonzero, the virtual carrier-sensing function indicates that the medium is busy and when the NAV reaches 0, the virtual carrier-sensing function indicates that the medium is idle. In this manner, the hidden nodes will not transmit for the specified NAV duration.

Fragmentation

Another mechanism is often used to improve the reliability in the presence of interference. Wireless LAN stations may attempt to fragment transmissions so that interference affects only small fragments, not large frames. By immediately reducing the amount of data that can be corrupted by interference, fragmentation may result in a higher effective throughput. Once the sender has gained the access to the medium, it sends fragments and their acknowledgments after a SIFS period. The sender can send all the fragments in sequence, in a "fragmentation burst" because the SIFS period allows it to keep the medium. Nevertheless, NAV is also used to ensure that other stations do not use the channel during the fragmentation burst. RTS/CTS mechanism is often also used with fragmentation.

Summary of CSMA/CA: how does it work?

1. If the medium is free for longer than the DIFS, transmission can begin immediately.
   - If the previous frame was received without errors, the medium must be free at least the duration of DIFS.
   - If the previous transmission contained errors, the medium must be free for the duration of the EIFS.
2. If the medium is busy, the station must wait for the channel to become idle. The station waits for the medium to become idle for the DIFS and prepares the exponential backoff (see above for a description of this algorithm).

There is an error of transmission either when the sender of a frame does not receive an acknowledgment or either when a station fails to gain access to the medium. The station thus tries to retransmit the frame or to regain the access of the medium after having waited for a DIFS period and a contention period. This contention period is determined by the backoff algorithm.

As we have already mentioned, a station can transmit an acknowledgment or the frames transmitted in RTS/CTS exchange (except for the first RTS frame) or fragments in fragment sequences after a SIFS period. It allows the station to have the priority.

IEEE 802.11 frame

There are three types of frames: data frame, control frame and management frame. The figure 1.8 depicts the generic format of the frame:

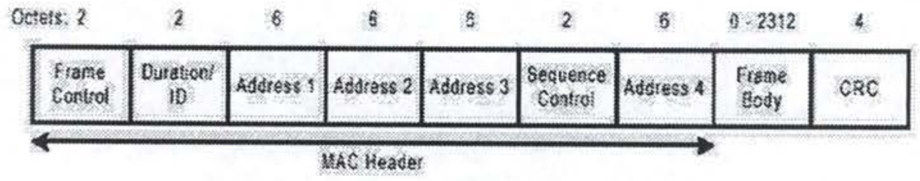

FIGURE 1.8: The 802.11 Frame

Unlike IEEE 802.3 frame, we cannot find the length field and the preamble. For the preamble, we have in fact already seen it at physical layer (PLCP layer) while length field is present inside the header on the data carried in the 802.11 frame.

- Frame control: we can find information like the protocol version, the type of frame, the more fragments bit, the power management bit, the more data bit,...
  - The type of frame is a set of 4 bits that defines the type of frame (RTS, CTS, Acknowledgment, data,...).

  - Power management bit says if the sender is in a power-saving mode after the completion of the current frame exchange. "1" indicates that the station will be in power-save mode, and "0" indicates that the station will be active.
  - If the "more data fragment"is enabled, it means that a fragment of the frame will still follow this one. It is used to distinguish the different fragments of a frame from the last fragment of this frame that contains the more data fragment set to 0. We should not confuse with "more data" bit that is used by the access point when a station is in power-saving mode.

- Duration/ID field: this field is used either to set the NAV, either during the contention-free services or for the PS-Poll frames. We will see later the use of PS-Poll frames.

- Address fields: these fields are perhaps the biggest difference with IEEE 802.3 frame. Altough these address fields are 48-bit long like in IEEE 802.3, there are here 4 address fields instead of two.
  - Destination address: the MAC identifier that corresponds to the final recipient: in other words, the station that will hand the frame to higher protocol layers for processing.
  - Source address: the MAC identifier that identifies the sender of the frame.
  - Receiver address: this MAC address references the station that should process the frame. If it is a wireless station, then this field has the same value that the destination address field. Otherwise, if it is a station on an IEEE 802.3 network connected to an access point, the receiver address corresponds to the wireless interface of the access point while the destination address is the address of the station attached to IEEE 802.3 network.
  - Transmitter address: another MAC address that we will not describe.
  - Basic Service Set ID (BSSID): this is the MAC address of the wireless interface in the access point. This allows to a station to define to which BSS it belongs.

Most data frames in fact use only three fields for source, destination, and BSSID.

- Sequence control field: a field used in particular for defragmentation.

- Frame body: field that contains higher-layer data to transmit.

- Cyclic redundancy check (CRC): this field allows the receiver to check the integrity of received frames. The header and the data are included in the CRC. Although 802.3 and 802.11 use the same method to calculate the CRC, the MAC header used in 802.11 is different from the header used in 802.3, so the CRC must be recalculated by access points when frames passes through the two networks.

Comparisons with IEEE 802.3

A great deal of mechanisms is used in IEEE 802.11 to avoid collisions and errors of transmissions due to interferences. Even if the network administrator can often configure these mechanisms, the direct consequence is that the WLAN protocols have usually a higher overhead than their wired counterpart (such as IEEE 802.3). The efficiency of IEEE 802.3 is thus often better than IEEE 802.11.

Why do we use a positive acknowledgment and MAC level retransmission in IEEE 802.11 and not in IEEE 802.3?

As we have seen above, the errors of transmission are more frequent in a wireless environment than in a wired one. It is bounded to the nature of wireless environment. Besides, the CSMA/CA mechanism does not detect collision like CSMA/CD. The main problem is that the upper layers like TCP do not like very well when a lot of packets are lost at MAC layer. TCP believes really that there is congestion and the congestion avoidance algorithm implemented at TCP layer slows down significantly the throughput. Because of this, TCP does not use all the available bandwidth. Implementing retransmission at MAC layer allows detecting packet losses and so TCP sees a reliable channel and has no reason to slow down. However, implementing an ACK mechanism at MAC layer causes also some problems. We will see this in chapter 5.

1.4.3 Power consumption in infrastructure network

Wireless devices are not only made to hide wires on desks of offices, but their main objective is to allow mobility. That is why the devices used in WLAN are often portable. It is not rare to find laptops, handheld devices and embedded[3] systems in WLAN. All the mechanisms that we have just seen in previous sections are implemented especially to overcome the difficulties of transmitting in the air. Nonetheless, if we want to be free with our portable devices, we want it for the longest time and without depending on the power supply. As a consequence, it poses the problem of power consumption of portable devices.
IEEE 802.11 defines a mechanism to save the power of portable devices. It needs the cooperation of the portable device that wants to save power and the access point. Even if it is also possible to save power in ad-hoc networks where an access point is not present, the power saving is not as important as in infrastructure networks. We will not describe power-saving in independent networks.

[3]An embedded system is a special-purpose computer system built into a larger device. An embedded system is typically required to meet very different requirements than a general-purpose personal computer. Personal Data Assistant (PDA) and cellular phones are examples of embedded systems. From `http://www.webopedia.com/TERM/E/embedded_system.html` and `http://www.wikipedia.org/wiki/Embedded_systems#Examples_of_embedded_systems`

How can we save power?

Stations can maximize their battery life by shutting down the radio transceiver and sleeping periodically. For that, the station uses the power management bit (see section 1.4.2) to indicate to the access point that it is in power-saving mode after the current frame exchange and no longer in active mode. During power-saving mode, the station is shutting its wireless interface and during these sleeping periods, access point buffers any frames for the sleeping station. These frames are announced by subsequent special access point frames, the beacon signals. To retrieve buffered frames, the newly awakened station uses PS-Poll frames (see section 1.4.2) and send them to the access point. In fact, PS-Poll frames indicate that a power-saving mobile station has temporarily switched to an active mode and is ready to receive a buffered frame.

The beacon signal

A beacon signal (broadcast signal) is a frame sent by the access point to each station of the BSS at regular intervals (often 100 milliseconds but it can be set). Beacon frames carry information about the BSS parameters. In fact, beacon frames are regarded as management frames (see section 1.4.2). They allow mobile stations to find and identify a network, as well as match parameters for joining the network. This data is therefore important and so all mobiles must listen to beacons. In other words, mobile station wireless interface must be "on"to receive these frames. The access point profits of this to store information in this beacon signal to announce to each station wether there are buffered frames or not at the access point. The mobile stations do not thus spend energy to poll periodically the access point. Indeed, they spend less energy to power up the transceiver to listen to the beacon signals than power up to transmit polling frames. If a station learns that there are buffered frames in the access point that are intended to it, it sends the control PS-Poll frame to the latter to advertise that it is ready to receive a buffered frame.

Traffic Indication Map (TIM)

Inside a beacon signal, a map, which is called Traffic Indication Map, notifies stations that they have at least one buffered frame. The more the number of stations with buffered frames, the bigger the map. Nevertheless, if the number of stations with pending frames is not high, the network capacity is saved.

Listen interval

When a station joins a BSS, the information that it must provide is the listen interval. The listen interval defines the number of beacon periods for which the mobile station chooses to sleep. In other words, when the mobile station is in power-save mode, the access point engages to keep the buffered frames during this listen interval. If the mobile station fails to check for waiting frames after each interval, they may be discarded without notification. Listen interval is therefore a key parameter: if the listen interval is high, the frames will remain longer in the buffer of the access point with the risk that it overruns but more power saving can be achieved.

Additional rules

It is worth noting that:

- Each PS-Poll frame allows retrieving only one buffered frame and this last frame must be positively acknowledged before it is removed from the buffer.
- If there is more than one buffered frame (multiple frames), the "more data bit"in the frame control field (see section 1.4.2) is set to 1. This special bit is used by the access point to say to the mobile station that there is still at least one buffered frame. Mobile stations can then issue additional PS-Poll requests to the access point until the "more data bit"is set to 0. Once all the traffic buffered for a station is delivered, the station can resume sleeping.
- When multiple stations have buffered frames, all stations with buffered data must use the random backoff algorithm before transmitting the PS-Poll. The stations that have not access to the medium must stay awake during their contention window (backoff timer). They continue to receive the beacon signal and if they note that there are no more buffered frames because of discarding (the buffer inside the access point is out of run for example), they can return sleeping.

There are two distinct situations:

1. When an access point receives a PS-Poll frame, it can transmit a buffered frame immediately after a SIFS period. The station can return sleeping when it receives from the access point a frame that carries the "more data bit"set to "0"or when it receives a beacon signal in which the access point announces thanks to the TIM that there are no more buffered frames.
2. However, the access point can choose to defer the sending of the buffered frame instead of an immediate delivering of this buffered frame to the mobile station. The access point transmits instead an acknowledgment, after a SIFS period. In this case, the station knows that it receives the buffered frame at some point in the future and so the station must stay awake until it receives a beacon signal where it is no more designed in the TIM.

Multicast and broadcast frames

Buffering of multicast and broadcast frames are identical to the unicast case, except that frames are buffered whenever any station associated with the access point is sleeping. The stations do not send any PS-Poll to retrieve the buffered frame because this frame is not for a station but for a group of stations. Multiple buffered frames are transmitted in sequence; the more data bit in the frame control field indicates that more frames must be transmitted. A new parameter specifies the number of beacons intervals before transmitting broadcast and multicast frames: Delivery Traffic Indication Map (DTIM). Just after sending this special TIM, the access point sends the broadcast and multicast frames. The standard IEEE 802.11

is not clear about that at such point that a network administrator can configure stations to sleep for its listen periods without regard to DTIM transmissions. In this case, stations miss all broadcast and multicast frames but more power is saved.

1.5 Conclusion

We have seen in this chapter the traditional wired IEEE 802.3 network and the new wireless reference network: IEEE 802.11b. The comparisons showed us that the IEEE 802.11 networks must overcome a lot of difficulties in order to transmit data "properly". This results in an increasing complexity at the MAC and Physical layers (CSMA/CA, a new layer (PLCP), positive acknowledgement,...). In spite of that, mechanisms cannot assure a reliability as high as in the case of wired network. However, the wireless network allows mobility and envisages the power consumption constraints of the mobile devices by defining a low-power mode. In the next chapter, we present such a system: the iPAQ 3600 series equipped with a wireless LAN Cisco Aironet 350 series card.

Chapter 2

Overview of an embedded system: the iPAQ 3600

2.1 Introduction

We have introduced in the previous chapter the definition of an embedded system. We have also said that PDA was an example of an embedded system. Restricted at the beginning with an evolved diary, PDAs have extended their functionalities to become genuine PC multimedia. We will see through this chapter a particular and well-known advanced PDA: the iPAQ 3600 equipped with a PC Card Cisco Aironet 350 series Client Adapter. Figure 2.1 shows what the iPAQ looks like.

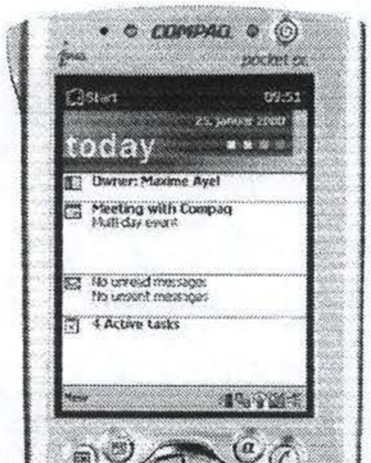

FIGURE 2.1: The Compaq iPAQ Pocket PC 3600

The market of PDAs has known a great expansion during the last few years. The reason is mainly that one manages today to offer PDA increasingly small, less and less expensive and increasingly powerful. They have inherited the fall of the prices of 32-bit processor and memories at such point that the devices have now the storage capacity needed to take advantage of the many advanced features applications require. The popular Compaq iPAQ handheld computer is a good example of such a system. It was the first to offer as many multimedia capabilities (and even more) as its main competitor, the Palm Pilot[1]. In fact, there exist three or four PDA families but they can be categorized as: Pocket PC family and the Palm family. iPAQ is a Pocket PC and runs the Pocket PC/WinCE (formerly Windows CE) operating system from Microsoft while Palm Pilot has Palm OS operating system.

iPAQ PDA became a real reference and it is often selected in many scientific projects or open source projects that encourage the creation of open source software for use on handheld computer[2]. This is undoubtedly also due to the fact that Compaq is contributing considerable resources to making Linux work on the machines. Compaq hosts and sponsors the **`http://www.handhelds.org`** site, which is a community resource center, primarily for developers. The site focuses on Linux under StrongArm SA11xx processors, such as the iPAQ. In fact, it is possible to replace the Pocket PC operating system with the Linux operating system. This open source operating system allows the open source community working on handheld devices to improve the quality and the reliability of softwares, operating systems and standards specifically developed for handheld devices.

The iPAQ Pocket PC has been invented by Compaq[3] in November 1999. Since Hewlett Packard[4] (HP)'s acquisition of Compaq, the product has been marketed by HP. They have developed since this date a complete PDAs 3x00 series : the 3100, 3600, 3700, 3800 and 3900 series. They distinguish each others simply by the performance of their components but keep the same physical packaging.

Here are the specifications of the iPAQ 3600:

- 206MHz StrongArm SA-1110 processor
- 320x240 resolution color TFT LCD
- Touch screen
- 32MB SDRAM / 16MB Flash memory
- USB/RS-232/IrDA connection
- Speaker/Microphone
- Lithium Polymer battery

[1]http://www.palm.com/us/

[2]A handheld computer is a synonym of a personal digital assistant (PDA). Handheld means that the computer can conveniently be stored in a pocket and used when we are holding it.

[3]`http://h18000.www1.hp.com/`

[4]`http://www.hp.com/`

- PCMCIA Card Expansion Pack and CompactFlash Card Expansion Pack

2.2 StrongArm SA-1110 processor

There are a lot of different existing architectures in the handheld world contrary to the world of the Personal Computer (PC), which is limited to just a few competing architectures, chiefly Intel's x86, and the Apple/Motorola/IBM PowerPC, used in the Apple Macintosh. It means that PDA market is still new and emergent. We focus here on the StrongArm SA-1110 processor, which is optimized for meeting portable and embedded application requirements.

The SA-1110 was chosen for its low power and high performance. Indeed, the SA-1110 incorporates a 32-bit processor capable of running at up to 206 MHz. The SA-1110 has also three power management features in order to save power: normal (full on), idle (power-down) mode and sleep (power-down) mode. In normal mode, the CPU and the peripherals are fully powered while in idle mode the CPU is disabled but peripherals can be still active. During the sleep mode, all are disabled and so power saving is high but it takes more time to wake up. In addition to that, the SA-1110 may be run at a variety of frequencies, ranging from 39 MHz up to 206 MHz. We will see in the next chapter how a processor can save power when it can run at different frequencies.

Sa-1110 have several built-in controllers like a controller to manage several types of memories, a LCD controller and an integrated two-slot PCMCIA controller. We will see in the next sections what LCD and PCMCIA mean.

2.3 Memory

In personal computers, it is frequent to meet hard disks (HD) as non-volatile memory. In embedded systems, it is rarer because HD is in fact not recommended for embedded applications. Hard disks are encumbering, noisy, sensitive to the shocks, to the temperature, to electromagnetic fields and greedy in energy. Moreover, if there is a cut in the power supply during a writing, it could be disastrous for the integrity of data. We can thus easily understand that the hard disk is not convenient for battery-operated embedded systems.

Instead of an hard disk, the iPAQ 3600 contains 2 Flash memory chips on board. The Flash memory is a compact memory, quiet, resilient to shocks, radiations and low energy oriented. Flash memory is thus suitable to portable devices like the iPAQ, which have limited energy source from batteries. However, this type of memory does not have only advantages. Flash memory is in fact The "write-once"and "bulk-erasing". Because flash memory is write-once, the existing data cannot be overwritten directly. Instead, the newer version of the data will be written to some available space elsewhere. The old version of the data is then invalidated and considered as ŞdeadŤ. The latest version of the data is considered as ŞliveŤ. As a result, physical locations of data change from time to time because of the adoption of the out-placeupdate scheme. A bulk erasing could be initiated when flash-memory storage systems have a large number of live and dead data mixed together. A bulk erasing could involve a significant number of live data copyings since the live data on the erased region must be copied to somewhere else before the erasing. That is so called garbage collection to recycle the space occupied by dead data. Writing a Flash memory is thus often a slow operation. The block in Flash system has a size of 256KB (while the traditional sectors of an hard disk is of 4KB-size). In addition, the number of writings in Flash memory is limited (about 100000 before the block fails). Flash memory is also expensive compared to hard disk and finally, like the hard disk, it does not like cuts of electricity.

However, the JFFS2[5] file system is specifically adapted to Flash memory. In fact, it offers a *wear leveling* mechanism that tries to spread the erase and write operations evenly across all sectors of Flash. JFFS2 performs Flash erase/write/read on the sector better than other file systems. This file system does never rewrite data at the same place and keeps the previous versions of the file. This technique is thus perfectly appropriate in the case of a sudden interruption of the iPAQ, because we have always a coherent version of the file. It allows keeping the Flash memory usable for a long time. The only drawback of this JFFS2 is that it tends to slow down a great deal when the filesystem is full or nearly full.

Like in PC, the iPAQ 3600 has also a volatile memory. The iPAQ 3600 offers a capacity of 32 MB of SDRAM[6].

2.4 Display

The iPAQ offers a color thin film transparent liquid crystal display. A TFT LCD (Thin Film Transparent Liquid Crystal Display) is a standard display device for all kinds of portable systems, such as laptop computers and PDAs. The iPAQ 3600 LCD display offers 4096 colors. The user can interact with the iPAQ through the screen with a stylus: it is a touch screen.

2.5 PCMCIA

One of the most interesting characteristic of the iPAQ is that it is possible to acquire a Card Expansion Pack that allows connecting a Personal Computer Memory Card International Association-compliant (PCMCIA)[7], also called more recently PC cards. The figure 2.2 shows a Card Expansion Pack and how it can be combined with the iPAQ in order to offer two PCMCIA interfaces.

[5]JFFS2 stands for Journaling Flash File System Version 2

[6]SDRAM (synchronous DRAM) is a generic name for various kinds of Dynamic Random Access Memory (DRAM). Because its definition exceeds the scope of this thesis, we may simply consider SDRAM like the main read-write memory of the iPAQ.

[7]http://www.pcmcia.org/

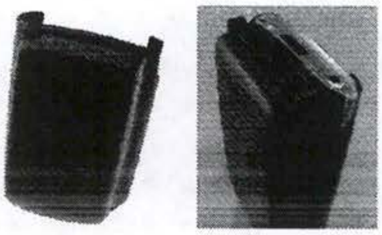

FIGURE 2.2: A Card Expansion Pack on the left and on the right, the iPAQ is plugged into the Card Expansion Pack.

The range of peripherals available in this form factor is interesting, and includes Ethernet adapters, wireless LAN interfaces, Flash memory, modems, speech coprocessors, and more. In addition, a follow-on standard which is a subset of PCMCIA, known as CompactFlash, provides similar functionality in a form factor smaller than a book of matches. The SA-1110 can support up to two slots of either type. Figure 2.3 shows two CompactFlash cards and one PC card adapter that allows connecting a CompactFlash card to a PCMCIA interface.

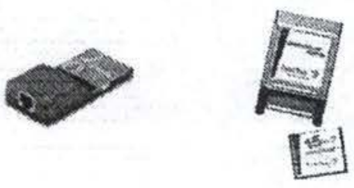

FIGURE 2.3: A CompactFlash Ethernet card, a PC card adapter and a CompactFlash (Flash) memory

2.6 The Cisco Aironet 350 card

Thanks to the Card Expansion Pack and the support of PCMCIA cards by the SA-1110 architecture, it is possible to add a wireless PCMCIA card in the iPAQ. The wireless PC Card Cisco Aironet 350 series Client Adapter can be thus directly plugged into the Card Expansion Pack. The figure 2.4 is a picture of the PC Card Cisco Aironet 350 series Client Adapter.

Based on direct sequence spread spectrum (DSSS) technology and operating in the 2.4-GHz band, the Cisco Aironet 350 Series Client Adapter complies with the IEEE 802.11b standard. It supports several data rates: 1, 2, 5.5 and 11 Mbps. The Cisco Card utilizes Carrier Sense Multiple Access With Collision Avoidance (CSMA/CA) protocol to access the medium. The transmit power can be set to different values from 1 mW to 100 mW (default value). Having a high transmit power will help to emit signals stronger than the interferers in the band (and other systems) but having a high transmit power will drain the batteries faster. Moreover, if we want to put many different networks in areas close to each other, they tend to pollute each other if we have defined a strong transmit signal. With less transmitted power you can make smaller cells - smaller networks and avoid interferences between them. This is why Cisco 350 card proposes to select different transmitted powers.

Another important information about wireless card is the sensitivity values: it is the measure of the weakest signal that may be reliably heard on the channel by the receiver. In reality, lower the value better the hardware (i.e., it is able to read the bits from the antenna with a low error probability). The problem is that all manufacturers and standards do not use the same reference. The 802.11 specifies the sensitivity as the point when the system suffers from 3% of packets losses while some products use 50% packet losses as the definition of sensitivity, which of course gives a better number. Comparisons are thus not easy to make.

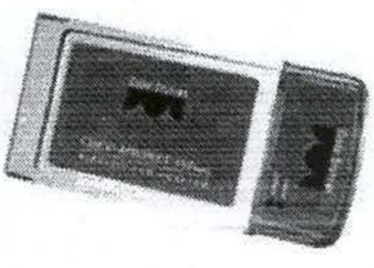

FIGURE 2.4: The Cisco Aironet 350 Card

As we have seen in the first chapter, IEEE 802.11 compliant card must offer at least two modes of functioning: a normal mode and a low-power mode. The Cisco card provides in fact three modes that are described in the following:

- CAM: it is the mode for machines where power consumption is not an issue. It keeps wireless powered up all the time, so there is little latency for responding to messages. This mode is recommended for devices where high availability is desired.
- PSP: it is recommended for devices where power consumption is a major concern. It causes the Access Point (the wireless device you interface with) to buffer incoming messages for your wireless client, which must wake up periodically and poll the Access Point to see if there are any buffered messages waiting for it. The wireless client can request each message and then go back to sleep. It corresponds to the low-power mode defined in IEEE 802.11 standard.
- PSPCAM: it is a blend of CAM and PSP modes. PSPCAM mode switches between PSP and CAM based on network traffic. When receiving a large number of packets, PSPCAM Mode will temporarily switch to CAM mode to retrieve the packets. Once the packets are retrieved, it switches back to PSP mode. This mode is a specific Cisco mode and thus it is not defined in IEEE 802.11 standard.

These different modes can be defined at any time by the user through client utilities, available for a lot of platforms (Windows, Linux,...). In fact, client utilities are programs sold with the card that allow configuring and managing the Cisco Aironet card by giving commands to the hardware through its driver.

