Sommario Esteso

In questa tesi abbiamo preso in esame il problema della carenza di garanzie sulla *qualità del servizio in reti wireless* basate sullo standard *802.11*. Tale problema è stato preso in considerazione da *802.11e*, un emendamento allo *standard 802.11* pubblicato nel novembre del 2005, il quale ha apportato una serie di modifiche al livello *MAC 802.11* con il fine di introdurre meccanismi per la gestione della qualità del servizio. *802.11e* definisce due nuovi protocolli per l'accesso al mezzo trasmissivo: *EDCA*, un protocollo a contesa, che si basa sulla differenziazione del traffico all'interno dei singoli elementi della rete *wireless*, ed *HCCA*, controllato da un coordinatore centrale, che fornisce garanzie di qualità del servizio tramite un meccanismo di *polling* che concede il diritto di trasmettere ai vari elementi che compongono la rete *wireless* sulla base delle decisioni prese da un apposito *scheduler* al suo interno. Molte ricerche sono state fatte su *EDCA*, per quanto riguarda la configurazione migliore dei parametri che regolano la differenziazione del traffico, e su *HCCA*, per quanto riguarda l'algoritmo di *scheduling* da adottare.

Non essendo fino ad ora disponibile *hardware* con il supporto *802.11e* ed essendo i costi dei *kit di sviluppo hardware* proibitivi, le ricerche si sono per lo più svolte con l'ausilio di simulatori *802.11* opportunamente aggiornati per supportare *EDCA* ed *HCCA*. Un approccio innovativo è invece quello seguito dal progetto *Host QS* per quanto riguarda *EDCA* e da questa tesi, con il progetto *Host HCF*, per quanto riguarda *HCCA*: si è provveduto ad introdurre meccanismi propri di *802.11e* in ambiente *Linux* al di sopra del livello *MAC 802.11 standard*, in modo da poter eseguire test in una rete *wireless* reale. Questo lavoro è stato reso possibile dall'esistenza del *driver* per dispositivi *wireless Host AP*, che consente di trasformare un personal computer dotato di una comune scheda *wireless* in *access point*: in questo modo si è potuti intervenire sul *driver* di rete delle stazioni e su quello dell'*access point*, per apportare le modifiche necessarie ad introdurre i meccanismi di differenziazione del traffico introdotti da *EDCA* ed il meccanismo di *polling* di *HCCA*.

Il lavoro di questa tesi ci ha portati quindi alla modifica di un *driver* di rete ed alla creazione di un componente *software* che interagisce con il cuore di un sistema *GNU/Linux*, realizzando un modulo per il *kernel Linux* volto

all'introduzione di *HCCA* all'interno di una rete *wireless 802.11b*.

Segue un breve riassunto degli argomenti trattati nei vari capitoli che compongono questa tesi.

1. Introduction

Nell'introduzione poniamo in evidenza la sempre maggior diffusione di tecnologie *wireless* per garantire connettività internet ovunque, focalizzando la nostra attenzione sulle reti *wireless* basate sullo standard *802.11* e sulle estensioni introdotte da *802.11e*.

L'impossibilità di fornire garanzie sulla qualità del servizio da parte dei meccanismi che regolano l'accesso al mezzo trasmissivo in *802.11* hanno portato al rilascio dell'emendamento *802.11e* che introduce *EDCA* ed *HCCA* al fine di garantire la qualità del servizio. L'inesistenza di *hardware* con il supporto ad *802.11e* e la volontà di testare le capacità di *HCCA* e dei relativi *scheduler* in una rete *wireless* reale e non simulata, ci hanno spinti ad implementare il modulo *Host HCF*, che, operando all'interno del *kernel Linux* ed interagendo con il *driver wireless Host AP*, ci consente di introdurre e testare alcuni dei meccanismi previsti da *HCCA* in una rete *802.11b*.

2. Linux Kernel Development

In questo capitolo ci addentriamo nelle problematiche legate allo sviluppo di codice per il *kernel Linux*.