2.7 Conclusion

The iPAQ is a complete and powerful system whose the functionalities can be very easily extended thanks to the Card Expansion Pack. It becomes thus a real mobile handheld device "IEEE 802.11-compliant"once equipped with the Cisco Aironet wireless card.
A color display, a 206 MHz CPU and a wireless interface are of course power-consuming components. We have just seen until now how the power issue is addressed in commercially available network interfaces through the Cisco Aironet 350 PC Card. We will see in the next chapter how the power consumption problem can be handled and how we can reduce more power in embedded systems by the overview of the Dynamic Power Management (DPM).

Chapter 3

Dynamic power management

3.1 Introduction

One of the most important technical evolution of the last decade has been the emergence of portable systems, such as laptop computers, cellular phones and PDAs. The increasing popularity of such systems encourages the development of more and more sophisticated devices. Designing a portable system requires to tackle about the problem of delivering high performance with a limited consumption of electric power. High performance is required to support complex applications (for instance multimedia) that are running on these portable systems. Low-power consumption is required to achieve acceptable autonomy in battery-powered systems, as well as to decrease battery weight. Stationary systems[1] are also concerned with power conservation, because of the cost and noise of cooling systems, the cost of electric power (especially for large systems) and stricter environmental impact regulations.

The battery capacity has improved very slowly while the demands of computation capacity have drastically increased over the same time. Better low-power circuit design techniques have helped to increase battery-lifetime, but benefits obtained by such techniques do not compensate all the needs. In addition, the pressure for fast time-to-market has become extremely high, and it is often unacceptable to completely redesign a system merely to reduce its power dissipation.

Electronic systems are generally designed to deliver peak performance, but in many cases peak performance levels are not needed for most of the operation time. Cellular phones and portable computers are two examples of systems with non-uniform workload. When the user is sending or receiving a call with a cellular phone (or is running an application on a laptop computer), he wants to have the maximum performance. However, when the user is carrying the phone in his pocket (or is thinking to what to write during a text-editing session on a laptop computer), he does not need the full computational power of the system.

Power management for computer systems is traditionally focused on regulating the power

[1]i.e. non portable systems.

consumption by switching the system in a low-power state, which does not allow to use the system. This state is a de-activating state, generally requiring a user action to re-activate the system. More recently, some researches focus on the development of power management techniques performed while programs are running.

Dynamic power management (DPM) is a design methodology that dynamically reconfigures an electronic system to provide the requested services and performance levels with a minimum number of active components or a minimum load of such components. Dynamic power management encompasses a set of techniques that achieve energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are idle (or partially unexploited). A component is in an *idle* state if it has no request to serve; it can then be put in a *sleep* state to reduce its power consumption. When a request arrives, the component wakes up and switches into a *run* state in order to serve this request[2]. Moreover, a component can be completely shut down in order to do not consume power. The component is thus in the *off* state.

The fundamental premise for the applicability of DPM is that systems (and their components) have to support a non uniform workload during their operation times. Such an assumption is valid for most systems, both when considered in isolation and when inter-networked. A second assumption of DPM is that it is possible to predict, with a certain degree of confidence, the fluctuations of workload. Finally, a third assumption is that the workload observation and prediction should not consume significant energy.

We examine hereafter the three levels where dynamic power management (DPM) can be applied: first at component level, then at system level and finally at network level.

3.2 Dynamic power management at component level

We can see a system like a set of interacting components. The definition of a component is general and abstract. It may be a chip (such as the CPU) or a board (hard disk, memory, wireless interface, video display,...) but in the current context, it is a black box: no detailed knowledge of its internal structure is needed.

A power manageable component (PMC) is characterized by multiple states of operation that span the power-performance trade-off. This allows to distinguish these components from those always operating at a given performance level and power consumption. The ideal case for a PMC is always to have a lot of states of operation in order to minimize the power by calibrating perfectly the performance needed by the requests served by the component. Nevertheless, a problem appears when the number of states increases because the hardware complexity and overhead become more pronounced. In fact, the PMC's cannot switch from a state to another one without a cost. The cost may be a performance lost, a delay or even a power transition

[2]Other names for these states are sometimes used in the literature, such as active, disable, on, work,....

cost. The relation that we often note is: low-power states (such as sleep) have lower performances and larger transition costs, with respect to states with higher power. The transition cost has an important impact and has to be taken into account in the power management models.

3.2.1 Power state model

We can thus define for each component a power/performance behavior. In other words, it can be defined for each PMC the power states it accepts (for instance run, off, sleep, and idle), the associated performances, the transition costs and the power consumptions when it switches from one power-state to another one. We can represent this information with a power state machine for example (see figure 3.1). To summarize, we can say that a PMC is needed to

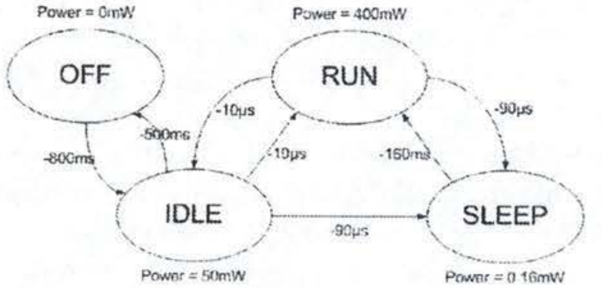

FIGURE 3.1: Example of power state machine.

develop some power management techniques and that a component cannot switch from one of its state to another one without a cost. This leads us to consider the power management model as a non-trivial optimization problem.

3.2.2 Under the black box

We have defined a component as a black box. It is time to see now how DPM is implemented inside the *box*. We examine in this section two of the most dynamic techniques implemented in components themselves: the ***clock gating*** technique and the ***power off supply*** technique. In reality, these techniques are generally implemented in the hardware circuit of the component. That is why they are also called physical mechanisms.

One of the most common DPM techniques at the component level is the clock gating. This addresses digital components that are clocked (CPU, display,...). Power can be saved by reducing the frequency of the component clock (or some component clocks, if the component uses several clocks), and at the limit by stopping the clock. Clock gating can be applied both during idleness periods or activity periods. During activity periods, slowing the clock decreases the performance and, for some components (like the CPU), extends the execution time to perform a task. In this case, power saving is generally weak. For this reason, power saving can be more important during idleness period. The main challenge is then efficient idleness detection. Moreover, to go back to normal system activity, clock gating requires a short transition time: the clock should be re-initialized in one or a few clock periods.

Even by stopping the clock, power dissipation is not completely eliminated. Power consumption of an idle component can be avoided by the technique of powering off the component. This radical solution requires controllable switches inserted in the electrical line supplying the component. A major disadvantage of this method is the recovery time, which is typically higher than in the case of clock gating because the component's operation must be re-initialized.

3.2.3 Internal controller and external controller

We have said above that the transition latency can have an important impact on the power management model. In reality, when it is possible to switch a component to a *sleep* state without compromising the performance (or with little performance degradation) because the transition between the *sleep* state and the *run* state is nearly instantaneous, an *internal controller* - internal to the component - can be implemented. This internal controller decides for example to decrease the frequency (clock gating) or shut down the component (supply shutdown) if there are no requests to serve. Internally managed components are also called self-managed components. The main drawback is the lack of observability of the overall system operation and of the need of tolerating little or no performance degradation, since no assumptions can be made on how demanding the component's environment will be.

Clock gating is often implemented in a component with an *internal controller* while supply shutdown will be rather implemented with an *external controller*. The difference between the two techniques resides in the transition time to switch the component into its normal state. Indeed, when power transitions take a long time or consume a lot of energy, it is needed to take into account the workload of the system and decide when it is worthwhile to switch to a low-power state. Otherwise, the performance could be disastrous. The example that we have in mind is the limit case of workload with no idle periods longer than the time required to enter and exit the *sleep* state. If we decide to shut down the component as soon as an idle period is detected and if the power consumption associated with state transitions is of the same order of that of the *run* state, this could reduce the performance without saving any power. Workload information is thus required for all *advanced* power management approaches. The goal is even to predict the workload to identify exactly the idle periods. Several approaches and models can be produced to capture the workload information. Diverse techniques based on predictive approach or stochastic control have already been developed. We can easily imagine that these models are very complex. It is thus the role of an *external controller* - external to the component - to control transitions based on the workload of the system.

Let's see now how an external controller can be implemented at the system level by a power manager.

3.3 Dynamic power management at system level

3.3.1 Structure of a system-level power manager

The power management idea is to profit from idle periods (e.g., when the component is not used) to put the component in its *sleep* state, or in one of its *sleep* states if multiple low-power states are available, without compromising too much the performance of the whole system. Regarding the system, the assumption made here is that it is not always entirely active; this means there are some periods during which some components are idle.

The activity of components (PMC and also non-PMC) is coordinated by a *system controller*, which is generally implemented in a software routine of the operating system. That is why the control of the system consumption is also implemented in software and especially in operating system as a module of the system controller.

We call *power manager* (PM) the system part (hardware or software) that performs DPM at system level. A power manager is composed of an *observer* monitoring the workload of the system and its resources, a *controller* issuing commands for forcing state transitions in system resources, and at least one *policy*, which is a control algorithm for deciding when and how to force state transitions (based on information provided by the observer). The figure 3.2 depicts the general structure of a power manager.

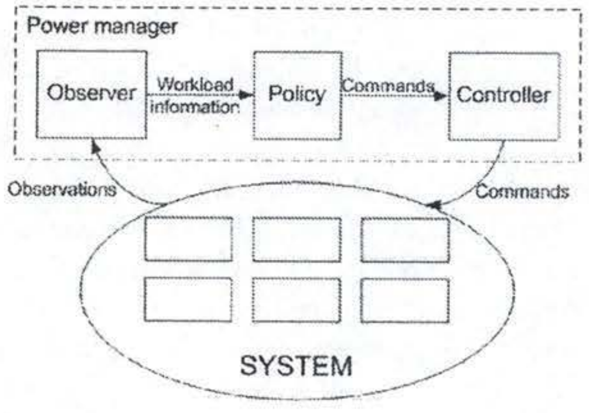

FIGURE 3.2: Structure of a system-level power manager.

There are some reasons for implementing the power manager at software level. Software power managers are easy to write and to reconfigure. Then, in most cases, the designer cannot or does not want to interfere with and modify the underlying hardware platform. Finally, as DPM implementations are still a novel art, the experimentation with software is easier than with hardware.

3.3.2 Power manager controller

The power manager must be able to control the state of each PMC. In order to develop a power manager as generic as possible, it should be interesting to dispose of an interface between the power manager and a PMC. This interface should be able to understand each command from the power manager and to effectively perform it. The way to control the state of a component is very hardware dependent. For this reason, the interface should be implemented in the component-closest software part, which is generally the *driver* of the component.

Industrial designs have been also proposed to encourage the standardization of interfaces. For instance, Intel, Microsoft and Toshiba propose a standard called *advanced configuration and power interface*, in short ACPI. ACPI is an OS-independent general specification that defines the interface between the operating system and an ACPI-compliant hardware. This interface can be used both for hardware configuration and power management. The front-end of the ACPI is the *ACPI driver*, which is OS-specific. The OS kernel interacts with hardware through the ACPI driver, which maps each OS request to an ACPI command and each ACPI response/message to a signal/interrupt (see figure 3.3).

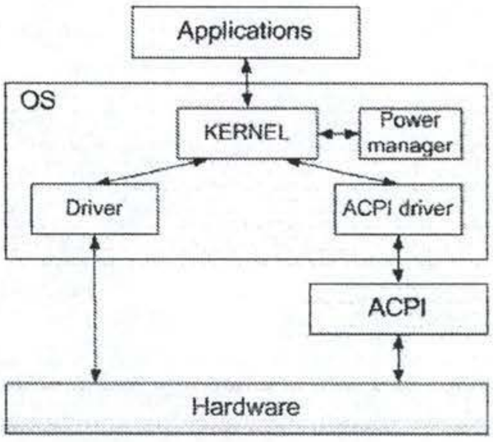

FIGURE 3.3: ACPI interface.

The ACPI allows the operating system to put the system in five possible *global power states*:

- *Working*: The system is ON and fully usable.
- *Sleeping*: The system appears to be OFF and the power consumption is reduced. The system returns to the working state in an amount of time inversely proportional to the power consumption.
- *Soft off*: The system appears to be OFF and power

  consumption is very low. A full OS reboot is needed to restore the working state.
- *Mechanical off*: The system is OFF, with no power consumption. It needs to be reconnected to the power supply to go back to the working state after a full reboot.

- *Legacy*: This state is entered when the system does not comply with ACPI.

Additionally, the ACPI specifications define power states for system components. There are two types of system components, *devices* and *processor*, for which four power states are defined in the Table 3.1:

	Device power states		**Processor power states**
D0	This state has the highest level of power consumption. The device is fully active.	*C0*	The processor is fully operational and executes instructions.
D1, *D2*	The details of these states are device dependent. A device in D1 is expected to save less power than in D2, but it preserves more context (hence, wake-up is faster).	*C1*, *C2*	The processor is not executing instructions. The processor in C1 is expected to save less power than in C2, but switching from C1 to C0 is performed in a negligible time.
D3	Power has been fully removed from the device. The device cannot be used in this state and has the longest restore time. The OS will re-initialize the device when powering back on.	*C3*	This state offers improved power savings with respect to C2. The hardware latency to resuming execution is larger than that in C2.

TABLE 3.1: Devices and processor power states

3.3.3 Power manager observer

A power manager needs information to predict future workloads. Two extreme approaches allow to collect this information. In the first approach, the power manager observes the requests soliciting the managed component and try to predict the future idleness length, and on this basis determines the best power state. It selects the power state without direct interaction with requesting application. The figure 3.4 depicts this approach.

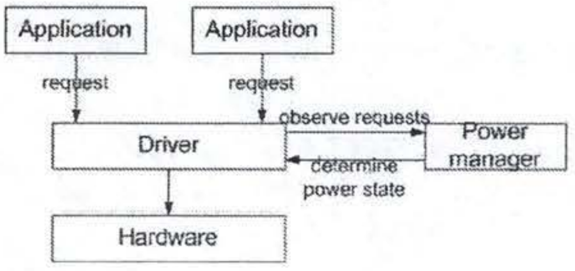

FIGURE 3.4: The power manager observes requests.

In reality, however, requests may be generated by multiple requesters. For example, requests for a network interface card may come from different programs (such as ftp, telnet or netscape). These programs work differently and have different performance requirements. Without information about requester programs, the power manager cannot precisely predict the exact moment when the component is not used anymore and thus wastes energy to maintain performance unnecessarily or cause delays and performance waste to save power.

At the other extreme, requester programs directly control power management through an application programming interface (API). For example, programs can keep a component in a run state, wake up a component in the sleep state or know the current state of a component. The figure 3.5 describes this approach. The main disadvantage of this approach is that

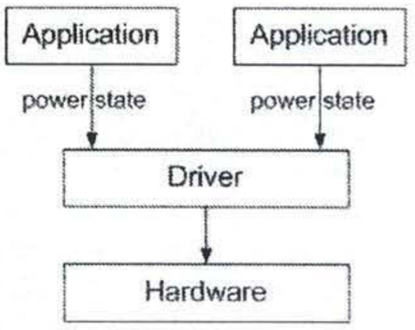

FIGURE 3.5: Applications control power states directly.

programs must include specific power management instructions. Moreover, programs do not know very well the components, especially the transition costs of each of them. They are not aware if changing state saves power.

Other solutions have been proposed, like in [24], which are generally situated between the two previous extreme approaches. Generally, in these other solutions, programs provide to the power manager an information about their needs of components. The power manager decides, according to the programs needs and its knowledge of the components transition costs, to put some components in sleep state if their are idle for a sufficient long period.

3.3.4 Power manager policies

A power management *policy* is an algorithm that selectively shuts down idle resources based on the observation of present and past workload and operating conditions. We survey here two different approaches to policy optimization, the *predictive techniques* and the *stochastic control.*

The rationale in all *predictive approaches* is to take DPM decisions based on predictions concerning the duration of idle periods. A generic predictive method observes the time-varying workload, and, based on this observation, computes a predicted duration T_{pred} of the upcoming idle time. The power manager then decides the transition to the sleep state if $T_{pred} \geq T_{BE}$, where T_{BE} is the *break-even time*, the minimum idle time amortizing the state transition cost.

Good predictive approaches should minimize the time for which the system wastes power because it does not immediately detect the beginning of an idle period. They also should minimize the mis-predictions $T_{pred} \neq T_{idle}$, where T_{idle} is the actual duration of an idle period. Predicting a too small T_{pred} ($T_{pred} < T_{idle}$) wastes power, while a too long T_{pred} ($T_{pred} > T_{idle}$) decreases performance.

The most common predictive PM policy is the *fixed timeout*. The policy can be summarized as follows: when an idle period begins, a timer is started with a certain duration T_{T0}. If after T_{T0} the system is still idle, then the PM forces the transition to the sleep state. The system remains in sleep state until it receives a request from the environment that signals the end of the idle period. The fundamental assumption in the fixed timeout policy is that the probability of T_{idle} being longer than $T_{BE} + T_{T0}$, given that $T_{idle} > T_{T0}$, is close to one. Hence, this policy assumes that if the idleness duration is longer than the timeout duration ($T_{idle} > T_{T0}$), the idleness duration will continue a sufficiently long time to allow the system to save power by changing the power state ($T_{idle} > T_{T0} + T_{BE}$). Timeouts are "implicitly" predictive even if they never generate an actual T_{pred}, in the sense that they predict a long idle time if the system has been idle for a while. The critical design decision is obviously the choice of the timeout value T_{T0}.

Timeouts have three main limitations: fixed timeouts may be ineffective when the workload is non-constant. Moreover, power is wasted while waiting for the expiration of the timeout. Finally, performance penalty is always paid upon wake-up. *Adaptive timeouts* have been developed to improve effectiveness, by dynamically reducing/increasing the timeout value when idleness duration increases/decreases. The figure 3.6 illustrates the timeout-based policies.

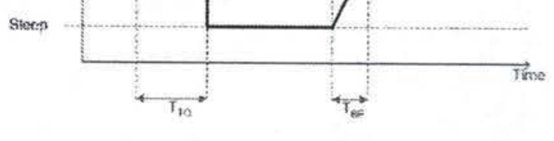

FIGURE 3.6: Timeout-based policy.

Another predictive policy consists of shutting down or putting in sleep state the system or a component as soon as it becomes idle, if the policy predicts $T_{pred} > T_{BE}$. A prediction of idle time duration is made available as soon as the idle period begins. Predictions are made based on past idle and activity period durations. The system wakes up either only upon arrival of a request from the environment, or at the end of the predicted idle period. The ***predictive shutdown policy*** is illustrated in the figure 3.7.

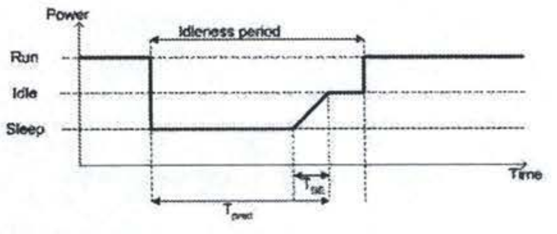

FIGURE 3.7: Predictive shutdown policy.

Predictive approaches have some limitations: first, they do not precisely take into account the workload variations. Then, predictive algorithms are based on a two-state system model, while real-life systems have multiple power states. Policy involves not only the choice of when to perform state transitions, but also the choice of which transition should be performed.

The ***stochastic control*** approach considers the workload as a ***Markov chain*** of R states. A Markov chain is a probability function of the evolution of the system state, according to the state at present time. A Markov chain defines probabilities for the system to enter into a state at a moment $t + 1$, according to the system state at the moment t. Thus, the workload can be represented by a two-state Markov chain, where the two states are: $R0$ when no request is issued by the environment and $R1$ when a request is issued.

In the same way, power states are also represented by a Markov chain of S states. The transitions are probabilistic, and probabilities are controlled by commands issued by the power manager. A power manager is then a function $R \times S \rightarrow A$ where A is a command to control the future state of the system. Such function is an abstract representation of a decision process: the PM observes the power state S of the system and the state R representing the workload, takes a decision, and issues a command A to select the next state of the system.

3.4 Dynamic power management at network level

In many cases, the systems are not isolated and interact between them. We call network a set of communicating systems. We can thus imagine a power manager that tries to minimize the global consumption of the whole network, rather than the individual consumption of each system. The power manager is so implemented as a ***distributed*** algorithm. This distributed algorithm takes autonomous decisions for each system in the network based either on local information, or on incomplete global network status data.

Currently, the DPM in such networked systems is not really still well developed.

3.5 Examples of dynamic power management techniques

This section describes some examples where dynamic power management techniques are used. Each subsection is focused on one of the main components of a portable computing system. Thus, we present here low-power techniques concerning respectively the battery, the processor, the memory, the hard disk and the video display.

3.5.1 Battery-driven dynamic power management

Battery lifetime extension is a primary design objective for portable systems. Traditionally, battery life-time has been prolonged mainly by reducing average power consumption of system components. An analysis of battery discharge characteristics has allowed to develop new opportunities for life-time extension. [7] and [8] propose a class of policies, whose decision rules controlling the system operation state are based on the observation of both system workload and battery output voltage.

The proposed policies are based on three battery physical properties. First, the effective voltage of a battery decreases as the state of charge decreases. As a matter of fact, a battery is considered exhausted when its output voltage falls below a given voltage threshold (such as 80% of the nominal voltage). The second property is that the actual usable capacity (Number of Amps $\times$ Number of hours) depends on the discharge current. More clearly, in a theoretical battery, requiring more power (for example twice more power) reduces battery lifetime proportionally (for example a reduction of a factor two). In a real battery, the lifetime reduction is more accentuated. The reason is that at higher discharge current, a battery is less efficient to convert its chemically stored energy into available electrical energy. The third property says that a battery can recover some of its deliverable charge if discharge periods are interleaved with rest periods (i.e. periods in which no current is drawn).

The simplest policy is threshold-based. It aims at maximizing battery lifetime by lowering the performance when the battery is almost discharged. If the battery is fully charged, the system is kept in a normal-performance state. When the battery output voltage falls below a threshold, the system is forced into a low-performance and low-power state until the battery is fully discharged. The rationale for this policy is to provide acceptable degradation of system performance as the battery discharges.

Some modern portable appliances can accommodate two batteries in the same case. The batteries are normally used sequentially: The second battery starts supplying the current only when the first battery is totally discharged. As a battery can recover some of its deliverable charge if it is let at the rest, a dual-battery system can alternatively use each battery to draw its energy. In this way, the battery temporarily disconnected from the load can recover, while the other one powers the system. An efficient policy to manage the batteries alternation consists of defining three regions of operation. In the first region, the switching between the two batteries has constant frequency, and the system works in a normal-performance state.

The second region is entered when the output voltage of one battery first reaches a threshold. The system still works in normal state, but switching between batteries is voltage-controlled. When the output voltage of the loaded battery reaches the threshold, it is disconnected from the load (to give it some recovery time). The second region is exited when the output voltage of the battery temporarily disconnected from the load does not reach a level close enough to the threshold during the recovery time. In the third region, the fixed frequency-switching scheme is restored, and the system is transitioned into a low-performance state until both batteries are fully discharged.

The battery-driven policies proposed here, applied in isolation or altogether with another technique(s), allow a very important lifetime extension in some systems.

3.5.2 Power management for the processor: Dynamic voltage scaling

DPM aims at reducing energy consumption at the system level by selectively placing components into low-power states during idle periods. The power manager can completely turn off the component (power off supply) or disable some clocks to do that (clock gating). In the microprocessors, other techniques have been developed to decrease significantly the energy consumption. If clock gating is an efficient technique to reduce the energy in clocked components, ***dynamic voltage scaling*** (DVS) is often an additional technique that can more reduce the power consumption. DVS can be used simultaneously to the clock gating.

We can compare in fact DVS to a kind of DPM at the component level except that DVS is targeting specifically the processor. DVS is close to clock gating because both act on the clock frequency except that DVS does not turn off the clock but changes the frequency of the processor (the speed of the processor) and voltage at run-time, depending on the needs of the running application. Thus DVS do not thus detect idleness period but rather estimate the needs of the running application. When peak performance is needed (e.g. peak computational loads), the processor operates at its normal voltage and frequency (which is also its maximum frequency). During the rest of the time, when the load is lower, the operating frequency is reduced to meet the computational requirements (performance). In fact, once again, it is assumed implicitly that the ***workload*** is not constant and that the average computational throughput is often much lower than the peak computational capacity needed for adequate performance.

DVS algorithms modify the frequency according to the needs of the running application and go beyond that because they scale the operating voltage of the processor along with the frequency. In fact, the vast majority of microprocessors today has a voltage-dependent maximum operating frequency, so that when they are used at a reduced frequency, the processor can operate at a lower supply voltage. In other words, the maximum operating frequency increases (within certain limits) with the increased operating voltage; when the processor is run slower, a reduced operating voltage suffices. The interest of reducing the operating voltage along the frequency is obviously to save energy. There is a second characteristic also shared by the vast majority of microprocessors today, allowing saving energy: the energy consumed by the processor per clock cycle scales quadratically with the operating voltage, so that even a small change in voltage can have a significant impact on energy consumption. Moreover, important energy savings can be got because high performance is needed only for a small fraction of the time, while for the rest of the time, a low-performance and low-power processor would suffice. By dynamically scaling both voltage and frequency of the processor based on computation load, DVS can provide the performance to meet peak computational demands, while on average, providing the reduced power consumption benefits typically available on low-power performance processors.