Sviluppare codice in questo ambiente richiede di affrontare problematiche e limitazioni non presenti nell'ambito dello sviluppo di "normali" applicazioni utente. Prima di poter affrontare lo sviluppo di un modulo per il *kernel* della complessità di *Host HCF*, è necessario conoscere a fondo concetti fondamentali come la differenza tra *kernel-space* ed *user-space* e i vari *contesti* in cui il codice del *kernel* può essere eseguito (*process* ed *interrupt context*). È inoltre richiesta uno studio approfondito dei diversi meccanismi forniti dal *kernel* per eseguire il codice secondo varie modalità ed in vari contesti e per ritardare l'esecuzione di codice tramite *timer*. La conoscenza di questi argomenti è fondamentale e consente inoltre di comprendere come risolvere al meglio problematiche di vitale importanza come l'*accesso concorrenziale a strutture dati condivise*, problema esasperato dall'elevata concorrenzialità che si riscontra operando a livello *kernel*, dove diverse porzioni di codice possono essere eseguite nello stesso momento in diversi contesti di esecuzione (ognuno con i sui limiti e vantaggi rispetto agli altri) ed avere accesso ad un insieme condiviso di dati. Le difficoltà riscontrate nello sviluppo del codice si riflettono anche nella fase di *debugging*, la quale richiede l'utilizzo di tecniche diverse rispetto a quelle impiegate con codice appartenente ad applicazioni standard.

3. QoS and 802.11e

In questo capitolo descriviamo il concetto di qualità del servizio (*QoS*) e le

innovazioni introdotte dall'emendamento *802.11e* nel livello *MAC 802.11*.

I requisiti di qualità del servizio sono specificati da una serie di parametri, come il *ritardo end-to-end* e la *velocità di trasmissione minima* relativi ad una comunicazione, ad esempio. L'introduzione della qualità del servizio all'interno delle diverse architetture di rete ha portato alla definizione di due diverse architetture, *Differentiated Services* e *Integrated Services*, la prima basata sul raggruppamento del traffico in diverse categorie con determinati requisiti di *QoS* e la seconda basata su un meccanismo di segnalazione dei requisiti di *QoS* che consente di offrire determinate garanzie riservando le risorse necessarie per ogni flusso. Entrambi gli approcci sono stati presi in considerazione dall'emendamento *802.11e*, il quale ha introdotto due nuove modalità di accesso al mezzo trasmissivo, *EDCA* (ispirata a *Differentiated Services*) ed *HCCA* (ispirato ad *Integrated Services*), con lo scopo di introdurre il concetto di *QoS* sia nella fase di accesso a contesa (*EDCA*) che nella fase di accesso controllato (*HCCA*). I meccanismi di accesso al canale di *802.11*, *DCF* e *PCF*, non sono in grado di offrire garanzie di *QoS*: *DCF* è un protocollo di accesso a contesa che non distingue tra i diversi tipi di traffico e che può fornire solamente un servizio di tipo *best effort*; *PCF*, nonostante fosse stato sviluppato con l'intento di offrire garanzie sulla qualità del servizio, ha mancato il suo obiettivo in seguito a problemi nel design e non è supportato dalla maggior parte dei dispositivi *wireless 802.11*. Una volta chiarito il concetto di *QoS* e descritti *PCF* e *DCF* è possibile affrontare la descrizione di *EDCA* ed *HCCA*, con una maggiore attenzione sul secondo che è al centro del lavoro di questa tesi: *EDCA* introduce un sistema di code all'interno delle varie stazioni, ognuna contenente traffico appartenente ad una determinata categoria, le quali si contendono tra loro *a livello virtuale* il diritto di tentare la trasmissione sul canale *wireless* (dove l'accesso è regolato da *DCF* e dove avviane la contesa *reale*); *HCCA* utilizza invece un sistema di *polling*, simile a quello di *PCF*, ma più evoluto, che mira a risolvere i problemi di *PCF* tramite l'introduzione di *specifiche di traffico* (per definire in modo dettagliato i requisiti dei flussi di traffico), di un *meccanismo di segnalazione* tra stazioni ed *access point* (per lo scambio di informazioni relative alla gestione della qualità del servizio), di procedure volte al *controllo dell'ammissione* di nuovi flussi (per evitare di accettare nuovi flussi di dati quando non ci sono risorse sufficienti a garantire loro la qualità del servizio richiesta e a preservare quella garantita ai flussi precedentemente ammessi) e tramite la possibilità di creare uno *scheduler* più flessibile do quello previsto in *PCF*. Per quanto riguarda lo *scheduler HCCA*, *802.11e* non impone l'utilizzo di un algoritmo specifico, lasciando ai produttori di hardware la libertà di implementarlo come meglio credono. È proprio questa "libertà" fornita da *802.11e* che rende interessante la possibilità di testare vari meccanismi di *scheduling* tramite simulatori o tramite soluzioni come quella proposta in questa tesi.

4. Host HCF

In questo capitolo si presenta l'architettura generale del modulo *Host HCF* da noi sviluppato e si descrivono i dettagli relativi alla sua implementazione.

La realizzazione di questo progetto è stata possibile grazie all'esistenza del driver per dispositivi *wireless Host AP*, che ci ha consentito di realizzare una rete *802.11b* compresa di *access point* utilizzando solamente dei comuni *personal computer* e delle schede *wireless* con esso compatibili, e grazie alla sua modalità di sviluppo di tipo *Open Source*: *Host AP* abbraccia inoltre la filosofia del *Free Software*, essendo rilasciato sotto licenza *GNU GPL*. Ciò ha reso quindi possibile modificare il *driver* in modo da inserire il nostro modulo *Host HCF* al di sopra del livello *MAC 802.11*. *Host HCF* include alcuni dei meccanismi introdotti da *HCCA*, escludendo la negoziazione dei parametri di *QoS* tramite specifiche di traffico, la cui mancanza è stata sopperita attraverso un meccanismo che ci consente di abbinare i flussi di dati a diverse categorie di traffico sfruttando uno dei campi dell'*header IP*, denominato *Type of Service*. Le attività che devono essere svolte da *Host HCF* sono molteplici ed includono: accodamento, sulla base della categoria di appartenenza, del traffico in uscita dati diretto al *driver Host AP* , all'interno di code presenti in ogni stazione e all'interno dell'*access point* (il quale gestisce un insieme di code per ogni stazione ad esso associata); introduzione di alcune nuove sequenze di *frame* e del meccanismo di segnalazione per il controllo della *QoS*, tramite un protocollo da noi sviluppato e denominato *ACCA*, operante a livello *LLC* (la cui implementazione ha richiesto inoltre l'intercettazione del traffico in entrata, incapsulato all'interno del nostro protocollo); gestione all'interno dell'*access point* di strutture dati relative alle stazioni ad esso associate (che possono essere aggiunte e rimosse durante l'esecuzione in seguito all'associazione o disassociazione di stazioni); infine, l'ultima importante attività che deve essere svolta da *Host HCF* riguarda lo *scheduling* dei diversi flussi di traffico, che viene svolta tramite uno *scheduler* di tipo *open-loop*, ossia che non considera il *feedback* fornito dalle stazioni all'*access point*, volto ad offrire garanzie di *QoS* a flussi di *traffico vocale interattivo VoIP*. L'intera implementazione si è dovuta scontrare con le difficoltà relative alla programmazione di codice per il *kernel Linux*, come l'accesso concorrenziale ai dati condivisi, e con le limitazioni imposte da un'implementazione *software* rispetto ad una *hardware*, tra cui l'impossibilità di trattare intervalli di tempo al di sotto del millisecondo, che portano ad un inevitabile calo di performance ma che ci hanno consentito comunque di testare un meccanismo di *scheduling HCCA* e di preparare l'infrastruttura necessaria per implementare e testare nuovi *scheduler* che possono essere aggiunti in una fase successiva.

5. Tests

In questo capitolo, presentiamo i risultati ottenuti sperimentando il modulo *Host HCF* in una rete *wireless 802.11b*.

Tali test ci consentono di giustificare alcune delle scelte implementative effettuate all'interno di *Host HCF*, riguardanti per lo più lo *scheduler*, e di mostrare come si comporta il nostro modulo in varie situazioni, variando ad esempio la dimensione dei pacchetti e il carico a cui la rete è sottoposta. Infine, è stato possibile testare la capacità dello *scheduler* sviluppato di proteggere le chiamate vocali interattive *VoIP* dal traffico di tipo *best effort* in situazioni che danneggerebbero irreparabilmente la qualità di tali chiamate all'interno di una rete *wireless 802.11* sotto il controllo di *DCF*.