To decrease the frequency along with the needs of the application, the processor must operate over a range of frequencies. For that, it must be specifically hardware-designed to support dynamic clock frequency adjustment. The number of frequencies is thus predefined by the design of the microprocessor. As we have already pointed it out in previous sections for the different *states* in the DPM model, there is also some overhead here when switching from a frequency to another one. This overhead is also called transition time, like in the DPM model. Nevertheless, even if it depends a lot from the hardware, the transition time overhead is often very short.

If DVS relies on a specific hardware design, we can say that it relies also on software at the system level, or even at the task level. DVS algorithms have in fact two main functions: the first one identifies the needs of the application and the second one adjusts effectively the CPU frequency and voltage. To accomplish the first "mission", the execution time of the application has to be predicted (by analyzing the workload). It is important to note that this execution time can be increased (performance decreases) if the frequency is decreased, but the main challenge is to respect the needs of the running application or, in other words, to keep an *adequate* performance. In fact, "respect the needs" or "keep an adequate performance" can be formulated like the respect of the deadline of a task in a real-time environment where tasks must be completed before a deadline, or like keeping a frame delay constant for a MPEG or MP3 streaming application. There are in fact a lot of models implemented in software (at the system level or at the task level) that try to get accurate information on the *workload* in order to adapt the frequency of the processor on the basis of this information while keeping an adequate performance . We can also define for DVS the notion of *policy*, which is an algorithm that selectively adjusts the clock frequency and voltage of the processor based on the observation of the *workload*.

In fact, we can easily integrate the DVS technique into the DPM model presented in previous sections at the system level. We can consider that the processor *remains* in a *run* state when the DVS algorithm is applied. It allows reducing the energy consumption of the processor when the processor is in a *run* state. Let us remind that, when in previous sections, we talk about the *run* state, the component consumes a lot of energy because it is fully powered on. In reality, we can see here the *run* state like a set of sub-states.

If the model describing the ***run*** state is the same as the one characterizing the other states, the transformation, from the "original" DPM ***run*** state into multiple "new" ***run*** states taking into account the different performance and power levels of DVS, is completely compatible with the rest of the model and so, the power management ***policy*** that we develop can make decisions for both dynamic voltage setting and the transition into the low-power states. For example, in the case of a MPEG streaming application where keeping an adequate performance consists in maintaining the frame delay constant, the power manager can check if the rate of incoming or decoding frames has changed, and then adjusts the CPU frequency and voltage accordingly. Once the decoding is completed, the system enters in an ***idle*** state. At this point, the power manager observes the time spent in the idle state, and depending on the policy, it decides when transiting into one of the ***sleep*** states. When a request arrives for video decoding (after receiving a new frame on the network interface), the power manager switches the system back into the active state and starts the decoding process. That is actually the ratio between the rate of incoming frames and the rate of decoding frames that determines the frequency and the voltage levels of the processor.

It is to be noted that a dilemma appears and can be stated as follow: lowering the threshold voltage to reduce ***run*** power can increase ***sleep*** power. In fact, increasing the execution time of applications by reducing the frequency causes the system to be more often in the ***run*** state than in the ***sleep*** state. In other words, if a system is frequently ***idle***, it could be more interesting to save power by putting the system in the ***sleep*** state instead of decreasing the frequency and the voltage of the processor.

3.5.3 Selective instruction compression for memory energy reduction

We have already seen that power management can be applied to several embedded components. It is time now to see how we can save power in the memory management.

The first developed techniques are bus encoding techniques and memory organization techniques. Both are based on a reduced switching activity on the processor-memory bus. In fact, the first technique changes the format of the information transmitted on the processor-memory bus. Thus, it reduces the switching activity on the bus and so, in the same way, reduces the power consumption. The second techniques change the way information is stored in memory so that the address streams between the processor and the memory cause a low-transition activity on the bus. The processor-memory interface is a major contributor in the power consumption and so, these techniques can be efficient.

Other techniques have been additionally developed through instruction memory bandwidth. These techniques use a set of special instructions which are smaller in size (e.g., contain less bits than normal instructions) and hence, achieve to reduce the bandwidth needed to run the program. Either these special instructions are another set of instructions supported by the processor but in this case it often requires additional software tools to allow users to generate these special machine instructions from the task level, or these special instructions are simply a subset of the original instructions supported by the processor. In this latter case, if we consider only a subset of the original instructions, we decrease the number of potentially instructions used and so, we do not need the entirely bit-width associated to all original instructions. Thus this subset of instructions can be replaced by binary patterns of limited width. For example, if we identify in an application the utilization of 512 distinct instructions among the 8192 instructions offered by the processor, we can then only use 9 bits to encode the 512 instructions (log_2 $512 = 9$) instead of 13 bits needed to encode 8192 distinct instructions (log_2 $8192 = 13$). The 9-bit instructions take less place in memory than 13-bit instructions. It thus reduces the memory bandwidth usage and in consequence the total energy because the unused memory banks can be disabled.

We suppose in fact that the processor can disable the memory banks that are not currently used in order to save energy. It is an hardware requirement to save energy. For example, if the memory consists of four 8-bit banks and that we have succeeded in reducing the size of the instructions from 32 to 8 bits, it requires only one memory bank instead of 4. The 3 unused banks can be disabled to save energy. Moreover, if we assume that the memory access is 8-bit wide, the number of fetchings between the processor and the memory passes from 4 to 1: the total bus utilization is thus also reduced, which improves performance (i.e., the program utilization time decreases).

We can simply implement a table whose the role is to match the original instructions of the subset with the compressed instructions. In reality, the program is stored in memory in compressed format, i.e., each instruction is replaced with a $[log_2 N]$-bit binary pattern which is in one-to-one correspondence with the original instruction. Every time an instruction is fetched from the memory, it is first decompressed (i.e, the original format is restored) by means of the ***instruction decompression table*** and then passed to the processor's decoding logic. The advantage of the table is that we do not modify the architecture of the processor. The figure 3.8(b) depicts the solution and especially the ***instruction decompression table*** (IDT). The main drawback of this technique is the energy saving depends on the ratio between the

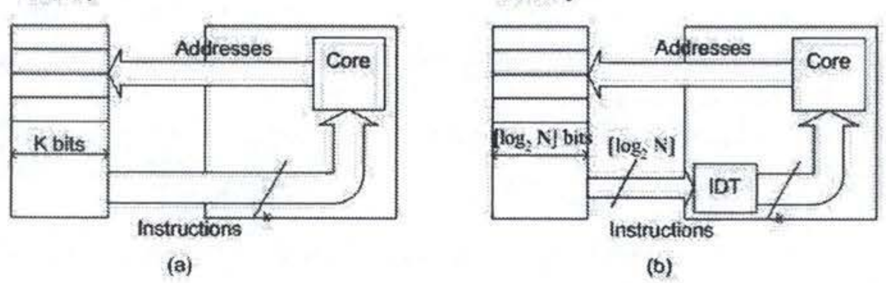

FIGURE 3.8: Processor-memory normal architecture (a) and power-saving architecture (b)

number of different instructions that forms the subset (compressed instructions) and the total number of machine instructions. The more distinct instructions are used, the more there are instructions in the subset and therefore, fewer energy is saved. Indeed, the bit-width of these instructions may become similar to the bit-width of the original instructions, thus making negligible the reduction in memory bandwidth. The second drawback is the implementation of the table because this can be very complicated to encode and decode instructions especially if the size of the compressed instructions is not compatible with the byte-addressable memory. Indeed, usually memories can be only accessed for example by a multiple of 8 bits. If we refer to the example in the previous paragraph where the application uses only 512 instructions among the 8192 instructions supported by the processor, the 9-bit instruction is not compatible with the 8-bit access scheme of the memory. In fact, two bytes are needed to store each compressed instruction and so, it results in a waste of space and consequently in a waste of energy. Moreover, it requires two operations to store this 9-bit instruction in memory and so, it does not decrease the program utilization time.

One solution consists of taking a subset of fixed cardinality. That does not depend anymore on the number of distinct instructions used by each application running but rather on the probabilities of the instructions to be used. In fact, an assumption is indirectly made: the number of machine instructions used by most software programs, although limited with respect to the total number of instructions supported by the processor, has a highly non-uniform statistical distribution. In other words, some instructions are usually much more used than others. We can hence consider a subset of original instructions that are executed more often whatever the applications; less probable instructions are left unchanged and stored as they are in memory. As there are as well non-compressed (few) instructions as compressed (many) instructions in memory, that requires a controller which properly handles instruction fetching. The figure 3.9 depicts the second power-saving architecture in the case where the compressed instruction has 8-bit width.

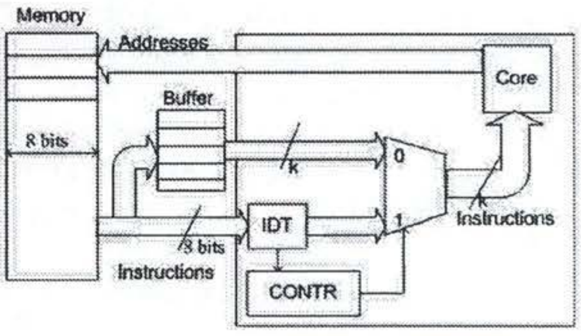

FIGURE 3.9: Processor-Memory second power-saving architecture

The percentage of energy that can be saved depends mainly on the architecture and especially on the way in which the memory is organized and accessed. The technique presented here is indeed a new way of managing and accessing the memory since there are different bit-width instructions. We have not presented all the implementation and changes that this technique required in the memory management. However, it is directly implemented in hardware and so, it is application-independent. It is important to say also that, like in all other power management techniques, there are some *costs* to take into account in this power management technique. In fact, the ***instruction decompression table*** (IDT) consumes a certain time to decode and to encode instructions, and therefore it consumes power. This cost in power and performance of the decompression block is not still well-known and is currently under investigation. In addition to that, to distinguish compressed and original instructions, it is needed to put sometimes some bits in memory and this causes also some overheads in the number of memory accesses, as well as a waste of space. Thus, the costs in term of performance or energy exist also for these techniques aiming at reducing the consumption of energy by the memories. Like in all other power management techniques, there is always a trade-off between performance and power.

3.5.4 Power management for hard disks

The hard disk drive is one of the major power consuming subsystems in a computer, since it can consume more than one fifth of the total power used by the computer. The main way to reduce power consumption of hard disks consists of stopping plates spinning during the periods when no disk requests are made. However, this approach encounters three problems: first, when a request occurs while plates are not spinning, it can not be performed before the plates have taken back a sufficient speed. This delay strongly decreases the performance of the system. Moreover, accelerating the plates generates extra power dissipation. Finally, too frequent on/off cycles tend to accelerate the degradation of the hardware systems, causing a problem of reliability. Now a desirable power management scheme should save energy while providing high performance and low failure rates.

Disks accesses generally occur in burst: the activity of a disk is often characterized by sets of close requests, separated by long idle periods. It can be made use of these idle periods to reduce power consumption. In [25], a method is proposed to do that, on the basis of the concept of *sessions*, which are time intervals when requests frequently occur. A session starts with a disk access, and is separated from the previous one by a long period of inactivity. Requests close in time are regrouped in the same session. A threshold τ is used to separate sessions. If no requests occurs during τ seconds after a request occurrence, the current session ends. The figure 3.10 depicts an example of sessions with different τ.

A first observation is that, during inter-session periods, there are no disk activities, and then the system can spin down the plates to save power. Moreover, smaller τ may separate adjacent requests into two sessions, while a larger value can regroup them into one. Hence, a large τ allows to reduce the number of inter-sessions, while a small τ increases their occurrences. Thus, a variable τ can be used to improve performance according to the access frequency: by increasing τ during access bursts, one increases the waiting time before the session ends, and

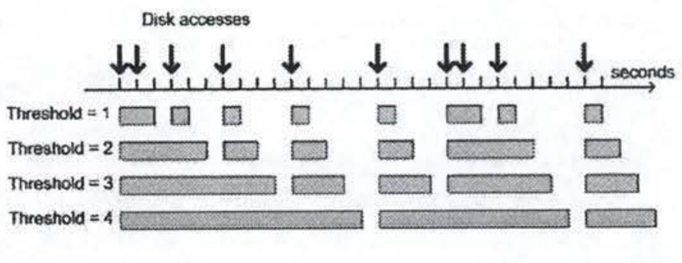

FIGURE 3.10: Example of sessions for different threshold.

thus reduces the risk to spin down the plates before the end of the burst. In the same way, τ can be decreased when the disk activity is characterized by long idleness periods separated by short bursts.

3.5.5 Software techniques for low-power TFT LCD displays

This section describes four software techniques to reduce the power consumption of the LCD display.

A LCD (Liquid Crystal Display) is a standard display device for portable systems, such as laptop computers and PDAs. Display power consumption is often the most significant contributor to the overall power budget for many portable devices, especially when multimedia applications are run.

Defining a power management strategy on a LCD display system is not a simple task, because the display does not experience explicit idle times. As a matter of fact, the display can be considered as idle when the user does not watching it, a situation that the system can not detect. For this reason, most policies concerning LCD displays are based on applications requirements (for example, a text editor requires lower resolution than a video player) or on user interventions (reducing the performance when no key is pressed for a long time).

Variable dot clock

The first technique, called ***Variable dot clock***, consists of decreasing the dot clock frequency, used for processor-display communication. This is a typical example of clock gating. Reducing the dot clock frequency slows down the pixels transmission between the ***frame buffer*** - the memory area where pixels are stored before being displayed - and the LCD screen, and consequently decreases the display refresh rate.

Thus, dot clock can be set to the lower possible frequency until flicker becomes excessive. This setting causes a reduction of power consumption for every component of the display system. The dot clock frequency can be configured according to the type of application: a simple text editor could use the lowest frequency, while programs where images change quickly should use higher frequencies.

Variable frame refresh

The second technique is called *Variable frame refresh* and consists of repeatedly turning off and on the LCD controller. Turning off the LCD controller closes the communication to the display, so that no image is transferred and the screen is not refreshed. Thus, power is saved because the whole display system is shut down. This method is viable because liquid crystals of the LCD display can maintain their orientation for a short time. During this time, the last displayed image before turning off the controller persists on the screen and its luminance decreases progressively. When the controller is turned on, pixels transmission starts again and the screen is refreshed. The delay between turning off and on the controller must be regulated in order to keep a suitable refresh rate. Power saving is obviously more important if the shut down period is long.

Liquid crystals orientation shift

The *Liquid crystals orientation shift* is based on the observation that user applications are often displayed in a window that occupies just a portion of the screen. In this case, the user does not need to see other portions with maximum clarity. It is possible to modify the characteristics of useless portions of the screen, in order to reduce power consumption. The color generated by a liquid crystal depends on the charge it receives. In order to reduce power consumption, it is possible to provide a null charge to every liquid crystal located into an useless part of the screen. This is simply performed by affecting a null value to the pixels corresponding to useless areas of the screen. As a consequence, each of the useless liquid crystals will adopt the default color of the screen (generally black or white).

Backlight shifting

The last proposed technique is based on the observation that the display has higher visibility in poorly illuminated environment because there is a high contrast between luminosity of the display and that of the environment. Decreasing light intensity of the light illuminating the screen (i.e. the backlight) allows to significatively reduce power consumption. It is possible to measure the luminosity of the environment with help of a light sensor, and automatically regulate the backlight intensity to keep a sufficient contrast with respect to the environment. This method gives very significant power saving, due to the large consumption of the backlight with respect to the whole display system.

3.6 Power management for wireless network cards

We have seen in the first chapter that IEEE 802.11 standard defines a power management mechanism. This power management requires the cooperation of the access point and the stations. We have also presented in details how it really works. Taking into consideration the present chapter, we realize that the power management defined in IEEE 802.11 corresponds to the definition of a new low-power policy at the component level, and this compared to devices that have not this low-power policy and that are powered all the time. Indeed, we have seen first that the IEEE 802.11 standardizes the wireless interface by describing a standardized physical and MAC layers of the network stack. However, we know that physical and MAC layers are in fact mainly implemented in the wireless card or, in what we have called in this chapter a component. Hence, the power management technique defined in IEEE 802.11 does not concern the operating system but only the component. Second, IEEE 802.11 defines two power-states: an ***run*** state and a ***sleep*** state. ***Run*** state is when power is applied to the radio. Transmit and receive operations can occur. ***Sleep*** state is a low power state where no transmission or reception can take place. IEEE 802.11 defines in fact two policies: the first policy corresponds to a mode where the card is constantly awake while the second policy corresponds to a mode where the card sleeps for a period - during this period, the card is in the ***sleep*** state - before wakening up to retrieve buffered frames from the access point - the card is in ***run*** state - and then returning back into the ***sleep*** state. This latter policy resembles one of the predictive approaches that we have seen previously in this chapter. We can say thus that the IEEE 802.11 power management mechanism consists of two component policies using a less-power state (***sleep*** state) and the normal power state (***run*** state).

We have seen in the second chapter an example of an IEEE 802.11 compliant wireless card: the Cisco Aironet serie 350 card. We have seen that this card defines three modes of operation: CAM, PSP and PSPCAM. From a power management view, these modes correspond in fact to policies, the Cisco card having always only two states (the ***sleep*** and ***run*** states)[3]. The specific additional Cisco policy is the PSPCAM mode that allows the card to be in CAM (Constantly Awake Mode) when traffic is heavy and to be in PSP (Power Save Polling) when the traffic is low. As seen in the second chapter, these latter policies correspond to these ones specified in the IEEE 802.11 standard.

If we have said that this card can be put easily by the user in PSP, CAM or PSPCAM mode at any time, this card offers also an interface that allows the user to shut down (***sleep*** state or even ***off*** state or to wake up the card (***run*** state) at any time and so, whatever the mode of the card. This card becomes thus a power manageable component (PMC) that can be switched off (***sleep*** state or ***off*** state) or switched on (***run*** state) by an ***external*** controller. As we have seen in this chapter, we can thus define policies at the ***system*** level ***independently*** of the policies implemented at the component level. The System-level requires a higher level of abstraction. The operating system seems thus suitable to implement the ***external*** controller and policies.

We can hope in reality to save more power if we implement policies at the ***operating sys-***

[3]In fact, we can even say that the Cisco card has three power states: ***sleep***, ***run*** and ***off*** states. In this last state, the card is no more powered. If the power savings can be great in this mode, we have to take into account that the time's wake-up is longer than in the ***sleep*** state. If the time's wake-up from the ***sleep*** state to ***run*** state takes about 300 ms, the time's wake-up from the ***off*** state to ***run*** state takes a few seconds. In this text, even if we say that the card is shutdown or off, we refer generally to the fact that the card is put into its ***sleep*** state and not into its ***off*** state (even if the ***sleep*** state could be most of the time replaced by the ***off*** state).

tem level. Indeed, the problem that we can notice when wireless cards are in the IEEE 802.11 low-power mode (or when the Cisco Aironet serie 350 card is in PSP mode) is that they wake up periodically to check wether outstanding packets for the client are buffered at the access point, independent of wether such packets are actually present, potentially leading to unnecessary and significant energy overheads. We can so hope reducing power consumption by having a greater observability of the workload.
For this reason, in the next chapter, we assemble levels of abstraction and we introduce the operating systems that are currently used for embedded systems and especially for the Compaq iPAQ PDA.

Chapter 4

Open source operating systems for embedded systems

4.1 Introduction

Since embedded systems appeared, operating systems did not stop multiplying. Embedded systems become indeed increasingly complex needing to implement an operating system in software. An operating system is thus the set of basic programs and utilities that make the embedded run.

iPAQ series 3600 is sold with the Pocket PC (formerly Windows CE) operating system. Pocket PC[1] is a proprietary operating system. The source code is not accessible to the public. There are in fact a lot of proprietary operating systems and many of them are used by vendors for their own products. The advantage is they are finely tuned for particular hardware and functionality, so they do not take much in the way of resources. The drawback is they usually are not flexible or powerful enough to handle complex applications like connecting networks and are often incompatible with open standards like Internet Protocol (IP). That is why other operating systems like Pocket PC, Linux[2] or Vxworks[3] are ported to handheld devices.

4.1.1 The Linux operating system

The Linux Operating System (Linux OS) is a free Unix clone with a very large support base. The complete kernel and practically all necessary applications to build a complete Unix system[4] are available for free and in source code under the GNU General Public License (GPL)[5]. There are currently a lot of Linux distributions specifically developed for embedded

[1]`http://www.microsoft.com/`

[2]`http://www.linux.org/`

[3]`http://www.windriver.com/`

[4]The kernel is the core of an operating system. An operating system contains indeed also additional tools like, in the Linux world, grep, more, less, make, gcc,... As the majority of these tools comes from the GNU project, we use also the name GNU/Linux.

[5]The GPL can be characterized by the following sentence: "This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free

systems and Linux could even become in the future the reference as operating system in the embedded world.

RTLinux

Technically, RTLinux (RealTime Linux) is not really a Linux distribution, but rather an extension to Linux that handles time-critical tasks. In RTLinux, a small hard-realtime kernel and standard Linux share one or more processors, so that the system can be used for applications like data acquisition, control, and robotics while still serving as a standard Linux workstation.

Familiar distribution

The Familiar distribution is hosted by `http://www.handhelds.org/` and can be find at `http://familiar.handhelds.org/`. The Familiar distribution puts towards producing a stable, and full featured Linux distribution for the Compaq iPAQ series of handheld computers, as well as applications to run on top of the distribution. Applications can be ported directly to the iPAQ without requiring major rewrites because Familiar distribution uses the mainstream Linux kernel. Except for storage and screen size issues, it is not a stripped-down variant and so, we can install Debian[6] packages on the iPAQ without having to recompile. Familiar includes indeed a new packaging system called *ipkg*, which is like RPM[7] or DEB packages[8] for desktop Linux. For all these reasons, we have opted for this Linux distribution and we have replaced the existing Pocket PC.
We present in next sections the main characteristics of the Linux operating system.

4.2 Process

A process is an instance of an executing program. Linux is a multiprocessing operating system. Processes are separate tasks each with their own rights and responsibilities. If one process crashes it will not cause another process in the system to crash. Each individual process runs in its own virtual address space and is not capable of interacting with another process except through secure, kernel managed mechanisms.

Processes communicate with each other and with the kernel to coordinate their activities. Linux supports a number of Inter-Process Communication (IPC) mechanisms (shared memory, signal, sockets[9],...).

Software Foundation; either version 2 of the License, or (at your option) any later version".

[6]`http://www.debian.org/`

[7]`http://rpm.redhat.com/`

[8]`http://www.debian.org/distrib/packages/`

[9]Sockets will be described later in further details (Chapter 5).

4.3 User space/kernel space

The role of the operating system, in practice, is to provide programs with a consistent view of the computerś hardware. In addition, the operating system must account for independent operation of programs and protection against unauthorized access to resources. This nontrivial task is only possible if the CPU enforces protection of system software from the applications.

Under Unix, the kernel executes in the highest level (also called supervisor mode), where everything is allowed, whereas applications execute in the lowest level (the so-called user mode), where the processor regulates direct access to hardware and unauthorized access to memory.

We usually refer to the execution modes as kernel space and user space. These terms encompass not only the different privilege levels inherent in the two modes, but also the fact that each mode has its own memory mapping - its own address space - as well.

Unix transfers execution from user space to kernel space whenever an application issues a *system call* or is suspended by a *hardware interrupt*. Kernel code executing a system call is working in the context of a process - it operates on behalf of the calling process and is able to access data in the processś address space. Code that handles interrupts, on the other hand, is asynchronous with respect to processes and is not related to any particular process.

4.4 Linux system calls

The functions that can call an application fall into two categories, based on how they are implemented.

- A library function is an ordinary function that resides in a library external to the program. Many of the library functions are in the standard C library. A call to a library function is just like any other function call. The arguments are placed in processor registers or onto the stack, and execution is transferred to the start of the functionś code, which typically resides in a loaded shared library.
- A system call is implemented in the Linux kernel. When a program makes a system call, the arguments are packaged up and handed to the kernel, which takes over execution of the program until the call completes. A system call is not an ordinary function call, and a special procedure is required to transfer control to the kernel. However, the GNU C library (the implementation of the standard C library provided with GNU/Linux systems) wraps Linux system calls with functions so that you can call them easily. Low-level I/O (Input/Output) functions such as *open()*, *read()* and *write()* are examples of system calls on Linux.

The set of Linux system calls forms the most basic interface between programs and the Linux kernel. Each call presents a basic operation or capability. Linux currently provides about 200 different system calls.