6. Conclusions and Further Work

In questo capitolo riassumiamo in breve l'intero lavoro svolto in questa tesi, evidenziando come l'obiettivo che ci eravamo preposti, ossia riuscire a costruire l'infrastruttura necessaria per testare degli *scheduler HCCA* all'interno di una rete *wireless 802.11b*, è stato raggiunto, ottenendo anche dei risultati interessanti con una prima implementazione di *scheduler open-loop*. Infine, presentiamo alcune delle possibili direzioni in cui può procedere lo sviluppo di *Host HCF*, come il miglioramento del sistema di *locking* per l'accesso alle strutture dati condivise al fine di ridurre i ritardi interni e migliorare le performance, l'introduzione del meccanismo di negoziazione delle specifiche di traffico e la realizzazione di uno *scheduler* di tipo *closed-loop*, ossia che prende in considerazione il *feedback* fornito dalle stazioni per adattare il proprio comportamento in base alla situazione delle stazioni stesse.

A. Software and Development Environment

Questo appendice raccoglie una lista degli strumenti *software* e degli *ambienti di sviluppo* utilizzati per la stesura della tesi e per la stesura e la compilazione del codice relativo ad *Host HCF*.

Chapter 1

Introduction

Access to the Internet is becoming more and more important for a lot of jobs, for educational purposes, for entertainment and for a lot of other reasons related to the large amount of information that can be found in the global network, and the ability to put in contact places which are geographically far away, without too much effort. One of the challenges that are of major interest is providing everyone with Internet access, no matter where he is: schools, museums, coffee bars and airports are only some examples of places in which an Internet access may be required or at least desired by people. “Fast connectivity” is usually provided through *cabled networks* (*optical* and **DSL* for example), which are able to reach very high speeds, while “everywhere connectivity” is being realized through other technologies, which may involve *satellite communications*, *cellular networks* and *802.11 wireless networks*.

Satellites’ networks can be useful to provide connectivity in isolated places (like desert or war fields), without the need of building any infrastructure on the ground.

Cellular networks, such as *GPRS* or *UMTS*, can provide Internet access almost in any place where there are populated areas (the needed infrastructure is quite costly and it can not be deployed where there is not a sufficient amount of users) but, at present, high costs and low bit-rates prevented them from being used by the majority of people for this purpose.

Wireless networks based on the *IEEE 802.11 Standard* [1, 2] are useful to grant Internet access in places where you do not want or you can not build cabled networks, because of reasons related to their cost or to the difficulty of realizing them. They are low-cost and can reach high speeds (not as cabled networks however, which can be two orders of magnitude faster). A wireless *hot-spot*, i.e. a place where wireless connectivity is available, can be provided quite easily once you have granted Internet connectivity to the *Access Point*, which acts as a *bridge* between the wireless devices and the *Internet gateway*.

Local Area Networks (LANs) based on the *IEEE 802.11 Standard* are *cellular networks*: each cell, called *Basic Service Set (BSS)*, is served by an *Access Point (AP)* to which one or more *Stations* can be connected. Small wireless hot-spots are made of a single BSS, but bigger *Wireless Local Area Networks (WLANs)* as the ones that cover entire campuses (as the Faculty of Science of the University of Trento, for example) are usually made of multiple BSSs, connected one to each other through a sort of *backbone* called *Distribution Service (DS)*, which is usually *Ethernet-based*. All the BSSs and the DS are seen as a unique 802.11 network which is called *Extended Service Set*. Along with this AP-based architecture, called *Infrastructured Network*, 802.11 defines also an *Ad Hoc Network* architecture, which can be used to build a wireless network among wireless devices without an AP or other infrastructures and that allows to establish *peer-to-peer communications* among the different devices.