4.5 Modules

The modules allow the kernel to be light. In truth, modules are often ***device drivers***. They are out of the kernel and can be loaded in memory through the ***insmod*** command at any time without needing a kernel recompile. Since they do not form part of the kernel, they allow to decrease the size of the used kernel memory. They can be found under the ***/lib/modules/*** directory in the compiled kernel tree.

4.6 File systems

A virtual file system (VFS) is a logical management of data on a disk or a partition. Devices that can contain file systems are known as ***block devices***. The files in a file system are collections of data and are identified by an inode. Inode contains some information about the file (size, date,...) and a number that identifies the file.
Linux supports more than 15 file systems; ext, ext2, msdos, vfat, proc, smb, ncp, iso9660,...

4.7 Device drivers

Linux interacts with hardware devices via modularized software components called device drivers. A device driver hides the peculiarities of a hardware deviceś communication protocols from the operating system and allows the system to interact with the device through a standardized interface.

Under Linux, device drivers are part of the kernel and may be either linked statically into the kernel or loaded on demand as kernel modules. Device drivers run as part of the operating system and are not directly accessible to user processes. However, Linux provides a mechanism by which processes can communicate with a device driver - and through it with a hardware device - via file-like objects. These objects appear in the file system, and programs can open them, read from them, and write to them practically as if they were normal files. Using either Linuxś low-level I/O operations (system calls) or the standard C libraryś I/O operations, the programs can communicate with hardware devices through these file-like objects.

Device files are not ordinary files - they do not represent regions of data on a disk-based file system. Instead, data read from or written to a device file is communicated to the corresponding device driver, and from there to the underlying device. Device files come in three flavors:

- A character device represents a hardware device that reads or writes a serial stream of data bytes. Serial and parallel ports, tape drives, terminal devices, and sound cards are examples of character devices.

- A block device represents a hardware device that reads or writes data in fixed-size blocks. Unlike a character device, a block device provides random access to data stored on the device. A disk drive is an example of a block device.
- A network device (i.e., also called the Network Interface Card (NIC)) is similar to block device but there is a big difference between them: block drivers operate only in response to requests from the kernel, whereas network drivers receive packets asynchronously from the outside. In other words, a block driver is asked to send buffer toward the kernel while the network device asks to push incoming packets toward the kernel.

The ***ioctl() system call*** is an all-purpose interface for controlling hardware devices. The first argument to ***ioctl*** is a file descriptor, which should be opened to the device that you want to control. The second argument is a request code that indicates the operation that you want to perform. Various request codes are available for different devices. Depending on the request code, there may be additional arguments supplying data to ***ioctl***.

4.8 Interrupts

Devices have to deal with the external world, which often includes things such as spinning disks, moving tape, wires to distant places, and so on. Much has to be done in a time frame that is different, and slower, than that of the processor. Since it is almost always undesirable to have the processor wait on external events, there must be a way for a device to let the processor know when something has happened.

An interrupt is simply a signal that the hardware can send when it wants the processorś attention. For the most part, a driver need only register a handler for its deviceś interrupts, and handle them properly when they arrive.

4.9 Networking

Linux supports several types of networks: TCP/IP, AX25, IPX, APPLETALK,.... The most famous network stack is certainly TCP/IP. Linux is nearly born at the same time as the Internet and it has supported quickly the TCP/IP protocol suite. Thanks to its success which made it deploy throughout the world, TCP/IP is a powerful network. We will limit in this thesis to the TCP/IP stack because of the success of TCP/IP and also because the wireless IEEE 802.11 can be very easily interfaced with TCP/IP.
The current implementation of TCP/IP and related protocols for Linux is called NET-4. NET-4 means the fourth implementation of TCP/IP for Linux. NET-4 supports nearly all the features we would expect from a TCP/IP implementation. The implementation of TCP/IP in Linux is recognized as a reference. That is one of the reasons why Linux is often proposed for embedded systems for which networking becomes more and more crucial.

4.10 Conclusion

If the iPAQ is sold with the Pocket PC operating system, we can easily replace it by another embedded operating systems. Indeed, the Linux operating system offers the advantage to be open source. The TCP/IP networking is moreover well implemented under Linux. Linux is also currently intensively adapting for embedded systems by several projects like the Familiar project which is developing a Linux distribution targeted specifically the iPAQ. Efforts have finally been made to facilitate the support of wireless interface in this distribution. That is why we chose this operating system and especially this Familiar distribution.
In the operating system, TCP/IP stack is presented logically in the form of a candidate to integrate some power management techniques relating to the network interfaces. That is why we overview this stack in the next chapter.

Chapter 5

TCP/IP under Linux: NET-4

The main goal of this chapter is to identify the key points of the TCP/IP stack under Linux. It is important to understand how TCP/IP is implemented if we want to build some power management technique for wireless interface. How the interprocess communication[1] (IPC) between distant applications is working will help us to know how to implement an efficient power management technique.

We overview first how TCP/IP stack works. The goal is not to describe deeply all the rules but to give a general overview of the protocol stack and remember some of its important mechanisms. Then we look at the implementation of the memory buffers and we see how data is sent through the network stack. Finally, we conclude this chapter by reviewing the power management opportunities.

5.1 Overview of TCP/IP

5.1.1 Introduction

We saw in the first chapter the Local Area Network (LAN) and Wireless Local Area Network (WLAN) and the differences that may exist between a wireless and a wired interface. IEEE 802.3 and IEEE 802.11 define standards that allow different cards operating each other whatever the brand of cards. It relates to especially the hardware component. To transport application data and offer services like reliable connection between applications located on distant hosts, we need rules to ensure compatibility whatever the operating system running on hosts. These rules are thus mainly implemented in a bundle of standardized protocols (in the operating system) to be as transparent as possible with the applications of the user space.

The name of TCP/IP refers to an entire suite of data communication protocols. It defines how machines should communicate with each other via a network, as well as internally to other layers of the protocol suite. The suite gets its name from two of the protocols that belong to it: the Transmission Control Protocol (TCP) and the Internet Protocol (IP). Although there are

[1]Interprocess communication is the transfer of data among processes

many other protocols in the suite, TCP and IP are certainly two of the most important. One of the great benefits of TCP/IP is that it provides interoperable communications between all types of hardware and all kinds of operating systems. TCP/IP is freely available[2]. It is also independent from specific physical network hardware. This allows TCP/IP to integrate many different kinds of networks. TCP/IP can be run over an Ethernet, a token ring and virtually any other kind of physical transmission medium. We will use it on top of a wireless interface. The worldwide Internet is even built on the TCP/IP protocol suite. These are some reasons why TCP/IP is one of the most used protocol suites in computer communication systems. We can thus easily understand why it would be great to develop some power management technique with this protocol suite.

5.1.2 The OSI reference model

We take again OSI model for understanding data communications terminology. We talked about the data link layer and the physical layer in the first chapter. The other OSI layers are:

- Network Layer: The Network Layer manages connections across the network and isolates the upper layer protocols from the details of the underlying network. In TCP/IP, it is the Internet Protocol (IP) that isolates the upper layers from the underlying network.
- Transport Layer: this layer in the OSI reference model guarantees that the receiver gets the data exactly as it was sent. In TCP/IP, this function is performed by the Transmission Control Protocol (TCP) or by the User Datagram Protocol (UDP), that does not perform the end-to-end reliability check.
- Session: this layer manages the sessions (connections) between cooperating applications. In TCP/IP, we can say that this layer is performed by the socket abstraction represented here by the Socket Layer.
- Presentation Layer: for cooperating applications to exchange data, they must agree about how data is represented. This layer does not really have its equivalent in TCP model.
- Application Layer: this is the level of the protocol hierarchy where user-accessed network processes reside. That relates to any network process that occurs above the Transport Layer.

[2]http://www.rfc-editor.org/ to get them

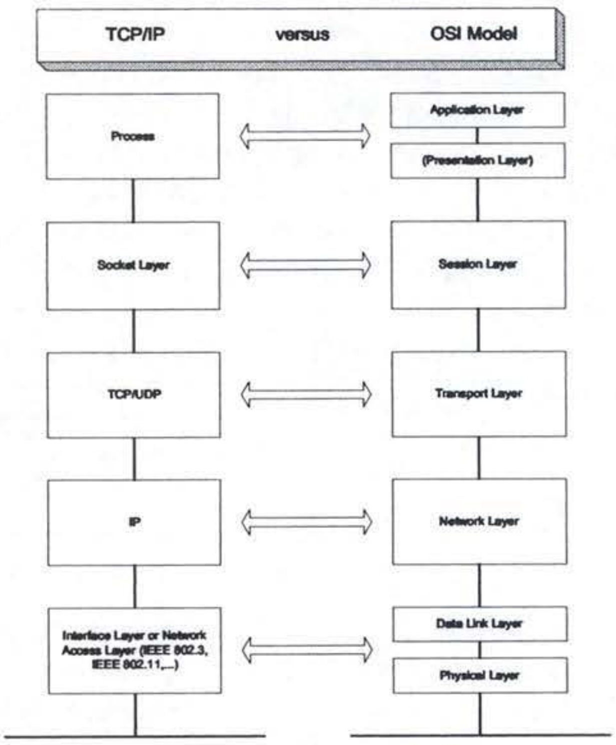

FIGURE 5.1: TCP Model versus OSI Model

The figure 5.1 depicts the two models. Keeping the OSI model like reference, we can see that the TCP/IP model can easily interface with the underlying layers like those defined in IEEE 802.3 or even in IEEE 802.11. The Interface Layer prepares and formats the data that will be sent to the hardware buffer of the network device (Ethernet card, wireless card,...). As we have seen, that does not mean that we cannot see any differences between an IEEE 802.3 network using the TCP/IP protocols with an IEEE 802.11 network using the TCP/IP protocols. The performance can be very different from one network to another and it is not only due to physical differences: for example, TCP may in reality react differently according to whether it is an IEEE 802.3 or an IEEE 802.11.

The structure of TCP/IP is seen in the way data is handled as it passes down the protocol stack from the Application Layer to the underlying physical network. Each Layer in the stack adds control information to ensure proper delivery. This control information is called a header because it is placed in front of the data to be transmitted. Each layer treats all of the information it receives from the layer above as data and places its own header in front of that information. When data is received, the opposite happens. Each layer strips off its header before passing the data on to the layer above. It is worth noting that each layer has its own independent data structures and its own terminology. For example, applications using TCP refer to data as a stream or a segment[3], while applications using UDP calls its data

[3]Some applications are called *streaming application* because they delivery "just in time"data, especially video or audio data. The term *streaming* refers more here to the type of traffic generated rather than the

structure a message or a datagram[4]. In reality, the data structures of a layer are designed to be compatible with the structures used by the surrounding layers for the sake of more efficient data transmission.

5.1.3 The Interface Layer

Unlike higher-level protocols, the Interface Layer must know the details of the underlying network. It must indeed format the data - IEEE 802.3 frame format or 802.11 frame format for example - to build frame correctly so that the network card can send it properly. The Interface Layer interfaces the higher-level protocols of the TCP/IP model with the hardware device. In other words, it interfaces the Linux operating system[5] with the hardware component by offering a generic interface. A new Interface Layer must be developed when new hardware technologies appear, like for the wireless card seen in the first chapter where a new Data Link Layer and a new Physical Layer specific for wireless communications were created. In the case of LAN (and also WLAN), this layer must still also find the MAC address corresponding to the IP address of the destination. The Address Resolution Protocol (ARP) is an example of protocol that maps IP addresses to MAC addresses. Protocols in this layer appear like a combination of a device driver (the driver of the wired or wireless card) and related programs that perform related functions such address mapping (ARP).

5.1.4 The Internet Layer

The Internet Protocol (IP) [RFC 791] is a connectionless protocol. This means that IP does not exchange control information (called a "handshake") to establish an end-to-end connection before transmitting data[6]. It is also an unreliable protocol because it contains no errors detection and recovery code. The only thing of which we are assured with IP is that when a packet is delivered to a host, the IP address of this host corresponds to the destination address registered in the IP packet[7]. A Cyclic Redundancy Code (CRC) is indeed used to preserve from corruptions the IP header. If a router notes that the IP header is corrupted, the packet is dropped[8]. We must remember that the 32-bit IP[9] identifies the destination network and

protocol used. This traffic corresponds indeed to a continuous flow of packets in opposition to a bursty traffic.

[4]For IP, the data structure is called a packet, while for Interface Layer it is called a frame.

[5]TCP/IP protocol suite is indeed implemented in the operating system, while the Interface Layer is implemented as well in hardware as in software.

[6]In contrast, a connection-oriented protocol, like TCP, exchanges control information with the remote system to verify that it is ready to receive data before any data is sent. When the handshaking is successful, the systems are said to have established a connection.

[7]It is worth noting that an IP address is not associated to a host but to a network interface. A host may thus have several IP addresses, each one of them being associated to a particular network interface.

[8]A CRC is a number derived from, and stored or transmitted with, a block of data in order to detect corruption. By recalculating the CRC and comparing it to the value originally transmitted, the receiver can detect some types of transmission errors.

[9]We describe here the fourth version of IP. Even if this version is still largely deployed, the IPv6 (Internet Protocol version 6) takes importance more and more. IPv6 brings additional features like a newly 128-bit address scheme. It should be solved the lack of IPv4 addresses because the demand for IP addresses exceed the available supply.

the specific host on that network. If the destination address is not on the local network, the packet must be passed to a router for delivery. The router looks at the destination address in the IP header to determine which network the packet must be sent to. IP makes the routing decision for each individual packet. A host is by definition a device that does not perform forwarding.

One other function provided by IP is the fragmentation. It may be necessary for the IP module in a gateway to divide the packet into smaller pieces. A packet received from one network may be too large to be transmitted on a different network. This happens when a gateway interconnects dissimilar physical networks.

If IP receives a packet that is addressed to the local host, it must pass the data portion of the packet to UDP or TCP according to the value written in the ***protocol number*** field in the IP header. Otherwise, if IP receives a packet that is not addressed to the local host and if it is not a router, the packet must be dropped.

It is worth noting that there is another protocol in the Internet Layer: the Internet Control Message Protocol (ICMP). ICMP is a protocol that uses IP to send its messages. ICMP is used to follow control, to report errors and informational functions for IP and even also for UDP and TCP.

5.1.5 The Transport Layer

This layer allows applications to choose the User Datagram Protocol (UDP) [RFC 768] or the Transport Control Protocol (TCP) [RFC 793]. UDP is similar to IP in the sense that it does not provide more services[10]. UDP adds to the data portion an UDP header of only 64-bits. This specifies among other things the destination and source port numbers allowing process to send data to another process located on a different machine. The source and destination port numbers are the characteristics of the Transport Layer.

On the opposite, TCP is a connection-oriented, byte-stream, reliable protocol.

First TCP is connection-oriented. It uses the three-way handshake mechanism. The goal of the handshake is to establish a connection before data is transmitted. The "three-way"is because of 3 steps. First, a host A begins the connection by sending to a host B a segment with the synchronize sequence numbers (SYN) bit set. This segment tells host B that A wishes to set up a connection, and it tells B what sequence number host A will use as a starting number for its segments. Sequence numbers are used to keep data in the proper order. TCP is reliable and keeps the data in the right order. Then, host B responds to A with a segment that has the acknowledgement (ACK) and SYN bit set. The B's segment acknowledges the receipt of the A's segment and informs A which sequence number host B will start with. Finally, host A sends a segment that acknowledges receipt of B's segment, and transfers the first actual data. After this exchange, A knows that the remote TCP is alive and ready to receive data. Data can be transferred. When both hosts have concluded the data transfers, they will exchange a

[10]In addition to port numbers, the only difference is that CRC in UDP covers all the datagram, while the IP CRC covers only the IP header. Nevertheless, the UDP CRC is not mandatory, so...

three-way handshake with segment containing the "No more data from sender"bit (called the FIN bit) to close the connection. It is important to note that the three-way handshake is not 100% reliable and that some packets can be lost.

Second, TCP is a byte-stream protocol because TCP views the data it sends as a continuous stream of bytes, not as independent packets. The sequence number in the TCP header relies indeed on a byte-numbering system and not on a segment-numbering system. The application data is broken into what TCP considers the best sized chunks to send. This totally different from UDP, where each write by the application generates a UDP datagram of that size.

Third, TCP is reliable because it implements the positive ACK mechanism. We saw this mechanism in the first chapter. IEEE 802.11 networks use in reality also this mechanism to avoid packet losses. In TCP, the receiver puts in an acknowledgment number field (in the TCP header) the sequence number of the next byte it expects to receive. The acknowledgment number is a positive acknowledgment of all bytes up to that number. This TCP ACK is in reality used for two functions: first for positive acknowledgment and second for flow control. For positive acknowledgment, because the receiver sends always a segment in response to a received segment. Indeed, the ACK number inside the receiver's segment is equal to the sequence number of the next expected segment. If this number is equal to a packet already sent, that means that receiver has not received it and it will be retransmitted by the sender after the expiration of a timer[11]. Second, for flow control, because it is said that if the sender does not receive any acknowledgment for a number of segments (that correspond to the size of its window[12]), the sender must stop and waits for the acknowledgements. In fact, the receiver of a TCP connection can inform the sender that it can continue sending segments as long as the total number of bytes that it sends is smaller than the window of bytes that the receiver can accept. The receiver controls the flow of bytes from the sender by changing the size of the window. A zero window tells the sender to cease transmission. This mechanism prevents a fast host from taking all the buffers on a slowest host. The receiver must wait until a larger window could be advertised to the sender in order to avoid the window silly syndrome: indeed, if the receiver informs the source immediately after its window becomes non-zero, the receiver will advertise small windows and the sender can transmit small amounts of data instead of waiting for additional data, to send a larger segment. This situation is often to avoid because of non-efficient TCP performance.

We tackled in chapter 1 the problem of TCP in wireless communication. We have said that packet losses cause a drop in the TCP throughput. It is indeed caused by the assumption of the congestion avoidance mechanism implemented in TCP[13]. This assumption is that packet

[11]We recall that we need a timer at the sender side to retransmit the lost segment because the receiver does not inform the sender of the lost packet. It is actually a *positive* acknowledgement mechanism.

[12]We can consider a window like a finite amount of buffer space.

[13]This mechanism is a way to deal with lost packets. TCP adapts its throughput according to lost packets. If lost packets appear, TCP decreases its throughput while it is increasing slowly its throughput when it does

lost caused by damage is very rare. It says that it is caused rather by the congestion. If it is true with a IEEE 802.3, it is not true with IEEE 802.11 network where much collisions can appear. The result of this is a drop in the TCP throughput. In addition to that, we can notice another problem that has the same effect: because TCP uses the ACK mechanism, the total transmission time of a TCP segment (including the acknowledgements) takes more time because IEEE 802.11 MAC layer implements already its proper ACK mechanism. As a matter of fact, when a host sends a data frame, the receiver will send an immediate 802.11 ACK and will also eventually send a TCP ACK. Finally, the sender must acknowledged the TCP ACK. These many exchanges of ACK cause also a drop in the TCP throughput. Many works are currently carried out today to improve the TCP performance above wireless networks[14].

It is worth noting that broadcasting and multicasting are not applicable to TCP. Only UDP can support them.

To summarize, TCP packetizes the application data into segments, sets a timeout any time it sends data, acknowledges data received by the other end, reorders out-of-order data, discards duplicate data, provides end-to-end flow control, and calculates and verifies a mandatory end-to-end checksum (CRC).
On the other hand, UDP is a simple protocol. Like other protocols in the stack, UDP "inter-acts"also with other protocols or mechanisms. For example, when a UDP packet must be sent to an unknown host on a IEEE 802.3 network, we know that the packet must be sent first in the ARP buffer in order to map the IP address with the MAC one. Because UDP does not keep copy of its packets (it is not a reliable protocol), it can lose some packets if there is a burst because ARP does keep only one packet in its buffer.

5.1.6 The Socket Layer

The socket is an abstraction which makes it possible applications to communicate with the network stack. More precisely, a socket is a bidirectional communication device (but it is completely soft, it is not a physical device) that can be used to communicate with another process on the same machine or with a process running on other machines. In other words, the socket layer is a generic interface between the protocol family (here TCP/IP but there are other protocol families) and the user space of the kernel. Indeed, the application must be able to send and receive TCP/IP messages by opening a socket and reading and writing data to and from the socket. It is possible because sockets are file descriptors[15]. This Socket Layer provides thus services like other layers because it simplifies program development: the programmer needs only worry about manipulating the socket and can rely on the operating system to actually transport messages across the network correctly.

not face with packet losses.

[14]`http://www.drizzle.com/~aboba/IEEE/`

[15]A file descriptor is an integer value that refers to a particular instance of an open file in a single process. It can be open for reading, for writing or for both reading and writing. It does not have to refer to an open file. It can represent a connection to a hardware device or in this case, an open socket.

A socket has three parameters: communication style, namespace and protocol.
The communication style specifies how the socket treats transmitted data, how the data are handled and how they are addressed from the sender to the receiver. There are two different communication styles:

- Connection style: this style guarantees delivery of all packets in the order they were sent. If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender. The address of the sender and the receiver are fixed at the beginning of the communication when the connection is established. In the TCP/IP protocol stack, this style corresponds to TCP.
- Datagram style: this style does not guarantee delivery or arrival order. Packet may be lost or reordered in transit due to network errors or other conditions. Each packet must be labeled with its destination and is not guaranteed to be delivered. This is a "best effort"service: packets may disappear or arrive in a different order than shipping. This style of communication relates to UDP, in the TCP/IP family protocol.

The socket namespace specifies how socket addresses are written. A socket address identifies one end of a socket connection. In the local namespace (i.e., when sockets are used for communications between processes on the same machine), socket addresses are ordinary filenames. However, when sockets are used by processes not located on the same machine, socket addresses are network addresses. The address scheme depends on the protocol family. If the protocol family is TCP/IP, a socket address is composed of the Internet address (IP address) of a host attached to a network and a port number. The port number distinguishes among multiple sockets on the same host.

Finally, a protocol specifies how data is transmitted. We can choose UDP or TCP protocol in TCP/IP according to wether we want respectively a connectionless or a connection-oriented protocol.

In Linux, sockets are flexible because applications from the user space can directly use special system calls to manipulate sockets. These system calls are known as SIOC. SIOC stands for ***socket ioctl***[16]. We can thus create (***socket()***), destroy a socket (***closes()***) but also label a server socket with an address (***bind()***), configure a socket to accept conditions (***listen()***), accept a connection and create a new socket for the connection (***accept()***). For instance, if a programmer wants to create a UDP socket, he calls: ***socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)*** that creates a connectionless (SOCK_DGRAM) UDP (IPPROTO_UDP) socket of the TCP/IP (AF_INET) protocol family. Each socket contains two buffers: a receive socket buffer and a send socket buffer. We will see later for what they are used.

The TCP/IP network interface in Linux is in fact based on the BSD[17] socket interface, which

[16]See chapter 4 for more detail about that.

[17]BSD stands for Berkeley Software Distribution. In the 1980's the Computer Science Research Group (CSRG) at the University of California Berkeley developed an advanced version of the Unix system than

is standard on most Unix OSs. In fact, Linux operating system supports more or less 30 protocol families[18]. We can thus divide this Socket Layer in two layers: the BSD Layer and the INET Layer (a socket layer for TCP/IP). BSD Layer has a structure containing the protocol families that have been initialized during boot[19] and for each of them, it has the create function (in TCP/IP, the *inet_ create()* that allows creating a socket of this family when a user from the user space invokes socket() call. The INET socket is defined by the structure *sock* in the */net/sock.h* file. It can be a UDP or a TCP socket depending on the value of the argument specified by the programmer in the *socket()* call.

5.1.7 The application Layer

The Application Layer is at the top of TCP/IP protocol. This layer include all processes that use TCP or UDP protocols to deliver data.

As we have just seen above, applications can choose the family protocol and between two communication styles and this, through socket commands. If the lower TCP/IP protocols must handle with packets, datagrams and segments, they do not examine or modify the data passed by applications. This allows a great flexibility in TCP/IP protocols. However, it is the role of applications to format and parsing messages at the receiver side in order to recover original data. Sender and receiver must thus agree on how the data will be formatted. Applications must also implement their own send and receive buffers and sometimes implement in addition complex buffer management technique (for example, with UDP, applications can implement buffer to reorder packets). These buffers are called "user buffers"or "application buffers"to distinguish them from other buffers implemented at other layers (in the kernel space or in hardware).

Different classes of applications generate different traffic patterns. A relevant class of applications includes streaming applications. Examples thereof are conversational multimedia applications (voice or video over IP), interactive multimedia applications (distributed simulations, network games,...) and non-interactive multimedia applications (distance learning, audio/video broadcasts, news on demand,...). These applications delivery packets and hope that packets are received "just in time"at the other end of the connection. They support more or less some packet losses but it depends on the specific application. Neither UDP nor TCP are really appropriate for this kind of applications. Some use UDP, other use TCP. TCP is reliable but it may incur additional latency because of the retransmissions eventually scheduled by the congestion control mechanism, while UDP is multicast capable but not reliable. The traffic generated by these applications is a continuous flow of data, with often a short/synchronous inter-gap space between packets.

included TCP/IP networking. It is the foundation for TCP/IP implementations. Today several groups have taken the code released from Berkeley, improved upon it, and added to it substantially.

[18]The */include/linux/socket.h* file indexes all protocol families supported.

[19]The function *void__init proto_ init(void)* in */net/socket.c* is responsible to initialize all the protocol families.

Other applications are not streaming: these are request-response applications, batch applications and interactive reliable applications. The traffic generated by these applications is bursty with long inter-gap space between packets.

The default behavior of a socket call is to block until the requested action is completed. For example, the *recvfrom()* function, which allows applications to retrieve data from the socket, does not return until at least a datagram from the server is received. Of course, a process with a blocked function is suspended. A problem can appear at the client if the server should send one packet and that this one is lost. Indeed, as the server uses protocol UDP, it does not worry to know if the customer received the packet or not and so, the customer can indefinitely be blocked. That is why it is possible to change the behavior of the socket so that calls are nonblocking. For such a socket, if a requested operation can be completed immediately, the call's return value indicates success; otherwise it indicates failure. In either case the call does not block indefinitely.

5.2 Implementation

In Linux, the implementation of the operating system is mainly in C language. That is why we sometimes refer in this section to some C pointers or C structures.

5.2.1 Memory buffers

Memory buffers are very important and the performance of the networking protocols is directly related to the memory management scheme used within the kernel. The main goal of memory buffers is to hold the user data, that travels from the process to the network interface, and vice-versa. We can consequently easily give an account of its role to us. Memory buffers contain also a variety of other miscellaneous data: source and destination addresses, socket options, and so on. These structures, which are also called ***sk_ buff***[20], can be created by the Socket Layer or by the Interface Layer according to whether the application sends or receives a packet. Indeed, each packet that arrives at the interface is encapsulated in a ***sk_ buff***. In this case, it is the driver that creates ***sk_ buffs***. On the contrary, data coming from the socket calls are also copied into ***sk_ buffs***, but in this last case they are created by the Socket Layer. Several functions are defined for each layer so that protocols in TCP/IP can handle the ***sk_ buff*** structures (allocate or free a ***sk_ buff*** structure, add data to the structure,...).
There is a one-to-one relationship between packets and ***sk_ buffs*** (i.e., one packet, one buffer).

The figure 5.2 depicts the ***sk_ buff*** structure. Here are the $skb \rightarrow next$, $skb \rightarrow prev$, $skb \rightarrow dev$ C pointers and other C pointers used to address the data in the packet.

- next: this points to the next buffer (next *sock_ buff*)
- prev: this points to the previous buffer

[20]sk_ buff stands for socket buffer. This structure is defined inside the */include/linux/skbuff.h* file.

- dev: this points to the device a packet arrived on/is leaving by.
- head: points to the beginning of the allocated space.
- data: is the beginning of the valid octets (and is usually slightly greater than head).
- tail: this points to the end of the valid octets.
- end: this points to the maximum address tail can reach.

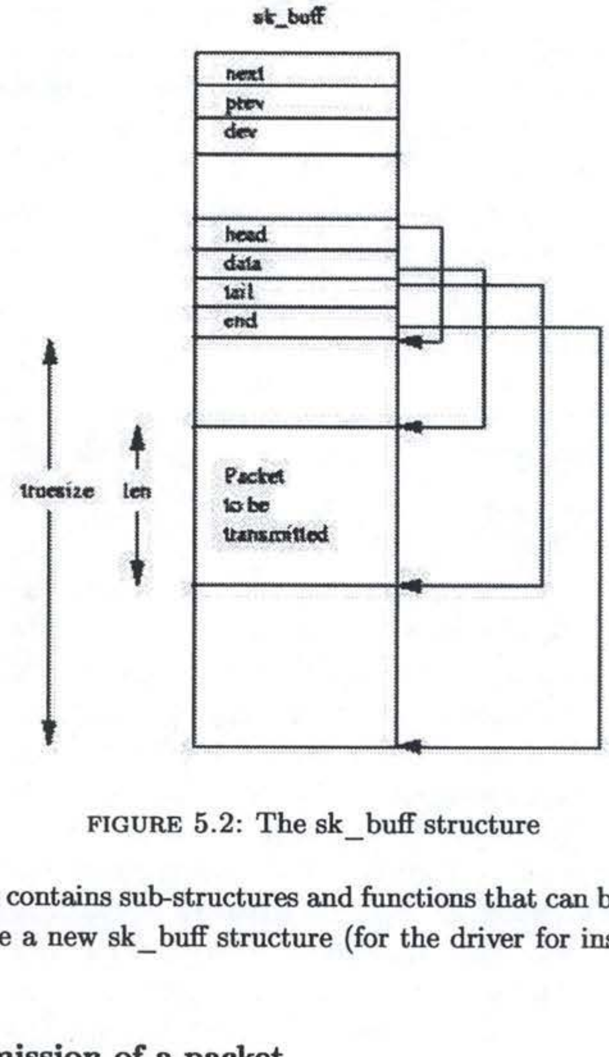

FIGURE 5.2: The sk_buff structure

The sk_buff structure contains sub-structures and functions that can be used by each protocol to append data, create a new sk_buff structure (for the driver for instance,...), fill up with options,...

5.2.2 The transmission of a packet

At the sender:

1. When a process in the user space wants to send a packet, it calls the *write()* function on a socket (i.e., that can be also sendto() or send() functions)[21].

[21]Indeed, the *write* function can be called because a socket is represented by a file descriptor. Thus, any technique to write to a file descriptor can be used to write to a socket. The *send()* function is specific to the socket file descriptor and provides just an alternative to *write* function with a few additional choices. The *recvfrom()* and *sendto()* functions are equivalent respectively to *recv()* and *send()* functions.

2. The data is copied by the Socket Layer from the process space (i.e., the user space) into **a sk_buff structure**.

3. The data (contained in a *sk_buff* structure) goes through TCP/IP and the packets are put into the **queuing discipline**, which is a queue attached to the network card. This queue is thus composed of a chain of*sk_buffs*.

4. Finally, the *hard_start_transmit()*transmission procedure of the driver is called. This method puts the data on an ongoing queue, so the **hardware buffer**. In our case, the driver Cisco Aironet method *airo_start_xmit()* is called. The socket buffer passed to *airo_start_xmit()* contains the physical packet as it should appear on the media, complete with the transmission-level. In fact, in our case, our Cisco driver (/drivers/net/wireless/airo.c) modifies the packet because the packet build by the kernel is an IEEE 802.3 packet. The Cisco card is indeed seen by the kernel like an IEEE 802.3 card[22]. The Cisco driver encapsulates the 802.3 packet into the 802.11 packet, by adding the proper header and tail. As the driver is part of the operating system, we can say that the operating system supports 802.11, while the kernel not.

At the receiver:

1. The packet is received by the card. A receive interrupt is generated and the interrupt handler of the driver copies data from **hardware buffer** into fresh *sk_buff*. This *sk_buff* structure is queued on **backlog**. The **backlog** queue is similar to the **queuing discipline** except that it is destined to receive packet while **queuing discipline** is destined to send packets. The handler interrupt schedules a softirq[23].

2. When the interrupt handler returns, the softirq (previously scheduled) triggers and executes the TCP/IP stack, then puts the data into the **receive socket buffer** and awakes the sleeping blocked process over this queue (e.g., with a blocking *recv()* on the socket, the process is put into sleep state if there are not any data in this receive socket buffer.).

3. Finally, the process copies the data from the kernel space into the **reception buffer**. This last buffer must be in fact implemented by the application itself. It is specified in the parameters of the *recv()* function. Indeed, applications must always define as parameter of *recv()* or *send()* functions the amount of data to be received or read.

5.2.3 The TCP buffering

We must make a distinction between UDP and TCP at Socket Layer. In fact, with UDP, after a call to *send()*, the Socket Layer copies the data in a *sk_buff* structure and these go

[22]We can now more easily understand here the allusion "Wireless Ethernet"which one sometimes makes with network 802.11! (See the introduction of chapter 1).

[23]Softirq are typically used by interrupt handlers. Indeed, the interruptions stop the task in progress and while they are treated by their handler no other task can be performed. However, part of work should not be necessarily carried out immediately and it is thus desirable that interrupt handlers can defer certain tasks to later so that they do not block the system a too long time.

immediately through the protocol stack until to be stored in the buffer attached to the network card (i.e., the queuing discipline buffer). That is why it is said that UDP has not socket send buffers. Indeed, because UDP packets (UDP *sk_buffs*) are sent immediately, there is not any queue of *sk_buffs* at socket layer.
On the contrary, with TCP, we can think of the sequence of all bytes (in one direction) on a TCP connection up to a particular instant in time as being divided into three socket queues (chains of *sk_buff* structures):

- Send queue: bytes buffered in the sockets layer at the sender that have not yet been successfully transmitted to the receiving host.
- Receive queue: bytes buffered in the sockets layer at the receiver waiting to be delivered to the receiving program, that is, waiting to be returned via a *recv()* call.
- Delivered queue: bytes already returned to the receiving program via *recv()*.

A call to *send()* appends bytes to send queue. TCP is responsible for moving bytes from the send queue to the receive queue. It is important to realize that this transfer cannot be controlled or directly observed by the user program and that it occurs in chunks whose sizes are more or less independent of the size of the application buffers passed in *send()* calls. Bytes are moved from receive queue to deliver queue as a result of *recv()* calls by the receiving program. The size of the transferred chunks depends on the amount of data in receive queue and the size of the buffer given to *recv()*.

5.3 Conclusion

We have identified two main parts of the iPAQ that can be exploited for energy efficient system management. First, at MAC layer implemented on the wireless card. IEEE 802.11 standard states that wireless cards must support at least two modes: a "normal"mode and a low-power mode. As for the Cisco card, it supports three modes: a "normal"mode (CAM), a low-power mode (PSP) and an intermediate mode (PSPCAM). However, the effectiveness of these techniques is limited by the poor observability of this layer. A second opportunity for power management resides in the TCP/IP stack. We have just revisited in this chapter the TCP/IP stack in order to introduce a power mechanism that could exploit the information based on the application needs to shut down the card when an idle period is identified and awake the card when sending or receiving packets is required.

It is now important to take note of a few considerations. We can easily understand that it is not obvious to integrate a power mechanism in the TCP/IP stack and this, for several reasons.

First, the wireless card is a special device compared to other devices. It is a special device in the sense that its utilization cannot be predicted as well as that of the other devices. The network traffic depends indeed on other hosts of the network. It is thus more difficult to identify the idle periods during which the card can be shutdown without incurring significant

latencies and power overheads. Moreover, the compatibility between network devices is not always acquired[24]. Finally, wireless cards are still different from other devices for the overheads (time) associated to their wake-up. In fact, these overheads are even higher than in computation sub-systems, since the wireless client may need to re-associate with the network for continued service. For example, the wake-up time of a Cisco Aironet wireless LAN card is almost twice the wake-up time for an Intel StrongARM SA-1110 processor.
Second, TCP/IP protocol stack is not simple. In spite of the reference to the layered OSI model, many mechanisms interact with each other: we have seen for example that UDP can badly interact with ARP by loosing some packets or that TCP had not its better performances above a wireless card. The network can be influenced by humans: if an application sends a buffer very large with UDP, it can cause the IP fragmentation, which is often to avoid. Programmer can also use blocking sockets or non-blocking sockets. The network administrator can as well modify some parameters of the wireless card or also some TCP or UDP parameters (like the size of the receive or send buffers). Finally, the traffic generated by applications can be very different - streaming applications versus non-streaming applications (i.e., also sometimes called interactive applications) - and it is difficult to imagine that integrate only one power mechanism in TCP/IP stack can be very efficient in all situations. Indeed, many of the current power management techniques proposed today in the literature targeted streaming applications - or rather a specific traffic profile. Sometimes, some papers study the energy consumption made by specific streaming applications, like Microsoft media, Real or Quicktime[25]. Many papers also target the other specific layers by developing techniques at physical or MAC layers.

We present in the next section a power management technique taken from the literature[26] that shows an example of how power management can be implemented by exploiting information available at the different layers of the TCP/IP stack.

[24]We experienced the case with wireless cards in Bologna. We used a Cisco Aironet card with a Lucent Access point. These components are both IEEE 802.11 compliant. However, because PSPCAM mode of the Cisco card is a specific Cisco power management policy, the card does not work properly in this mode with the Lucent Access point. Streaming applications crashed after some time they have been executed. This simple example shows that the compatibility is very important in networks and that power management techniques can be very efficient only if we take into account not only the local host but also all the network. At higher level, different TCP/IP implementations in a network can have also some impacts

[25]SURENDAR,C., Wireless Network Interface Energy Consumption Implications of Popular Streaming Formats. Department of Computer Science, University of Georgia, Athens, October 2001.

[26]BERTOZZI, D., RAGHUNATHAN, A., BENINI, L., RAVI, S., Transport Protocol Optimization For Energy Efficient Wireless Embedded Systems. *In EEEE Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2003.

Chapter 6

Power Management for wireless network cards

6.1 Introduction

It is time now how we can implement a power management technique inside the TCP/IP protocol stack. We saw in chapter 5 the main mechanisms of TCP/IP and we also saw some reasons why power management technique is not easy to implement in the TCP/IP protocol stack.

The power management technique presented in this chapter is taken from the literature and is an example of how power management can be implemented by exploiting information available at the different levels of the TCP/IP.

6.2 Transport protocol optimization for energy efficient wireless embedded systems

6.2.1 Low-Power TCP buffering

The proposed technique relates to the Transport Layer, and especially to the TCP protocol. In truth, power saving mechanisms at the Transport Layer have the advantage of being application-independent, and as most network applications run on top of TCP, the optimization of its energy efficiency allows a large range of applications to make an energy-aware use of the network interface. The proposed technique is thus specific to TCP, relying on the fact there are a lot of applications using TCP.

The monitored objects are the receive and send TCP buffers. The technique exploits the in-built signalling mechanisms of TCP, such as window advertisement and silly window syndrome (SWS) avoidance, to identify scenarios where low power modes can be triggered with minimal overheads. Minimal modifications of the original TCP are thus guaranteed. It is an advantage because it remains compatible with potential other non-modified host TCP/IP implementations. We show that the TCP buffering mechanisms can be exploited to significantly increase energy efficiency of the Transport Layer with minimum performance overheads. The goal is to identify periods of inactivity so that the network interface card can be sent in a low power mode (i.e., the card is in the ***sleep*** state[1]) without needing to periodically wake up to synchronize with the access point. Indeed, we have to remember the low-power mode defined in IEEE 802.11 specifies client must awake periodically to fetch potential buffered frame from the access point[2].

By monitoring the TCP receive buffer occupancy, we can detect two cases where TCP experiences inactivity: when the buffer is ***full*** and when the buffer is ***empty***. The proposed technique is convenient as well for streaming applications as for non-streaming applications. Indeed, for streaming applications, the receive buffer has a high average buffer occupancy while for non-streaming applications, it has a low average buffer occupancy. The empty and full buffer conditions seem thus to be convenient to detect periods of inactivity in the case of streaming applications (full buffer condition can be raised) as in the case of non-streaming applications (empty buffer condition can be raised).

What happens to TCP in the case of full buffer?
We saw in chapter 5 the flow control mechanism implemented in TCP. This mechanism prevents a fast host from taking all the buffers on a slowest host by sending a *zero* and a ***non-zero window advertisements***. When the buffer is full, the receiver sends ***a zero window frame*** to advertise the sender not to send any more TCP segments. The receiver knows thus that it will not receive any more packets from the sender. To avoid the silly window syndrome (SWS)[3], the receiver waits before sending a ***non-zero frame***, which allows the sender to take again the sending of TCP segments. This causes a decreasing of the receive buffer occupancy since the sender does not send any more TCP segment and the application continues to drain the receive buffer. This situation is showed on the figure 6.1 for a FTP transfer of 1.2 MB file. The occupancy buffer significantly deceases at approximately the 28th second until the ***non-zero frame*** is sent by the receiver and that the rate of data coming from the network is higher than the rate of data read by the application. At this time, the buffer occupancy increases again. The latency during which no more TCP segments are sent lasts about 3-4 seconds.

[1]We could replace this ***sleep*** state by the ***off*** state if we wanted it but in this last case, the card would be no more powered at all. See chapter 3, section 3.6 for further details.

[2]See chapter 1, section 1.4.3 for more details.

[3]See chapter 5, subsection 5.1.5.

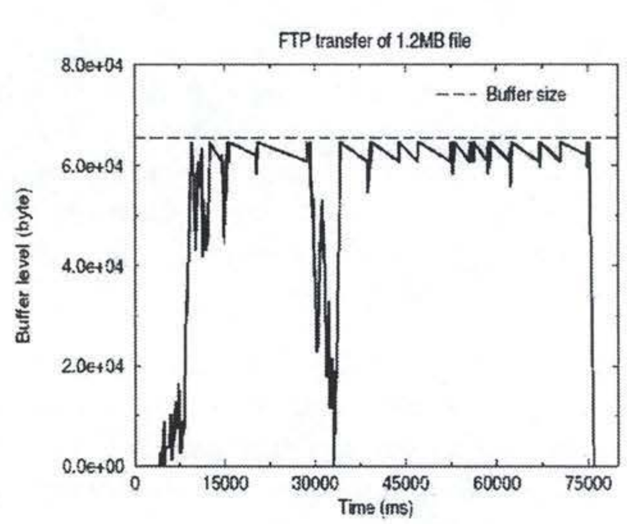

FIGURE 6.1: Buffer occupancy for an FTP download to FLASH memory

In order to save energy, the card can be thus shutdown during this latency. Moreover, this latency can be tuned: the SWS avoiding TCP mechanism can be tuned to extend the idle period by delaying the sending of this ***non-zero window advertisement***. We have just to modify the TCP threshold at which the receiver sends this ***non-zero window advertisement***. The lower bound of this threshold is zero buffer occupancy, i.e. , the case wherein the card is recovered from ***sleep*** state only when the buffer is completely drained by the application. In order to continue avoiding the window silly syndrome, we consider as the upper bound the default TCP threshold.

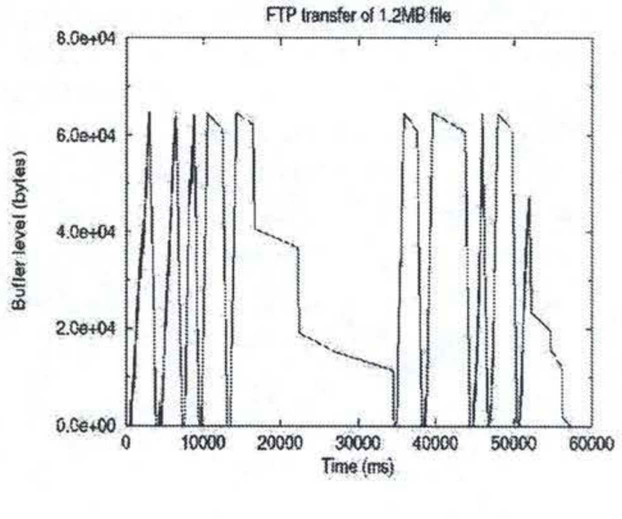

FIGURE 6.2: Buffer occupancy with a zero buffer occupancy condition

We can consider the figure 6.2 that describes the buffer occupancy for the same FTP transfer when the threshold is fixed at zero buffer occupancy. The latency lasts more than 15 seconds in this case, which allows to realize significant energy savings.

It is worth noting that the download of a file on the iPAQ requires the Flash memory to store it. Because writing a Flash memory is a slow operation[4], the receiver buffer has a high average occupancy. On the other hand, streaming applications such as streaming multimedia consists of the transfer of continuous streams of data and a heavy processing step. The streaming applications can be thus very easily *emulated* on the iPAQ by means of a transfer file because of the continuous streams of data and the slow write Flash operation that can emulate the heavy processing step. Moreover, referring to write accesses into Flash memory allows to apply the considerations above also to computation intensive applications.

What happens to TCP in the case of empty buffer?
The buffer empty condition may appear due to network inactivity. This situation may thus appear with non-streaming applications: for instance, when we click on a hyperlink on a web page, the traffic is bursty, while reading the web page, no more traffic is generated. We can see the impact of this traffic on the buffer occupancy on the figure 6.3.

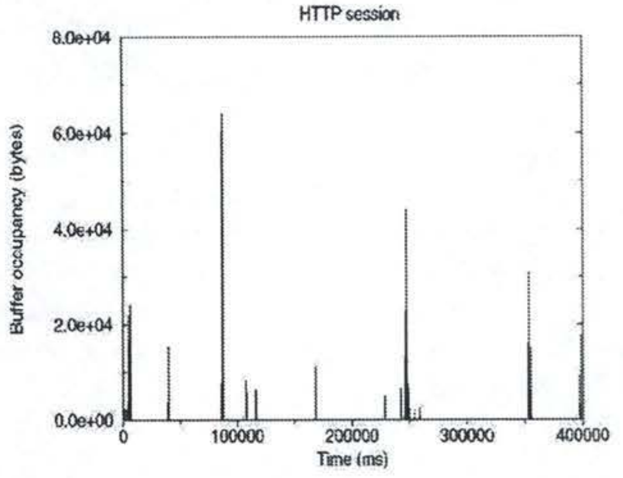

FIGURE 6.3: Buffer occupancy profile for an HTTP session

The main problem is the predictability of these idle periods. Indeed, even if the buffer is empty, packets may be in transit over the network. Thus, if we shutdown the card in these idle periods, the probability to lost packets is high because of the non-predictability of these incoming packets. The empty buffer condition may appear also when we experience delays on the network because of congestion for instance. The predictability of receiving packets in this case is also low. For these reasons, we decide not to take into account the buffer empty condition when we are waiting data but rather considering the buffer empty condition when **nobody/no application is going to use the network**. This results in a better predictability of the traffic flow. We can exploit the "knowledge"that we have on the application workload at TCP layer to predict when an application is going to use the network.

[4]See chapter 2, section 2.3 for further details on this topic.

We can indeed rely on the calls of socket functions offered to applications to know when applications expect to receive data and especially to *tcp_recvmsg()* function. If the buffer is empty and that no socket is actively using the network, we can predict that no segments are expected at the receiver side. A timeout mechanism can be efficiently employed in this context.

The technique relies on the Linux TCP *tcp_recvmsg* function, that transfers data from the TCP buffer to the user buffer (i.e., application buffer), and is called by the application-level *read()* instruction on a certain socket. We have to remember that the application must specify as parameter the amount of data to be read. The high-predictability of this technique implemented at transport layer comes from the fact that we know exactly the amount of data expected by the application (or data to be sent for the application) when it specifies as parameter the amount of data to be read (or to be sent).
When *tcp_recvmsg* returns, a buffer occupancy check is carried out and if the buffer is empty, a card shutdown timer is activated. This timer can be reset and re-activate if subsequent read or write operations occur before it expires. Otherwise, a timer handler is called that switches off the network interface card, because we predict that the card is not likely to be used for a long time. The card is re-activated when a new TCP packet transmission or read operation from the TCP buffer has to be carried out. A proper timeout value should be set.

We can note two overheads. First, the latency associated with card switching. When a TCP operation is carried out while the card is sleeping, the card has to be awaken and so, it takes some time to be again operational (about 300ms for a Cisco Aironet 350 card). Second, after a card shutdown, new packets try to reach the receive buffer before the TCP read operation is scheduled: in this case, they cannot be received because the card is sleeping, and would be re-transmitted by the server at the expiration of the retransmission timeout (TCP is a reliable protocol, see chapter 5).

We have not yet specified if the socket is blocking or non-blocking. The execution flow is in fact different according to whether the application uses ***blocking*** sockets or ***non-blocking*** sockets. For ***blocking*** sockets, when the application calls the *tcp_recvmsg* function with an amount of data to be read as a parameter, the execution flow gets out only when all data has been read[5]. For ***non-blocking*** sockets, the parameter that specifies the amount of data to be read is dynamically changed to 0, indicating that no time has to be spent inside the function waiting for incoming data. In this case, the function reads the amount of data effectively stored in the TCP buffer, and returns even though this amount is less than required. In both cases, after getting out of the function, low-power TCP triggers a timeout if the buffer is empty. However, if all data specified as parameter by the application are not read because there are

[5]We must be more precise because there are "two types"of blocking sockets: in Linux, the sockets are *blocking* by default but a call to a *default* blocking socket returns immediately when there are data available in the socket receive buffer. The call is *blocked* only when there are no data in the socket receive buffer and so blocks until data are available. Of course, it is possible to block the call until the requested data can be returned. For that, an additional socket option must be specified. In this thesis, in general (except in this chapter), the blocking socket corresponds to the default Linux blocking socket.

not enough data in the buffer, for ***blocking sockets***, the ***tcp_recvmsg*** function blocks and so, this prevents to shutdown the card in case further operations have to be carried out, while for ***non-blocking*** sockets, it allows pending data to be stored in the receive buffer, if a sufficiently significant timeout value has been specified[6]. However, we cannot say that, because the timeout cannot be set for ***blocking sockets*** as quickly as for ***non-blocking sockets***, energy savings will be less significant. We have to remember that the amount of data to transfer is the same in both cases (blocking and non-blocking socket cases). It means that, if all data cannot be read in only one ***tcp_recvmsg*** operation, the function will effectively return sooner in the case of ***non-blocking*** sockets but there will be also a smaller sleep time in this case (between the last packet arrival and the next ***tcp_recvmsg*** operation).
Experiments have to be carried out to determine which of these two cases saves the most energy and in which situation. We can see indeed that there are advantages and drawbacks for both cases. We can still note that the utilization of ***blocking*** or ***non-blocking*** sockets by the application (programmer) can have an impact on energy savings.