The 802.11 standard defines two distinct mechanisms that can be used in a BSS to regulate the access to the wireless medium, the *Distributed Coordination Function (DCF)* and the *Point Coordination Function (PCF)*. DCF is a *contention-based channel access mechanism*: each station must contend for channel access with the other stations each time it needs to send data on the wireless medium. This mechanism is absolutely not able to grant *Quality of Service (QoS)* and can provide only a *best effort* service to the stations. PFC, instead, was developed with the purpose of granting QoS to some of the stations of the BSS by introducing a *polling-based channel access mechanism* controlled by a *Point Coordinator (PC)* (collocated inside the AP) which grants to the different stations of the BSS the right to transmit. Due to some limitations in its design it failed its purpose and almost all of the 802.11 wireless devices support only the DCF mechanism.

The problem of introducing quality of service awareness inside the networks has been taken into consideration by the *Internet Engineering Task Force (IETF)* which defined two *QoS models*, the *Integrated Services (IntServ)* and the *Differentiated Services (DiffServ)*: the first one requires the introduction of *end-to-end signaling mechanisms* for the negotiation of the QoS requirements of the different traffic streams, while the second one aims to group the different traffic streams into some known *classes of traffic* which have particular QoS requirements. IntServ is more powerful than DiffServ because of its finer-grained traffic streams characterization, but it is also more expensive to introduce and less scalable due to the necessity to introduce the support to the signaling mechanism in each element of the network. In order to be able to meet the quality of service requirements of a communication, it is necessary that every network which is crossed by the data that belongs to the communication itself is *QoS-aware*.

As we previously said, 802.11 wireless networks lack the QoS support, whose introduction has been taken into consideration by the *802.11e amendment* [3] to the main standard, released in November 2005. It

added some functionalities to the *802.11 Medium Access Control (MAC) layer* and a new coordination function, the *Hybrid Coordination Function (HCF)*, which alternates between two new channel access mechanisms, the *Enhanced Distributed Channel Access (EDCA)* and the *HCF Controlled Channel Access (HCCA)*. EDCA is a *contention-based* protocol that is able to *differentiate traffic streams* and to *group* them into *traffic categories*, as in the DiffServ QoS model. HCCA is a *contention-free* protocol that grants QoS requirements to the single traffic streams through a polling mechanism: as in PCF, it requires a central entity, the *Hybrid Coordinator (HC)* that resides in the AP and that has the duty to grant channel access to the stations of the *QoS-aware BSS (QBSS)*. It aims to solve the problems that determined the failure of PCF and requires QoS signaling between stations and the access point for negotiating QoS parameters, being somewhat inspired to the IntServ QoS model (in this case the scalability and costs issues are not a concern, since QoS signaling is limited to the BSS).

802.11e compliant hardware devices are not yet on the market, and a lot of studies on the performance and the tuning of the parameters of EDCA and HCCA have been made on existing 802.11 simulators [4, 5, 6, 7] extended with these new channel access mechanisms. A different way of testing EDCA has been experimented by the *Host QS Project* [8] which introduced EDCA traffic differentiation mechanisms *over* the standard 802.11 MAC layer. They used the *Host AP* WLAN driver (which allows a normal computer with a standard wireless card with an *Intersil Prism 2/2.5/3 chipset* to become an 802.11 access point), to build a Linux-based BSS and then they modified the WLAN driver of the AP and the stations in order to redirect the flow of outgoing packets, directed to the wireless card for transmission, to a kernel module developed by them. Then, they used this module to manage the dispatching of the outgoing packets as it would be done under the rules of EDCA. Host QS allowed them to test different configurations of the EDCA parameters in a real wireless LAN.

We considered this approach interesting, since it is the only way to test 802.11e traffic differentiation mechanisms in a real setting, given that modifying the standard 802.11 MAC layer can be extremely difficult and can not be afforded by us due to the high costs of *hardware development kits*.

The objective of this thesis is to follow a similar approach to introduce the 802.11e controlled access mechanism, HCCA, over the 802.11 MAC layer (focusing on the 802.11b [9] amendment, which raises the maximum transmission rate of 802.11) through the realization of a Linux kernel module, called *Host HCF*.

In Chapter 2 we discuss *Linux Kernel Development*: we explain the limitations and issues related to kernel development and the main concepts that a kernel developer must know. Then, we describe some important topics such as *concurrency and locking* and *kernel debugging techniques*.