We will present in the next section energy savings that can be made with the modified low-power TCP (lpTCP). It is worth noting that experiments are performed using the three in-built MAC layer power management policies of the Cisco Aironet network interface card. Policies at MAC and TCP layers are compatible, lpTCP switching off the card when one of the conditions seen above appears. Experiments are also performed using ***non-blocking*** sockets.

6.2.2 Experimental results

The experimental testbed used to validate the proposed protocol optimization policies consists of a Compaq iPAQ H3800 PDA, provided with network connectivity by means of a Cisco Aironet 350 wireless LAN adapter. A PC card extender from Sycard Inc. is used to measure the current drawn by the network interface: a 1 ohm resistor is placed in series with the card's power supply, and a data acquisition board samples the current flowing through the resistor, allowing software processing by means of Labview from National Instruments Inc. The Linux Familiar distribution (Linux 2.4) runs on the iPAQ. For all experiments, the timeout value is set to 4s.

At buffer full:

A FTP transfer of 600kB file is performed. As we have noted above, the buffer full condition has a high probability to be raised because of slow Flash writing operation. Between the zero and non-zero window advertisements, when the sender is not supposed to transmit packets, the card is completely shut down.

[6]If it is not the case and the card shutdowns, lost packets will be retransmitted by the sender because of TCP.

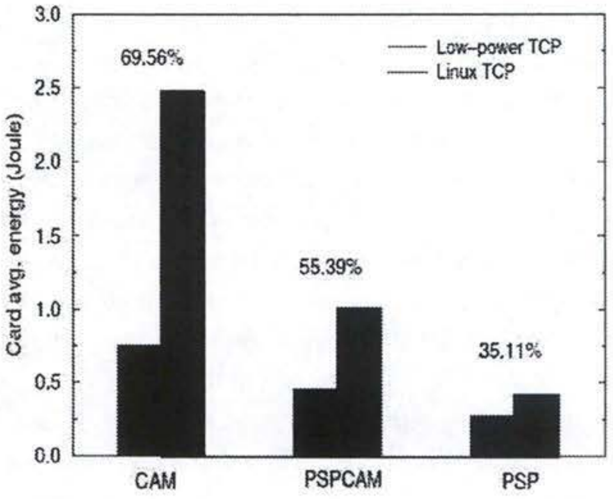

FIGURE 6.4: Energy savings of the proposed lpTCP compared to Linux TCP.

As we can see on the figure 6.4, lpTCP achieves the highest energy savings with the CAM mode because we can put the card to sleep state for periods of the order of seconds instead of keeping it in idle mode. In PSPCAM mode, energy savings achieve 55.39%. Energy savings are less important than in CAM mode, because, during periods of inactivity, PSPCAM switches the card to PSP. It consumes thus less energy than in CAM mode but consumes some energy because the card wakes up periodically to have buffered packets at the access point forwarded. In lpTCP, this latter case occurs fewer times because when the buffer is full, the card is completely shutdown until a non-zero window is sent instead of periodically awakening. In PSP, energy savings achieve 35%, which is yet significant considering the buffer full condition occurs fewer times than in the case of CAM or PSPCAM and considering card consumes less energy than in other modes.
We have said that the threshold (to which a non-zero window is sent after a zero window) was tunable. In fact, if the threshold is set to 0 (it means that the card is only awaken when the buffer is completely empty), the application must wait to receive packets (because of completely drained buffer) and partially depletes the energy savings obtained by switching off the card. On the other hand, when the threshold is set very high, a large number of power mode transitions take place and the available idle periods are very small. This increases the energy dissipation. The best energy savings can be achieved when the threshold is set in between.

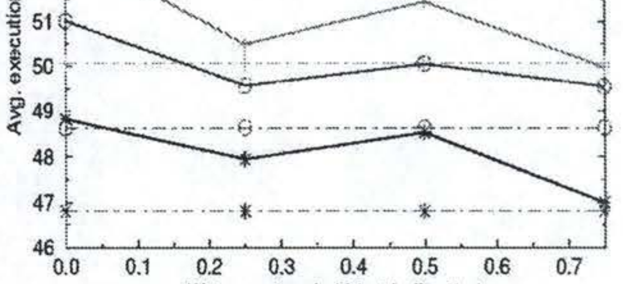

FIGURE 6.5: Performance impact

We must not forget that a power management technique leads always to a lost of performance (see figure 6.5). The performance lost, measured in term of delay, is at most 5.3%, which occurs when lpTCP is used with PSP, and with zero wake-up threshold. For high wake-up thresholds, the execution times tend to converge, because the card activation overhead is hidden by the high value of the TCP buffer occupancy when the threshold is crossed. While the card is coming up, the application can still rely on a large amount of buffered data to read. On the contrary, as the wake-up threshold becomes smaller, the remaining buffered data might not be sufficient and the application might end up waiting for new packet arrivals.

At buffer empty:

A web browsing of 25 minutes is performed. Energy savings depend on the content of web sites. Indeed, with text dominated content, the card entering sleep state immediately after a quick download while with image oriented content, the card remains active for a longer period because of image time-consuming downloads.

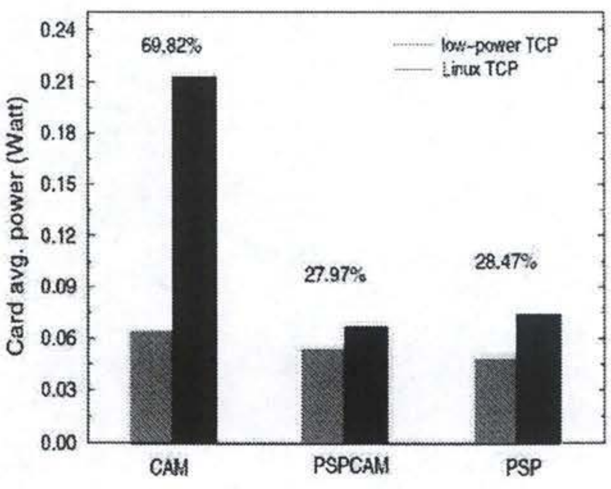

FIGURE 6.6: Power consumption of lpTCP versus linuxTCP for a HTTP session

As we can see on the graphic 6.6, significant energy savings can also be made for non-streaming applications.

6.3 Conclusion

In this chapter, we have seen a power management technique that can be implemented at the TCP Layer. This layer provides a high visibility of the application workload. Indeed, on the contrary, when we work on the driver level, we only know that a transaction has just occurred, but we do not know anything else. So, we cannot guess what the application is going to do. Therefore, we run the risk each time we decide to shut down the card of having to power it on again a few instants of time later. The predictability is thus very bad and costs in waste of power. On the other hand, at TCP layer, we do not know anything about the channel conditions. Therefore, we run the risk of allowing transmissions while the channel quality is bad and so, we also waste power because only retransmissions take place in this case. The power management presented in this chapter cannot be thus adaptive on channel conditions. Because of TCP, this technique cannot be also applied to UDP, which does not provide the same in-built mechanisms than TCP. Each power management technique has thus its own advantages and drawbacks and are also specific (for a specific application, traffic profile, protocol,...). But specific does not mean non-compatible and several techniques at different layers could be even applied at the same time to decrease more energy. However, in this case, it also requires a study of the trade-offs involved and the customizations required for unified operation.

The experimental results of the technique presented in this chapter shows us that we can significantly save the power consumption of the wireless card without causing significant losses of performance. In both cases seen above, the power savings achieve at most about 70% when the card is in CAM and at least 28%, in PSP. The power management technique is thus compatible with the low-power policies implemented at MAC layer and increases its effects on the power savings. These experimental results are important because they allow us to characterize the power consumption of the card with a specific power management technique implemented at TCP layer above the in-built MAC power techniques when the iPAQ runs a streaming or an interactive application.

In the next chapter, we will characterize the power consumption of the different in-built MAC layer power management techniques implemented in commercial wireless cards to make up for the lack into the open literature of real measurements about these ones. No such data can be indeed found in the majority of the commercial datasheets. However, this kind of information is important to designers to know where it is worth working to decrease power.

Chapter 7

Characterization of the Cisco Aironet policies

7.1 Introduction

In this chapter, we characterize the power consumption of the in-built power management policies of commercial wireless cards. The values that we can find in datasheets do not refer indeed to these policies but, if they are available, rather to the different states that are supported by the cards. We cannot thus efficiency evaluate the power/performance trade-off of these policies, which, according to the protocol stack, are implemented at MAC layer. For instance, no statistics are indeed available in the literature about the ratio of power consumed by a streaming application in presence of the different modes of working (for the Cisco card: PSPCAM, PSP and CAM). Only qualitative considerations are discussed.
Because these values are important for the designers to know where it is worth working, an investigation has been carried out to know how many power savings can be realized when the card is switched from one mode to another and what are the impacts on the performance.

The system is composed of an iPAQ and a wireless LAN Cisco Aironet 350 Series card in a wireless infrastructure network. The iPAQ runs a well-known streaming application during the tests. First, we compare the power savings by switching the CAM mode of the card to the PSP mode of this one. We report then the impacts of these different working modes of the card on the performance. Second, we wonder how the power consumption changes when we modify parameters at the TCP/IP level and especially when we change the length of the TCP buffers. This a soft approach, that relies only on a different configuration of the TCP/IP stack, and thus does not involve drastic changes in the system.

7.2 Description of the mpeg_player streaming application

MPEG is an acronym for Moving Picture Experts Group, a group charged with the development of video and audio encoding standards[1]. Since its first meeting in 1988, MPEG has grown to include approximately 350 members from various industries and universities. MPEG has standardized several compression formats:

- MPEG-1: includes the popular Layer 3 (MP3) audio compression format.
- MPEG-2: video and audio standards for broadcast-quality television. Used on most DVD movies.
- MPEG-4: expands MPEG-1 to support video/audio. "objects", 3D content, low bitrate encoding
- MPEG-7: a formal system for describing multimedia content.
- MPEG-21: MPEG describes this future standard as a Multimedia Framework.

7.2.1 MPEG-1

Its official denotation is now ISO/IEC/JTC1/SC29/WG11. This standard is independent of a particular application and therefore comprises mainly a toolbox. It is up to the user to decide, which tools to select to suit the particular applications envisaged. This implies, that only the coding syntax is defined and therefore mainly the decoding scheme is standardized. Berkeley Multimedia Research Center (BMRC)[2] developed the first widely-distributed software decoder for MPEG-1 video in November 1992. BMRC developed the *mpeg_play*, which is an MPEG player written in C. This decoder takes a video stream and display it. This video decoder does not handle real-time synchronization or audio streams.

We will use a modified version of the *mpeg_player* application[3] in order to carry out our experiments. There are two types of modifications. The first modifications consist mainly of the implementation of a *preroll-buffer* avoiding the fluctuations of bit-rate during transmission of data. The goal was to develop a working streaming application over a wireless network in a client-server architecture[4]. The second modifications are needed in order to carry out statistics.

7.2.2 The preroll-buffer

First modifications:
At the client side:

[1]`http://www.mpeg.org/`

[2]`http://bmrc.berkeley.edu/frame/research/mpeg/`

[3]The modified version of this application can be found on the CDROM provided with this dissertation.

[4]The streaming application has been realized by Luca Natali for his master thesis in electronic engineering at the Faculty of Engineering in Bologna (Italy).

A part of the application code can be found at the appendix(protosocket.h, client.c and server.c files). The *preroll-buffer* is implemented in the *protosocket.h* file. The main important file is *client.c* that initializes the *preroll-buffer* and defines the access *recv()* function to the socket[5] and that copies data in the *preroll-buffer*.

At the server side:
The *server.c* file implements the concurrent server.

The functioning of the application is depicted on the figure 7.1.

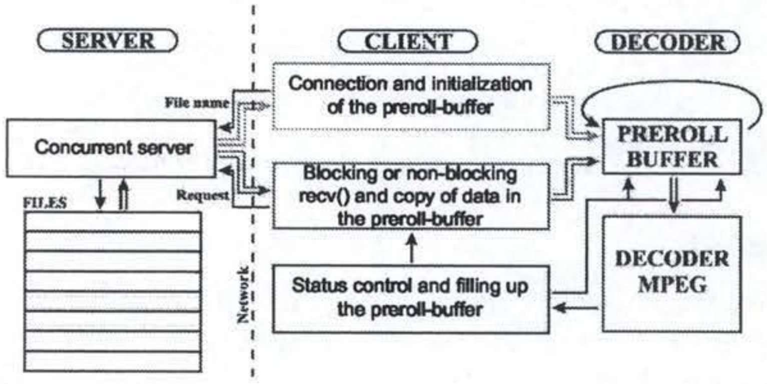
SERVER
CLIENT
DECODER
Connection and initialization of the preroll-buffer
File name
Concurrent server
PREROLL BUFFER
Blocking or non-blocking recv() and copy of data in the preroll-buffer
Request
FILES
DECODER MPEG
Status control and filling up the preroll-buffer
Network

FIGURE 7.1: mpeg_play application scheme

The client connects to the server and sends a file name to the server. The server answers to the client by sending the size of the requested file. The client receives data from the server and copies them into the *preroll-buffer*. When the buffer is full, the video decoder reads data from the *preroll-buffer*. The video decoder reads the *preroll-buffer* as long as the difference between what was already read from the *preroll-buffer* and what is not it yet do not fall below a REFILL value (see the constant REFILL defined in *client.c*). When this value is reached, the application reads from the socket to fill up the *preroll-buffer*. It means indeed that the video decoder drains too fast the *preroll-buffer* compared to the filling up. If the difference is again above this REFILL value, the application reads from the socket and fills up again the *preroll-buffer*. If the difference falls under a DANGER value (see the constant DANGER defined in *client.c*: DANGER < REFILL), the socket is blocked and the application is waiting data from the socket.

7.2.3 Performance statistics

Second modifications were brought:
At the client side:

[5]That is the function that allows the application to get data from the network.

- We can now execute the application by choosing the IP (i.e., Internet Protocol) address of the server, the name of the statistic file and the name of the mpeg file.
- The application interface buttons like REWIND and EXIT have been disabled[6]
- Each call to the *recv()* function is now "blocking"(in the *client.c* file, see appendix for the code.). This means that the *recv()* function returns only when some data can be read from the socket. The result of this function is to copy data in the *preroll-buffer*.
- The *protosocket.h* defines a new *buf_stat* structure. This *buf_stat* structure can be seen on the figure 7.2.

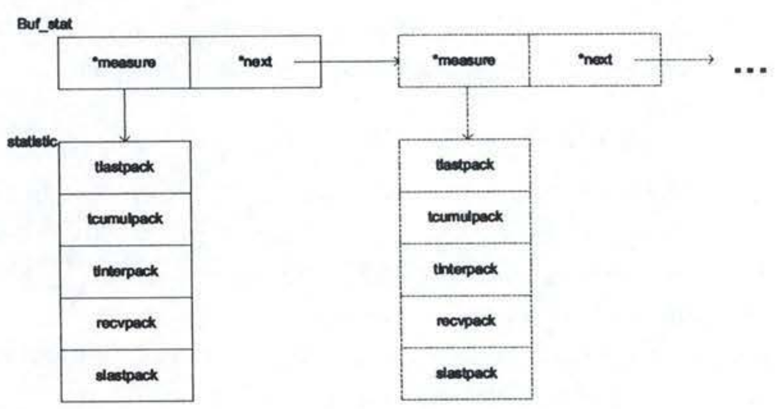

FIGURE 7.2: The *buf_stat* structure

The *buf_stat* structure is a chain of *statistic* structures. Each *statistic* structure contains 5 fields copied into statistic files in this order[7]:

  - stat.recvpack: number of calls to the *recv()* fonction.
  - stat.tlastpack: average time needed by the *recv()* function in order to return 1 byte (in μs)
  - stat.slastpack: amount of bytes returned by the *recv()* function (in *bytes*)
  - stat.tcumulpack: average time to receive "n"bytes during the "n"last calls (in μs)
  - stat.tinterpack: it is a marker of time before the call to the *recv()* function (in μs). The difference between two successive stat.tinterpack allows to know the time passed between two *recv()* calls.

A *buf_stat* structure is filled up with statistics each time the *recv()* function is called.

- At the end of the video, the statistics encapsulated in each *buf_stat* structure are copied into the file specified in the *mpeg_play* command[8].

[6]In the original *mpeg_play* application, it is possible to rewind the video.

[7]The statistic files are available on the CD-ROM provided with this dissertation.

[8]Indeed, the statistics are not written directly in a file on the Flash memory during the video streaming because writing a Flash is a slow operation (See the chapter 2, section 2.3 to see further details about this) and could have some significant impact on measurements.

At the server side:

- The server is continuously running and listening to client requests[9].

7.3 The experimental testbed

- A Cisco Aironet 350 Series Client Adapter for the iPAQ, which runs the Linux Familiar distribution (Linux 2.4.18-rmk3-hh20).
- The card is powered by a current of 5V.
- The Aironet driver for 4500 and 4800 series cards, which is developed by Benjamin Reed[10] and included in the Linux Familiar distribution, is used.
- An Cisco Aironet series 350 access point is used.
- A Sycard PCCextend card[11], inserted into the PCMCIA slot of the Expansion Card of the iPAQ PDA, which connects to the Cisco Aironet 350 wireless LAN card. This Sycard PCextend card is used to measure the current drawn by the network interface: a 1 ohm resistor is placed in series with the card's power supply, and a data acquisition board samples the current flowing through the resistor, allowing software processing by means of Labview[12] from National Instruments Inc.
- A desktop PC borrowed from the Center for Applied Information Science and Technology Department, University of Urbino[13], running Windows 2000 and PIV Processor is used to run Labview and logs the data.

The picture 7.3 shows the experimental set-ups, at the client side.

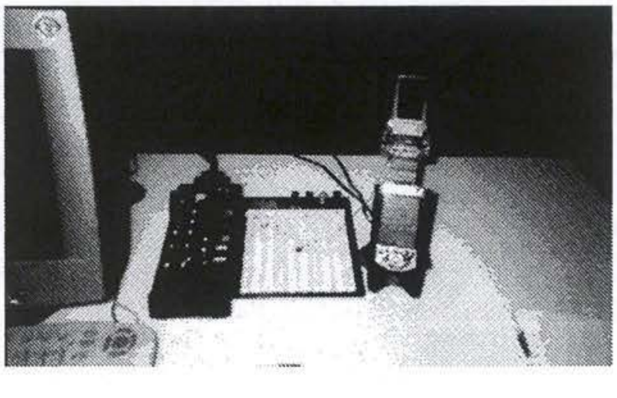

FIGURE 7.3: Testbench

[9]In the previous version of *mpeg_play*, the server was to be started again after each video streaming.

[10]breed@users.sourceforge.net. The drivers can be found at the appendix

[11]For more information: http://www.sycard.com/

[12]For more information: `http://www.ni.com/labview/`

[13]Because of certain problems of hardware compatibility and of organization in Bologna, we carried out our tests in University of Urbino.

- A Fujitsu Siemens laptop computer, PIV 2.00GHz, 512MB RAM, 40GB disk, running Linux Redhat 8 acts as server.
- The mpeg_play streaming application transferred a mpeg file of 374164 bytes across the air for each power mode.
- Approximate distance between iPAQ PDA and the access point was 3 meters and were in direct sight.
- The iPAQ is not powered by its battery but it is connected on electrical sector[14]

The figure 7.4 summarizes the topology.

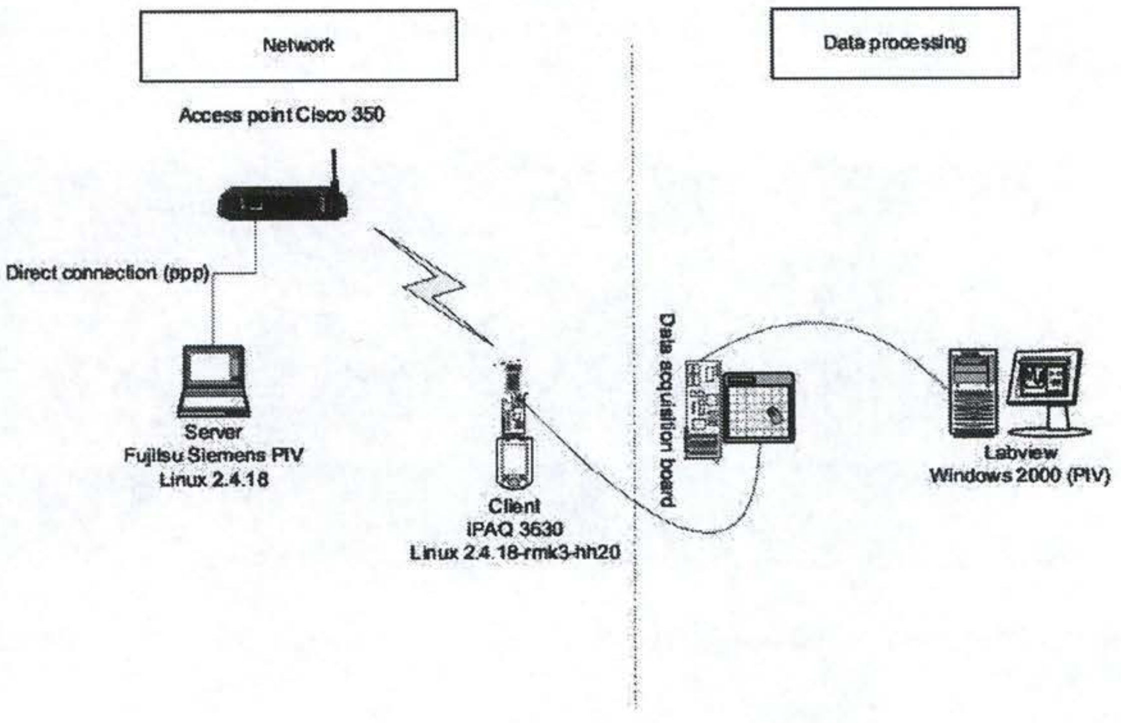

FIGURE 7.4: Testbench and network topology

7.4 Power measurements

We have seen that the Cisco Aironet 350 Series Client Adapter can be in three states: in a run state, off state and a sleep state. These states are used to implement the operating modes CAM, PSPCAM and PSP. These latter modes are the most basic policies that we can imagine and are implemented at the MAC layer of the OSI model: they have not the visibility that

[14]The experiments should be remade without connecting the iPAQ on sector but only when it is powered by its battery. In fact, connecting or not the iPAQ on sector has some impacts on the power consumption.

have the high-layer policies in the TCP/IP stack but they have information about the channel conditions. However, it is often difficult to evaluate their power effectiveness because of the lack of these data in commercial datasheets. We will see the power consumption of the card in CAM, PSP and PSPCAM.

- We take 10 measures for each mode (CAM, PSP, PSPCAM).
- The power consumption measurements are starting as soon as the *mpeg_play* is launched.
- In Labview, we fixe the sample rate at 1000 per second and the test lasts 20 seconds.

7.4.1 CAM

We can see on the figure 7.5 the power consumed by the Cisco wireless card when the card is in CAM mode - it is in an run state (the card is awake all the time) - and when the streaming application is running. The Y-axe is the power consumed by the card in mA. The X-axe is the time.

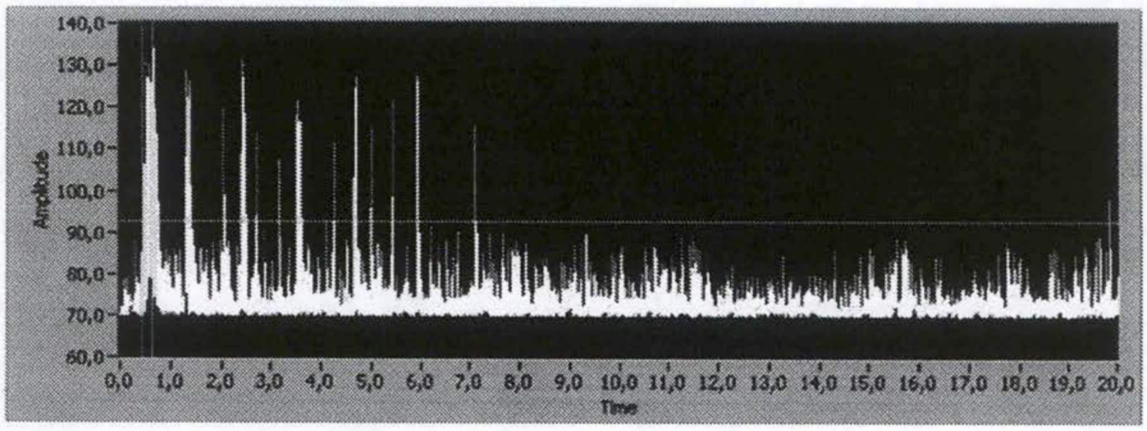

FIGURE 7.5: Power consumed by the card during one of the 20s tests: 72.643408 mA in average

We can see during the first 7 seconds that the card consumes more energy than the rest of the time. In fact, during these 7 seconds, the 374164 bytes of the streaming video are transmitted from the server to the client. The peak values correspond to the power consumed by the card when it transmits some data (acknowledgments because of TCP) and are about 120-130 mA. We can see also that the activity is still more important in the beginning (0 - 1 seconds) and corresponds to the exchanges of segments in the initialization of the TCP connection and also to the sending of the name of the video file and the acknowledgments associated to the received data. Finally, we do not forgotten that the client card receives the beacon signals[15] from the access point during all the tests and that is why, after the transmission of the streaming video data, we can note still peak values. These peak values correspond to the receive power consumption and are about 80-90mA. In addition, we can note that the card consumes about

[15]See chapter 1, section 4 for further details.

70mA during idle periods. Indeed, we know that the card receives about every each 100ms beacon signals but between these periods the card is idle. The "blank bottom line"in the figure 7.5 represents thus the lowest consumption of the card and corresponds to the idle periods.

It is worth noting that, if the card is always in the *run* state, the power consumption can vary: in fact, the card does not consume the same energy when it transmits data or when it receives data or even when it does not receive or transmit anything - idle periods. Referring to the figure 7.5, we can easily understand that the card can be put in a *sleep* or even in a *off* state when the card experiences "idle periods". That is the idea of the other policies: PSP and PSPCAM.
We have repeated the experiment 10 times to eliminate spurious voltage variations[16]. The average power consumption obtained is **72.6467** mA.

7.4.2 PSP

We can refer to the figure 7.6 to see the power consumed by the Cisco wireless card when the card is in PSP mode - the card switches from the *sleep* state to the *run* state every 100 ms by default to fetch its buffered frame - and when the streaming application is running. The Y-axis is the power consumed by the card in mA. The X-axis is the time.

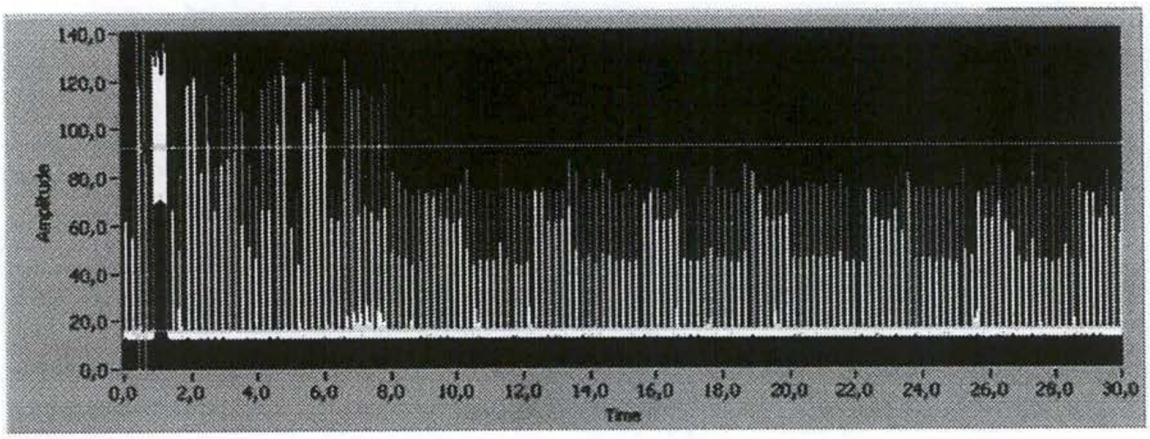

FIGURE 7.6: Power consumed by the card during one of the 20s tests: **19.625456** mA in average

The situation seems to be different from the CAM mode. In fact, we see clearly on the figure 7.6 the effect of the sleeping periods. The card is no more active during periods of 100 ms and the energy falls to more or less 12-15 mA. The card wakes up periodically to receive beacons (TIM[17]). We can see that the application "has the same effect on the consumption of the card than in the CAM mode"in the sense that the power consumption keeps the same profile being

[16]We should have had to undertake more experiments for all the modes to efficiently eliminate spurious voltage variations. However, for reasons external with our will, we have had to carry out our experiments in Urbino and we could not conclude some of our experiments.

[17]See the chapter 1, section 4 for more details.

adapting by the PSP mode. Indeed, all data are still transmitted quickly in the beginning. At 0-1 seconds, the card remains for a longer time active to retrieve its many buffered frame in the access point. These many frames are due to the TCP connection, the name of the file, the acknowledgments associated to the many sent frames. We can notice also that the power consumed during transmitting is about 120-130 mA and about 80-90 mA for the receiving like in the case of CAM. It is normal because the only difference between CAM and PSP is the management of the different states of the card. Therefore, in ***run*** state, the card consumes always about 120-130 mA in order to transmit and about 80-90 mA for receiving. Instead of being in a ***run*** state when the card experiences idle periods, the card, in PSP mode, is put into the *sleep* state. The cards consumes little energy in this case, only about 12-15 mA (that is the blank line at the bottom of the figure 7.6). Finally, we can see also that the client continues to receive beacon signals (after the transmission of video frames, we can see clearly that the card continues to receive some frames: the energy consumed is about 80-90mA) during all the test like in the CAM mode while it exploits the idle periods to sleep.

The client must not need to transmit any frame to check if there are buffered frames and energy savings can consequently be achieved thanks to that. In the ***run*** state, the difference of consumption between transmitting and receiving is more than 30 mA on average. The IEEE 802.11 has thus made a good choice not to wake up the card for transmitting but for receiving (the TIM) in order to see wether if there are buffered frames or not. Of course, as we can imagine from the figure 7.6, we can go further to reduce energy consumption by not to wake up periodically if we are not really sure to find buffered frame (Indeed, after the transmission of video frames, it is no need to wake up periodically because there are not buffered frames in the access point). That is the idea of certain high-layer power management techniques, like the one we have seen in the chapter 6.

We have also repeated the experiment 10 times. The average power consumption obtained is 19.6261 mA.

7.4.3 PSPCAM

The PSPCAM mode is specific to the Cisco cards. The card switches automatically from the CAM mode to PSP mode when the traffic is low and from PSP to CAM when the traffic is high. The figure 7.7 plots the power consumed by the card in this mode.

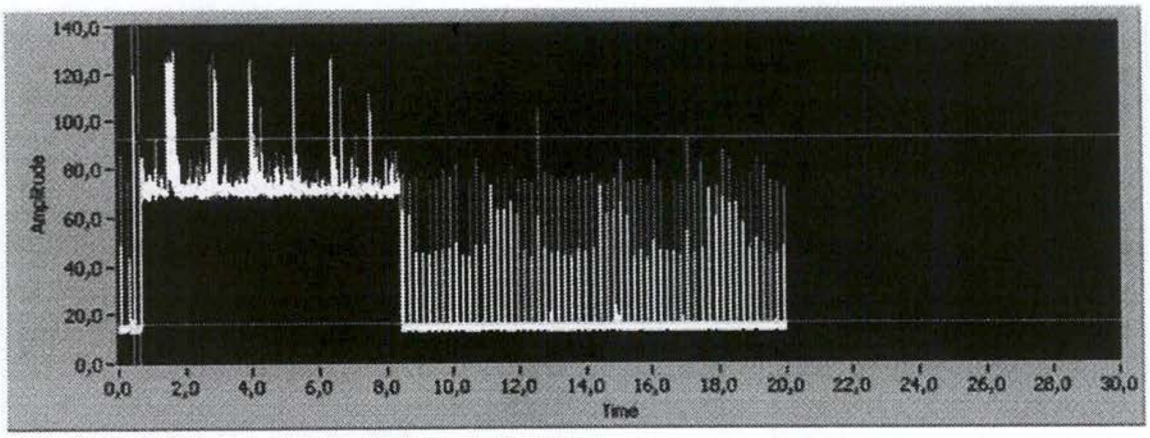

FIGURE 7.7: Power consumed by the card during one of the 20s tests: 38.541443 mA in average

The power consumption profile of the card is not surprising: we can see clearly that the card follows automatically the CAM policy during the video frame transmissions and then follows the PSP policy when all data have been transmitted. We can see that the card is always in a *run* state during the CAM mode and so, no power savings are made during idle periods. The card consumes at least 70 mA during these periods. When all streaming video data are transmitted (about 7-8 s; we have indeed to take into account a time (= cost) during which the card must realize that the traffic does not require any more to keep the card in the run state - CAM mode and puts again the card into the PSP mode.), the card switches into the PSP mode. In PSP mode, during idle periods, the card is put into the *sleep* state and so, the power consumption falls to 12-15 mA. However, again, we can see clearly that the card continues to wake up to receive beacon signals (TIM).

We have repeated the experiment 10 times. The average power consumption obtained is 38.4718 mA.

7.4.4 Summary

In our measurements, we have characterized the energy consumption between the different states of the card: the *run* and the *sleep* state.

- Run state:
  - Transmitting: the card consumes about 120-130 mA or 600-650 mW[18] on average.
  - Receiving: the card consumes about 80-90 mA or 400-450 mW on average.
  - Idle (periods): the card consumes about 70 mA or 350 mW.
- Sleep state:

[18]The magical formula is $W = V * I$. V = Volt I = Ampere

- The card cannot transmit or receive data in this mode and the consumption is about 12-15 mA or 60-75 mW.

We have not "characterized"the power consumption of the *off* state because the card does not consume in this state! Power management techniques that use this state to implement policies are ensured to save a lot of energy but they must take into account that the card takes a few seconds to wake up to *run* state instead of 300 ms to wake up to *run* state from the *sleep* state.
The difference between the policies (CAM, PSP, PSPCAM) are the "utilization"of the different power states. In PSP mode, the card exploits the idle periods to put the card into the *sleep* state and the energy savings can be significant: during our experiments, the average difference between the power consumed in CAM and the power consumed in PSP are about $(72.6467 - 19.6261 =)53.206$ mA. The difference between PSP and PSPCAM is less important but reaches $(72.6467 - 38.4718 =)34.1749$ mA.
It confirms that the effectiveness of the power management techniques depends mainly on the exploitation of the idle periods and the power consumed by the card when it is in a *sleep* state while the total consumption of the card depends on the power consumption of the specific card in all modes. The PSP mode for the Cisco Aironet 350 card can save up to 53.206 mA compared to CAM mode; this last value can be compared to another commercial cards.

How much power does iPAQ consume?

The power consumption of the iPAQ is basically about 470mW in "idle mode"[19]. Taking our measurements into consideration, the power savings that we can realize with the wireless Cisco Card in the PSP mode compared to the CAM mode are around 266 mW. We can realize that this value can be significant compared to the basic 470 mW of the iPAQ (power consumption without wireless card installed). The low-power policy defined in IEEE 802.11 allows thus to save significant power. The specific Cisco PSPCAM mode is also interesting in the sense that the power savings reach 171 mW.

We will see in the next section the performance impact of these different policies on the video streaming application that runs on the iPAQ.

7.5 Performance measurements

If power savings can be made by switching the card from the CAM mode into PSP mode, we must see the impact that this latter mode has on the performance. We must find a criterion that allows us to evaluate the impact of the policies on the performance. The mpeg_play is

[19]The iPAQ runs under Linux, with the LCD display on, and when "most other chips are off"and CPU is idle. iPAQ lasts about 8 hours in this condition. We can note that iPAQ that runs under Linux consumes 1.9 times more power than iPAQ that runs Windows CE does in this condition. When the frontlight brightness is on (fixed at the value 16), the iPAQ consumes 460mW more power. Source: `http://pads.east.isi.edu/presentations/misc/sjcho-pm-report.pdf`

a video streaming application: the performance will be often evaluated by the user by taking into account the average number of frames by second that are displayed. In other words, in this case, the performance is only measured by the average frame rate. In fact, it is not a good indicator for the majority of streaming applications because, when a frame is not processed in time (because of data losses for instance), techniques are applied to display the frames keeping the same rate but in less quality. A good criterion is thus the number of frame violations or the latency between frames. Because a frame can be composed of a variable number of bytes, the performance will be evaluated in term of bytes and not in term of frames[20].
Indeed, in our experiments, we can notice that the frame rate is about 13.5 (frames/s) for the CAM mode and about 13.25 (frames/s) for PSP. It is so negligible that we could prematurely conclude that the impact of PSP compared to CAM on the performance is quasi null. Nevertheless, if we consider the statistics, we see that the 374164 bytes are transmitted in about 6.1 s in CAM mode and in 7.2s in PSP mode. The PSPCAM transmits the data in 7.0s. As we can see, PSPCAM does not really offer a good alternative here in term of performance between PSP and CAM in the sense that it takes almost as many time as in PSP.

The number of calls to the *recv()* function needed to get these 374164 bytes is on average 20 for CAM, about 104.5 for PSP and about 23 for PSPCAM. Because of the little difference in time between the modes (PSP/PSPCAM versus PSP), the amount of bytes received by the application per call is thus less important in PSP than in CAM and in PSPCAM. It confirms what we already know about the trade-off power/performance. More energy is saved, more the performance will be affected. In term of number of calls, PSPCAM is closer to CAM this time.

An interesting value is the average time (in μs) to get 1 byte when the application calls the *recv()* function[21]. We see that in CAM mode the application must wait on average $0.7437\mu s$ before receiving 1 byte and thus before the function returns. In PSP, the value is $2.8733\mu s$. In PSPCAM, the value is $0.7560\mu s$. We can conclude that the application must wait on average longer in PSP mode to find data available in the receive socket buffer than in CAM and in PSPCAM mode. The PSPCAM mode seems to be as well powerful as the CAM mode.

Another interesting value is the amount of bytes received per μs, we can note that the *preroll-buffer* can be fill up at more or less $0.0611 bytes/\mu s$ in CAM mode. In PSP, this value is $0.052 bytes/\mu s$. In PSPCAM, this value is $0.0534 bytes/\mu s$. The difference between PSP and CAM is predictable when we take into account the other indicators. On the contrary, PSP-CAM seems here again to resemble PSP. If the waiting time to get 1 byte when the application

[20]The values presented here have been computed thanks to the statistics collected in the *buf_stat* during the execution time of the *mpeg_play* application. All these files are available on the CDROM provided with this dissertation.

[21]We have to know that the *recv()* function returns immediately when there are some data in the receive socket buffer (see chapter 5 for more information on this topic and at top of this chapter) and thus the amount of bytes copied into the application buffer (preroll-buffer) is variable and depends really on what there are in the socket buffer and on TCP. In order to compute the time needed to get 1 byte when the application calls the *recv()* function, we have divided the execution time of the *recv()* function by the amount of bytes returned to the application by this function.

calls the *recv()* function is on average quasi equal between CAM and PSPCAM ($0.7437\mu s$ for CAM and $0.7560\mu s$ for PSPCAM), the amount of bytes received per μs in PSPCAM is closer to PSP than CAM. Because of the call to *recv()* depends on the application, we cannot say that the average time between two calls is more important in PSPCAM than in CAM. In fact, it seems that, the execution call of the *recv()* function takes on average more time than the execution time of this same function in CAM mode. Because the *recv()* function is not blocking[22], it means perhaps that the card keeps longer the receiving data (in its **hardware buffer...** [23]) than the two other modes. Is this related to the fact that the card must evaluate the traffic to know wether it must switch from CAM to PSP or from PSP to CAM? TCP does not seem indeed decrease its transmission rate because the amount of bytes returned by the recv() function is important. The number of calls indicator showed that the value in PSPCAM was similar to the value in CAM. We would be tempted to say that the application spends thus more time on average waiting data from the socket in PSPCAM than in other modes. This could prevent the application to do other works during this time. And what about if the *recv()* function is completely blocked (i.e., the function returns only when all data specified by the user can be read from the socket)?

7.6 Modification of the TCP buffers

We have reproduced the experiments that we have just described by modifying the TCP buffers size. We wanted to see in a general way if, by modifying parameters on the level of TCP/IP, the consumption and the performance could vary.
We based on studies which sought to improve TCP transmissions under Linux 2.4. The size of the buffers that were modified are[24] :

- /proc/sys/net/core/wmem_max: 65535 becomes 8388608
- /proc/sys/net/core/rmem_max: 65535 becomes 8388608
- /proc/sys/net/ipv4/tcp_rmem : 4096-43689-87378 becomes 4096-87380-4194304
- /proc/sys/net/ipv4/tcp_wmem: 4096-16384-65536 becomes 4096-65536-4194304

As expected, we noted an improvement of the performance and this, for each mode, since according to the indicators, we have:

- The number of calls needed on average for CAM is about 15.3 while for PSP is 94 and for PSPCAM 17.5. We can see that the "order"is respected.

[22]Normally, the function returns indeed immediately as soon as there are data available in the **socket receive buffer**.

[23]... Or the hardware buffer of the access point? We do not really know how this specific Cisco policy works and how it could interact with the access point. Indeed, we have experienced the case that the PSPCAM mode does not work at all with the other non-Cisco access points, so it means that the wireless Cisco Client Adapter needs to interact with the access point in order to work properly with the PSPCAM mode.

[24]Source: `http://www.psc.edu/networking/perf_tune.html/`

- The number of seconds needed to transmit the 374164 bytes is about 5.4 s while for PSP is about 6.4 s and for PSPCAM is about 6.3s.
- The average time to get 1 byte when the application calls the recv() function is about 0.69 μs for CAM, 2.95μs for PSP and for PSPCAM is about 0.71μs
- The number of bytes received per μs is 0.069 for CAM, 0.057 for PSP and 0.059 bytes for PSPCAM.

We note this values on average for the power consumption, after 10 measurements for each node:

- CAM consumes 73.445mA
- PSP consumes 20.7799
- PSPCAM consumes 35,7648

The power/performance law seems to be respected for PSP and CAM policies. Indeed by modifying simple parameters in order to increase performance, it has good effects on this last one but has also "bad"effects on energy. Increasing the performance seems irresistibly to increase also the power consumption.
However, for PSPCAM, it seems that it is not the case because it consumes less energy when the buffers are better-tuned[25]. No serious explanation to this phenomenon could be advanced. However, this result is important because, even if the difference of power savings and increased performance is low compared to the values obtained in the previous set of experiments (i.e., when the buffers are not modified), it means that "tuning"could improve the performance but also the power consumption at the same time. It would be just a better way of managing buffer that would positive effects as well on performance as on the power consumption.

7.7 Conclusion

We can conclude that there is a real trade-off between the power consumption and the performance. When the card is in CAM mode, the performance is high but the energy consumption is also high while in PSP the energy saved is important but the performance degradation is perceptible. The PSPCAM seems to be a trade-off between the PSP and the CAM mode as well in performance as in power consumption. However, in our experiments, as we have seen in the previous section, we have noted that the *recv()* call lasts longer in the PSPCAM than in other modes. Even if the performance is better in the PSPCAM than in the PSP, the function returns (with less bytes) more rapidly in the PSP mode than in PSPCAM mode and so, in this last case, this can prevent the application to do other work. However, more experiments should be carried out in order to evaluate what would be the impacts of this mode on different applications.

[25]One more time, more experiments must be carried out to verify the assumption.

Nevertheless, whatever the low-power modes of the card, in our case, the performance degradation on the quality of the video streaming is not perceptible and the average frame/sec is quasi similar to the one obtained in the CAM mode. This shows that a good management buffer at application layer can overcome the delay caused by the different low-power policies of the card[26].

Finally, with our last set of experiments, we have seen that it is possible to tune the TCP/IP stack to increase the performance. If this increasing of performance results in an increasing of power consumption in CAM and PSP, it seems that the "tuning"has an inverse effect on PSPCAM.

[26]It is worth noting that in our case, the *preroll-buffer* (i.e., the application buffer) is quite large (200000 bytes) compared to the size of the video (374164 bytes). That is perhaps why we can notice a so small performance degradation on the quality of the video streaming. Additional experiments should be carried out in order to see the impact of the *preroll-buffer* on the performance.

Chapter 8

Conclusion

We have seen in this dissertation through the iPAQ PDA that the portable devices can deliver high performance by running video streaming applications through the air. We have seen that, through the Cisco Aironet 350 series wireless, that the consumption can be very important compared to the basic consumption of the iPAQ when the card is especially in CAM mode because of the high-consumption level of the ***run*** state. Research should be thus still carried out to reduce the power consumption whatever the mode of the card.

However, in addition to this specific manufacturer improvements that are not always compatible[1], IEEE 802.11 has succeeded to define a low-power policy and to standardize it. However, this standardization refers to the physical and MAC layers and so, the information available for developing dynamic power management techniques at these layers refers more to channel conditions than to the traffic. It is thus useful to develop, in addition to the in-built power management techniques, other compatible techniques that have a visibility of the traffic. In order to develop efficient techniques above the MAC-layer policies, the characterization of the power consumption is important and helps the designers to know where it is worth working in the TCP/IP to decrease power.

This dissertation has characterized the power consumption of the different in-build policies of the commercial IEEE-802.11 compliant cards through the Cisco Aironet 350 Client Adapter when the iPAQ is running a streaming application. We have seen that the low-power policy defined by IEEE 802.11 allows to save significant energy without compromising too much the performance if the streaming application implements a good management buffer and algorithms that tradeoff the frame quality/performance. The specific PSPCAM mode of the Cisco card seems to work differently and needs to be compatible with the access point but presents a tradeoff between CAM and PSP. In addition, most experiments must be carried out in order to decrease the ratio power/performance. We have seen that by tuning the TCP/IP stack, we have achieved to increase the power while increasing the performance.

[1]We refer to the experience of the Cisco Aironet Client Adapter functioning in PSPCAM mode with a non-Cisco Aironet access point.

In the future, another power management techniques should be developed at high layers of the protocol stack and based on experimental results of another lower techniques they should better operate between them in order to decrease energy.

Bibliography

[1] The ARM Linux Project web site: http://www.arm.linux.org.uk.

[2] The Handelds.org web site: http://www.handelds.org.

[3] The Familiar Linux distribution web page: http://familiar.handhelds.org.

[4] Wikipedia The Free Encyclopedia web page: http://www.wikipedia.org.

[5] L. Benini, A. Bogliolo, and G. De Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. IEEE Transactions on VLSI Systems, Vol.8, No. 3, June 2000.

[6] L. Benini, A. Bogliolo, and G. De Micheli. System-Level Dynamic Power Management. VOLTA-99: IEEE Alessandro Volta Memorial Workshop on Low-Power Design, pages 18–23, March 1999.

[7] L. Benini, G. Castelli, A. Macii, E. Macii, and R. Scarsi. Battery-Driven Dynamic Power Management of Portable Systems. International Symposium on System Synthesis, pages 25–33, September 2000.

[8] L. Benini, G. Castelli, A. Macii, and R. Scarsi. Battery-Driven Dynamic Power Management. IEEE Design and Test of Computers special issue on Dynamic Power Management of Electronic Systems, pages 53–60, March/April 2001.

[9] L. Benini and G. De Micheli. Dynamic Power Management: Design Techniques and CAD Tools. Kluwer, 1997.

[10] L. Benini, A. Macii, E. Macii, and M. Poncino. Selective instruction compression for memory energy reduction in embedded systems. In Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'99, August 1999.

[11] D. Bertozzi, A. Raghunathan, L. Benini, and S. Ravi. Transport Protocol Optimization for Energy Efficient Wireless Embedded Systems. Proceedings of the Design, Automation and Test in Europe Conference Exhibition (DATE'03), 2003.

[12] O. Bonaventure. Réseaux - INFO2210, 2001. http://www.infonet.fundp.ac.be/LEARN.

[13] O. Bonaventure. Téléinformatique et réseaux Matières approfondies - info2231, 2001. http://www.infonet.fundp.ac.be/LEARN.

[14] D. P. Bovet and M. Cesati. Understanding the Linux Kernel. O'Reilly, 2001.

[15] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A Dynamic Voltage Scaled Microprocessor System. Proceedings of the International Solid-State Circuits Conference, 2000.

[16] I. Choi, H. Shim, and N. Chang. Low Power Color TFT LCD Display for Hand-Held Embedded Systems. In Proceedings of the International Symposium on Low-Power Electronics and Design ISLPED'02, August 2002.

[17] Craig Hunt. TCP/IPNetwork Administration. O'Reilly, Second Edition, December 1997.

[18] Gary R. Wright and W. Richard Stevens. TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley, 1995.

[19] F. Gatti, A. Acquaviva, L. Benini, and B. Riccó. Low Power Control Techniques for TFT LCD Displays. In Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems CASES'02, October 2002.

[20] Carel Bailey Germain. Programming with C. Mac Graw Hill, 1990.

[21] P. Greenawalt. Modeling Power Management for Hard Disks. International Workshop on Modeling, Analysis and Simulation of Computer ans Telecommunication Systems, pages 62–6, 1994.

[22] IBM and Monta Vista Software. Dynamic Power Management for Embedded Systems, Version 1.1. http://www.research.ibm.com/arl/projects/papers/DPM_V1.1.pdf, 2002.

[23] Karim Yaghmour. Building Embedded Linux Systems. O'Reilly, April 2003.

[24] Y.-H. Lu, L. Benini, and G. De Micheli. Requester-Aware Power Reduction. In International Symposium on System Synthesis, pages 18–23. Stanford University, September 2000.

[25] Y.-H. Lu and G. De Micheli. Adaptive Hard Disk Power Management on Personal Computers. In Proceedings of the IEEE Great Lakes Symposium, pages 50–53, March 1999.

[26] Jeffrey Oldham Mark Mitchell and Alex Samuel. Advanced Linux Programming. David Dwyer, June, 2001.

[27] F. Matià. Writing a Linux Driver, April 1998. http://www.linuxjournal.com/.

[28] Matthew Gast. 802.11 Wireless Networks: The Definitive Guide. O'Reilly, April 2002.

[29] Lar Kaufman Matthias Kalle Dalheimer, Terry Dawson and Matt Welsh. Running Linux, 4th Edition. O'Reilly, December 2002.

[30] Kenneth L. Calvert Michael J. Donahoo. The Pocket Guide to TCP/IP Sockets C Version. Morgan Kaufmann Publishers, 2001.

[31] T. Pering and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. Proceedings of the International Symposium on Low-Power Electronic and Design ISLPED'98, June 1998.

[32] P. Pillai and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. Proceedings of the 18th Symposium on Operating Systems Principles SOSP'01, October 2001.

[33] W. Richard Stevens. TCP/IP Illustrated, Volume 1: The Protocols. Addison-Wesley, 1994.

[34] Rob Flickenger. Building Wireless Community Networks. O'Reilly, January 2002.

[35] A. Rubini and J. Corbet. Linux Device Drivers, 2nd edition. O'Reilly, 2002.

[36] T. Simunic, A. Acquaviva, L. Benini, and G. De Micheli. Dynamic Voltage Scaling and Power Management for Portable Systems. Proceedings of the 38th Design Automation Conference DAC'01, 2001.

Appendix

Appendix A

Protosocket.h

- This file implements the *buf_ stat* structure.
- The modifications which we made follow "Modifications Blight"and end in "End Blight".

```
#define DIM_SOCK 200000

struct sock_buf {
    int i_write;
    int i_read;
    int n_write;
    int n_read;
    int last_write;
    int last_read;
    int lenght;
    unsigned char buf[DIM_SOCK];
};

//Modification Blight
struct statistic {
     float tlastpack;
     float tcumulpack;
     float tinterpack;
     long int recvpack;
     int slastpack;
         };
struct buf_stat {
    struct statistic *measure;
    struct buf_stat *next;
            };

extern int init_sock_buf(char nomefile[], char ip[]);
//extern int init_sock_buf(char nomefile[]);
//End Blight

extern int decide_fill();
extern int exit_socket();
extern int rew_socket();
extern struct sock_buf sb;
```

Appendix B

Client.c

- This file implements the functions that access the socket and that copy data into the *preroll-buffer*.
- The modifications that we made follow "Modifications Blight"and end in "End Blight".

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include "proto_socket.h"
#include <sys/time.h>

#define DIM_BUF1 50
#define DIM_BUF2 50000                /* dim. pacchetti via socket */
#define REFILL 150000                 /* dim. sotto la quale si effettua riempimento */
#define DANGER 50000                  /* dim. sotto la quale si eff. blocking refill */
#define DIM_RIEMP 150000              /* dim. riempimento iniziale buffer */

//Modification Blight
//FILE *fa;
static struct statistic stat;
static struct timeval start_time, end_time;
static struct timezone tz;
static struct timeval start_interval, end_interval;
static struct timezone tz2;
struct buf_stat *head_stat;
struct buf_stat *current_stat;
int counter, cond = 0;
//End
static int sd;
static struct sockaddr_in rem_indirizzo;
```

```
/* chiamata tipo: set_flag(1) */
int set_flag(int value)                          /* setta O_NONBLOCK se value != 0, ripristina BLOCK */
{                                                    /* se value = 0; ritorna 0 in caso di successo */
int oldflags = fcntl(sd, F_GETFL, 0);
if (oldflags < 0)
  return oldflags;
if (value != 0)
  oldflags |= O_NONBLOCK;
else
  oldflags &= ~O_NONBLOCK;
return fcntl(sd, F_SETFL, oldflags);
}


/* connessione e riempimento iniziale buffer */
/* chiamata tipo: init_sock_buf(nomefile) */

int init_sock_buf(char nomefile[], char ip[])
{
int nread;
int n, l;
// char ip[] = "137.204.56.67";
char file_lenght[12];
if (gettimeofday(&start_interval,&tz2) != 0) {
    printf("\n Problem to start the interval time\n");
    exit(1);}
if ((sd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("errore socket"); exit(1); }
bzero(&rem_indirizzo, sizeof(rem_indirizzo));
rem_indirizzo.sin_family = AF_INET;
rem_indirizzo.sin_port = htons(9878);
if (inet_pton (AF_INET, ip, &rem_indirizzo.sin_addr) <= 0)
  {
  perror("errore inet_pton");
  exit(1); }
l = strlen(nomefile);
printf("\nCLIENT 1)%s lungo %d", nomefile, l);
printf("\nCLIENT connect1");
if (connect(sd, (struct sockaddr *) &rem_indirizzo, sizeof(rem_indirizzo)) < 0) {
  perror("errore connect"); exit(1); }
printf("\nCLIENT connect2");
                              /* invio nomefile */
set_flag(0);                            /* blocking-IO */
printf("\nCLIENT flag settato");
printf("\nCLIENT 2)%s lungo %d", nomefile, l);
if ((n = write(sd, nomefile, 20)) < 0) {
    perror("errore write nomefile");
    exit(1); }
printf("\nCLIENT scritti %d caratteri",n);
printf("\nCLIENT scritto nomefile");
printf("\nCLIENT sgd ");
/* while(1) {} ; */
```

```
if ((nread = recv(sd, file_lenght, 12, 0)) < 0) {        /* lunghezza file */
      perror("errore read");
      exit(1); }
sb.lenght = atoi(file_lenght);
printf("\nlunghezza atoi= %d\n", sb.lenght);
// while(1) {} ;
//                            /* riempimento buffer */
sb.i_write = 0;
sb.i_read  = 0;
//Modification Blight
bzero(&stat, sizeof(stat));
current_stat = (struct buf_stat *)malloc(sizeof(struct buf_stat));
if (current_stat == NULL) {printf("MEMORY NOT AVAILABLE ?");exit(1);}
head_stat = current_stat;
current_stat->measure = NULL;
current_stat->next = NULL;
counter = 0;
end_interval.tv_sec = start_interval.tv_sec;
end_interval.tv_usec = start_interval.tv_usec;

while (( (cond = gettimeofday(&start_interval, &tz2)) == 0) &&
(((cond = gettimeofday(&start_time,&tz))) == 0)
&& ((nread = recv(sd, &sb.buf[sb.i_write], DIM_BUF2, 0)) > 0) &&
(((cond = gettimeofday(&end_time,&tz)) == 0))
&& (sb.i_write < DIM_RIEMP)) {

stat.recvpack +=1;
stat.tlastpack = (((float)((((long)(end_time.tv_sec - start_time.tv_sec))*1000000)
+ ((long)(end_time.tv_usec - start_time.tv_usec)))) / nread);
stat.slastpack = nread;
stat.tcumulpack += stat.tlastpack;
stat.tinterpack = ((float)((((long)(end_interval.tv_sec))*1000000) + ((long)(end_interval.tv_usec))));
current_stat->measure = (struct statistic *)malloc(sizeof(struct statistic));
counter = counter + 1;
printf("\n Counter: %d\n", counter);
if (current_stat == NULL)
{printf("\nNOT AVAILABLE MEMORY ?\n");
 exit(1);}

/*Save the current stats*/
current_stat->measure->recvpack = stat.recvpack;
current_stat->measure->tlastpack = stat.tlastpack;
current_stat->measure->slastpack = stat.slastpack;
current_stat->measure->tcumulpack = stat.tcumulpack;
current_stat->measure->tinterpack = stat.tinterpack;
printf("\nStats are copied into RAM\n");

/*New buf_stat structure*/
current_stat->next = (struct buf_stat *)malloc(sizeof(struct buf_stat));
current_stat->next->measure=NULL;
current_stat->next->next=NULL;
current_stat = current_stat->next;
```

```
end_interval.tv_sec = start_interval.tv_sec;
end_interval.tv_usec = start_interval.tv_usec;
sb.i_write = (sb.i_write + nread); }

/*Test after the while loop */
if (cond != 0 ) {
            printf("\nTimer Error ! EXIT!");
            exit(1);
        }

if (nread == 0) {
            gettimeofday(&end_time,&tz);
            /*Attention DIVIDE BY ZERO */
            stat.tlastpack = (((float)((((long)(end_time.tv_sec-start_time.tv_sec))*1000000)
            + ((long)(end_time.tv_usec-start_time.tv_usec)))));
            }
else {
    stat.tlastpack = (((float)((((long)(end_time.tv_sec-start_time.tv_sec))*1000000)
    + ((long)(end_time.tv_usec-start_time.tv_usec)))) / nread);
     }
stat.recvpack +=1;
stat.slastpack = nread;
stat.tcumulpack += stat.tlastpack;
stat.tinterpack = ((float)((((long)(end_interval.tv_sec))*1000000) + ((long)(end_interval.tv_usec))));
current_stat->measure = (struct statistic *)malloc(sizeof(struct statistic));
counter = counter + 1;
printf("\n Counter: %d\n", counter);
if (current_stat == NULL)
{perror("MEMORY NOT AVAILABLE ?\n");
 exit(1);}

/*Stats for last block of bytes received (saved in RAM)*/
current_stat->measure->recvpack = stat.recvpack;
current_stat->measure->tlastpack = stat.tlastpack;
current_stat->measure->slastpack = stat.slastpack;
current_stat->measure->tcumulpack = stat.tcumulpack;
current_stat->measure->tinterpack = stat.tinterpack;
printf("\nStats are copied into RAM\n");

/* Not forget to prepare a new buf_stat structure for the next and...*/
current_stat->next = (struct buf_stat *)malloc(sizeof(struct buf_stat));
current_stat->next->measure=NULL;
current_stat->next->next=NULL;
current_stat = current_stat->next;

 /*...Not forget to save the old time interval */
end_interval.tv_sec = start_interval.tv_sec;
end_interval.tv_usec = start_interval.tv_usec;
if (nread < 0 )
{
    printf("\nError of recv socket during initialisation buffer...EXIT!\n");
    close(sd);
    exit(1);
}
//End Blight

sb.i_write = (sb.i_write + nread); /* lettura ultimo ciclo cmq effettuata (sembra di NO!)*/
sb.n_write = sb.i_write;
printf("\nRIEMPIMENTO: sb.n_write =%d", sb.n_write);
return 0;
}

/* read da socket che gestisce buffer circolare */
/* chiamata tipo: readmia() */

int readmia()
{
int nr = 0;
int ciop;
unsigned char buffer[DIM_BUF2];

//Modification Blight
if (((cond = gettimeofday(&start_interval, &tz2))==0) && (((cond = gettimeofday(&start_time,&tz)))==0)
 && ((nr = recv(sd, buffer, DIM_BUF2, 0)) > 0) && (((cond = gettimeofday(&end_time,&tz))==0))){

  stat.recvpack +=1;
  stat.tlastpack = (((float)((((long)(end_time.tv_sec - start_time.tv_sec))*1000000) +
((long)(end_time.tv_usec - start_time.tv_usec)))) / nr);

  stat.slastpack = nr;
  stat.tcumulpack += stat.tlastpack;
  stat.tinterpack = ((float)((((long)(end_interval.tv_sec))*1000000) + ((long)(end_interval.tv_usec))));

  /*New buf_stat structure*/
  current_stat->measure = (struct statistic *)malloc(sizeof(struct statistic));
  counter = counter + 1;
  printf("\nCounter: %d\n", counter);
  if (current_stat == NULL)
   {
    perror("MEMORY NOT AVAILABLE ?\n");
    exit(1);
   }

  /*Stats for last block of bytes received (saved in RAM)*/
  current_stat->measure->recvpack = stat.recvpack;
  current_stat->measure->tlastpack = stat.tlastpack;
  current_stat->measure->slastpack = stat.slastpack;
  current_stat->measure->tcumulpack = stat.tcumulpack;
  current_stat->measure->tinterpack = stat.tinterpack;
  printf("\nStats are copied into RAM\n");

  /*Not forget for the next the buf_stat structure and the interval time!!!...*/
  current_stat->next = (struct buf_stat *)malloc(sizeof(struct buf_stat));
  current_stat->next->measure=NULL;
  current_stat->next->next=NULL;
  current_stat = current_stat->next;
  end_interval.tv_sec = start_interval.tv_sec;
  end_interval.tv_usec = start_interval.tv_usec;
  //End Blight

  if ((sb.i_write + nr) < DIM_SOCK)
    memcpy(&sb.buf[sb.i_write], buffer, nr);
  else {
    ciop = DIM_SOCK - sb.i_write;
    memcpy(&sb.buf[sb.i_write], buffer, ciop);
    memcpy(&sb.buf[0], &buffer[ciop], (nr - ciop)); } }

//Modification Blight

/*Checks after while loop*/
if (cond != 0)
    {
       printf("\nTimer Error ! EXIT!");
       exit(1);
    }
if (nr == 0) {
    if ((gettimeofday(&end_time,&tz))!=0)
        {
            printf("\nTimer Error ! EXIT!");
                    exit(1);
        }
    stat.recvpack +=1;
    stat.tlastpack = (((float)((((long)(end_time.tv_sec - start_time.tv_sec))*1000000)
    + ((long)(end_time.tv_usec - start_time.tv_usec))))); /* Attention Division by ZERO */

    stat.slastpack = nr;
    stat.tcumulpack += stat.tlastpack;
    stat.tinterpack = ((float)((((long)(end_interval.tv_sec))*1000000) + ((long)(end_interval.tv_usec))));

    /*New buf_stat structure*/
    current_stat->measure = (struct statistic *)malloc(sizeof(struct statistic));
    counter = counter + 1;
    printf("\n Counter: %d\n", counter);
    if (current_stat == NULL)
    {printf("NOT MEMORY AVAILABLE ?\n");
    exit(1);}

    /*Stats for last block of bytes received (saved in RAM)*/
    current_stat->measure->recvpack = stat.recvpack;
    current_stat->measure->tlastpack = stat.tlastpack;
    current_stat->measure->slastpack = stat.slastpack;
    current_stat->measure->tcumulpack = stat.tcumulpack;
    current_stat->measure->tinterpack = stat.tinterpack;
    printf("\nStats are copied into RAM\n");

    /*Not forget for the next the buf_stat structure and the interval time!!!...*/
    current_stat->next = (struct buf_stat *)malloc(sizeof(struct buf_stat));
    current_stat->next->measure=NULL;
    current_stat->next->next=NULL;
    current_stat = current_stat->next;
    end_interval.tv_sec = start_interval.tv_sec;
    end_interval.tv_usec = start_interval.tv_usec;
          }

/*Last checks*/
if (nr < 0 )
         {
        printf("\nError recv socket during transmission...EXIT!\n");
            close(sd);
        exit(1);
         }
//End Blight

return nr;
}

/* streaming */
/* chiamata tipo: fill_sock_buf() */

int fill_sock_buf()
{
int nread;

printf("\nNON B");
set_flag(0);                                    /* non-blocking-IO */
if ((nread = readmia()) > 0) {
  sb.i_write = ((sb.i_write + nread) % DIM_SOCK);
  sb.n_write += nread; }
return nread;
}

/* blocking streaming */
/* chiamata tipo: fill_sock_buf_block() */
/* int fill_sock_buf_block(int sd, unsigned char sock_buf[], int i_write) */

int fill_sock_buf_block()
{
int nread;

printf("\nBLOCC.");
set_flag(0);                                             /* blocking-IO */
if ((nread = readmia()) > 0) {
  sb.i_write = ((sb.i_write + nread) % DIM_SOCK);
  sb.n_write += nread; }
return nread;
}

/* rewind */
/* chiamata tipo: rew_socket() */

int rew_socket()
{
int nread;
char a[12] = "REWIND";

//Modification Blight
/*Not used for the stats...*/
printf("\n FUNCTION NOT USED FOR THE STATS! EXIT!");
close(sd);
exit(1);
//End Blight

decide_fill();                        /* sblocca la write bloccante sul server con una lettura TEMPI???*/
set_flag(0);                          /* blocking-IO */
write(sd, a, 10);
                              /* riempimento buffer */
sb.i_write = 0;
sb.i_read  = 0;
sb.n_read  = 0;
sb.n_write = 0;
sb.last_write = 0;
sb.last_read = 0;
while (((nread = read(sd, &sb.buf[sb.i_write], DIM_BUF2)) > 0)  && (sb.i_write < DIM_RIEMP))
{
    sb.i_write = (sb.i_write + nread);
}
/* sb.i_write = (sb.i_write + nread);   /* lettura ultimo ciclo cmq effettuata(sembra di NO!)*/
sb.n_write = sb.i_write;
return 0;
}

/* exit_socket */
/* chiamata tipo: exit_socket(); inserita in CTRL_EXIT di ControlLoop */

int exit_socket()
{
char a[12] = "EXIT";

//Modification Blight
/*Not used for the stats...*/
printf("\n FUNCTION NOT USED FOR THE STATS! EXIT!");
close(sd);
exit(1);
//End Blight

decide_fill();                        /* sblocca la write bloccante sul server con una lettura TEMPI???*/
set_flag(0);                          /* blocking-IO */
write(sd, a, 10);
close(sd);
return 0;
}

/* decide_fill */
/* chiamata tipo: decide_fill() */

int decide_fill()
{
int diff, i;
int eof = 1;
if ((diff = sb.i_write - sb.i_read) < 0)        /* i_write ï£¡a ciclo buffer success.risp. i_read */
  diff = sb.i_write + (DIM_SOCK - sb.i_read);
else
  diff = sb.i_write - sb.i_read;
  printf(" DECIDE diff=%d", diff);
if (diff < REFILL) {
  if (diff < DANGER) {
    if ((eof = fill_sock_buf_block()) <= 0) {                          /* ??? */
    sb.last_write = 1;
        close (sd);
        printf("\nDECIDE:A chiusura socket sb.n_write=%d\n", sb.n_write);
        printf("sb.last_write=%d\n", sb.last_write);
    }
  }
  else
    eof = fill_sock_buf();
}
printf(" DECIDE eof=%d", eof);
return eof;
}
```

Appendix C

Server.c

- This file implements the concurrent server in the *mpeg_play* application
- The modifications that we made follow "Modifications Blight"and end in "End Blight".

```
#include <stdio.h>
#include <signal.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#define DIM_BUF1 256
#define DIM_BUF2 500                      /* dim. pacchetti via socket */

int set_flag(int desc, int value)          /* setta O_NONBLOCK se value != 0, ripristina BLOCK */
{                            /* se value = 0; ritorna 0 in caso di successo */
int oldflags = fcntl(desc, F_GETFL, 0);
if (oldflags < 0)
  return oldflags;
if (value != 0)
  oldflags |= O_NONBLOCK;
else
  oldflags &= ~O_NONBLOCK;
return fcntl(desc, F_SETFL, oldflags);
}


main(int argc,char **argv)
{
int DIM_RIEMP=39000;                      /* dim. riempimento iniziale buffer */
int sd, fromlen, ns, fd1, fd2, a, b, n;
int dim_file=0;
int flag=0;
struct sockaddr_in mio_indirizzo, rem_indirizzo;
struct hostent *host;
```

```
char buf1[DIM_BUF1], nomefile[22], rew[12], lenght[12];
unsigned char buf2[DIM_BUF2];
int nread, nwrite, scritti=0, oldwrite;

sd = socket(AF_INET, SOCK_STREAM, 0);
bzero(&mio_indirizzo, sizeof(mio_indirizzo));
if (sd < 0) { perror("errore apertura socket"); exit(1);}
mio_indirizzo.sin_family = AF_INET;
mio_indirizzo.sin_port = htons(9878);
mio_indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);
if (bind(sd, (struct sockaddr *) &mio_indirizzo, sizeof(mio_indirizzo)) < 0) {
  perror("errore bind");exit(1);}
printf("\nSERVER listen1");
listen(sd,10);
printf("\nSERVER listen2");

for(;;) {
    //Modification Blight

    dim_file = 0;
    scritti = 0;
    scritti=0,
    //End Blight

    printf("\nSERVER accept1\n");
    ns = accept(sd, (struct sockaddr *) NULL, NULL);
    printf("\nSERVER accept2   ns=%d", ns);
                                /* apertura file */
    set_flag(ns, 0);                       /* blocking-IO */
    if ((nread = recv(ns, nomefile, 20, 0)) < 0) {
      perror("errore read");
      exit(1); }
    printf("\nSERVER ricevuto nomefile:%s lungo %d", nomefile, nread);
    printf("\nSERVER sgd");
/*    while(1) {}; */
    if ((fd2 = open(nomefile, O_RDONLY)) < 0) {
    perror("open file");
    exit(1); }
    while ((nread = read(fd2, buf2, DIM_BUF2)) > 0) /*trovo lunghezza del file */
        dim_file += nread;
    close(fd2);                        /* rewind */
    if ((fd2 = open(nomefile, O_RDONLY)) < 0) {
        perror("open file");
        exit(1); }
    printf("\nlunghezza decimale= %d", dim_file);
    sprintf(lenght, "%d ", dim_file);           /* trasmetto lunghezza del file */
    if ((n = send(ns, lenght, 12, 0)) < 0) {
        perror("errore write lunghezza file");
        exit(1); }
    printf("\nlunghezza in caratteri= %s", lenght);
    nread = atoi(lenght);
    printf("\nlunghezza atoi= %d\n", nread);
//    while(1) {};
                                      /* riempimento iniziale buffer */
   while (1) {
     set_flag(ns, 0);                       /* blocking-IO */
     while (((nread = read(fd2, buf2, DIM_BUF2)) > 0) && (scritti < DIM_RIEMP)) {
   scritti = (scritti+nread);
   send(ns, buf2, nread, 0); }
     if (nread > 0) {
   scritti = (scritti+nread);              /* trasmette gli ultimi byte che cmq aveva già letto*/
   send(ns, buf2, nread, 0); }

                                      /* inizio streaming */
     set_flag(ns, 1);                            /* non-blocking-IO */
     a = 1;                                 /* non rewind */
     b = 1;                                 /* non exit */
     nread = 0;
     nwrite = 0;
     oldwrite = 0;
     do {
    if (nread == oldwrite) {                    /* se write cicli preced.ha scritto tutto */
     oldwrite = 0;
     nread = read(fd2, buf2, DIM_BUF2); }             /* faccio una nuova read */
    set_flag(ns, 1);                        /* non-blocking-IO */
    if ((recv(ns, rew, 10, 0)) > 0) {
      if ((a = strcmp(rew, "REWIND")) == 0) {          /* se ricevo rewind dal client riparto */
        close(fd2);                          /* rewind */
            if ((fd2 = open(nomefile, O_RDONLY)) < 0) {
              perror("open file");
              exit(1); }
            break; }
      if ((b = strcmp(rew, "EXIT")) == 0)              /* se ricevo EXIT dal client esco */
        break; }

                                      /* finisce di scriv. byte presenti in buf */
        set_flag(ns, 0);                          /* blocking-IO */
    if ((nwrite = send(ns, &buf2[oldwrite], nread, 0)) > 0) {
        scritti  += nwrite;
            oldwrite += nwrite; }
    printf("\nnread %d == %d oldwrite", nread, oldwrite);
      }
      while (nread > 0);
      if ((a == 1) || (b == 0)) {                     /* se arrivo a EOF senza ricevere rewind */
    close(fd2);                            /* o ricevo EXIT dal client */
    break; }                               /* esco dal ciclo */
     }
   printf("\nTrasmessi %d bytes\n", scritti);
   close(ns); } }
```