In Chapter 3 we get in touch with the concept of quality of service and with the 802.11e standard: we discuss what is QoS and what are the parameters that are used to define QoS requirements and we describe the DiffServ and IntServ QoS models. Then we discuss the 802.11 DCF and PCF channel access mechanisms, in order to put in evidence the enhancements of EDCA and HCCA and later we examine into the details of the *Hybrid Coordination Function (HCF)*: we describe how EDCA and HCCA work, discussing *traffic specifications*, *QoS information signaling* and the *connection admission control* procedures, which are fundamental for granting QoS. We conclude the chapter with a discussion on the 802.11e *Sample Scheduler*: the HCCA scheduler is the component that determines how the hybrid coordinator polls the stations of the QoS BSS and 802.11e do not impose the utilization of a particular scheduler. 802.11e just suggests a simple implementation, called Sample Scheduler. This makes the implementation of the scheduler one of the main issues on which the research on HCCA is focused.

In Chapter 4 we discuss our project, Host HCF: we describe the general architecture of the system and then we go on with the implementation details. We explain what we implemented of 802.11e and how we solved the issues related to kernel programming that we faced during the development. We conclude the chapter with the description of our HCCA scheduler, called *Basic Scheduler*, which has been developed with the purpose of granting QoS to *interactive voice streams*, to preserve them from being damaged by *best effort* traffic.

Then, in Chapter 5 we discuss the tests we made with Host HCF to justify some of the implementation choices we made, to describe the behavior of Host HCF in different contexts and to show that it is able to preserve the quality of VoIP calls even in presence of high-load best effort traffic streams.

Chapter 2

Linux Kernel Development

In this chapter we discuss the issues related to the development of Linux kernel code [10, 11]. This introduction is necessary to fully understand the problems we faced developing our project, which as already said consists of a kernel module.We focus our discussion on 2.6 kernel series, which is the one we worked with (more precisely, we used a 2.6.15 kernel).

2.1 Limitations and New Issues

Developing kernel code is different from the "usual" application development: you must deal with a totally different environment, which provides less functionalities, emphasizes some problems and introduces some new issues and restrictions. Here we briefly discuss some of the peculiarities of kernel development:

- **No libc**: Kernel does not have access to the *C Library*. This is due to the fact that the C Library is too big and inefficient to be used by the kernel, even if only a subset of the most used functionalities is taken out from it. This means that a lot of functionalities that we can use when programming user-level application are missing when working in the kernel (some of the `libc` functions, such as those related to string manipulation, have been implemented also in the kernel sources). One of the most used C functions that are missing is `printf`: its work is done by `printk`, which is used in a similar way, apart from the fact that the messages that it prints out have a particular *log level* that is used to give different importance to them. `printk` does not prints the messages directly on the active console or terminal, but it sends them to the *kernel log buffer*, a circular buffer: this buffer is then read by programs like `syslog`, which shows them to the user. The particularity of `printk` is that it has been implemented in such a way that it is possible to call it from every point in the kernel, even

in interrupt handlers (Section 2.3), which makes it really useful for debugging purposes (Section 2.8);

- **No memory protection**: Memory protection is active for user applications because the kernel itself forbids illegal accesses and takes care of what each application does. If an illegal access is caused by a user application, kernel catches it and sends a `SIGSEGV` signal which causes a feared *segmentation fault* error that terminates the application and prevents damaging memory content. Kernel lacks such a protection because there is no one else that looks after the kernel: if a memory violation happens, it results in a kernel *oops*, which is a major error that can halt the entire system (if the violation is generated by kernel code that runs in *process context* (Section 2.3) the system could also survive the oops, although probably in a not healthy state because some important data has been corrupted);

- **Non pageable memory**: Every byte of memory allocated by the kernel is never paged and consumes a byte of physical memory. This must be kept in mind if developing for machines or embedded systems with few available physical memory;

- **No easy use of floating point**: It is not possible to do floating point computation in the kernel, and usually it is not really needed, without doing complex things such as directly interacting with floating point registers;

- **Small fixed-size stack**: User processes can allocate high amount of data on the stack since it can grow dynamically in size, while this is not possible for the kernel. In fact, it has a small 2 pages stack (8 kb on *x86* architecture) and so kernel developers should avoid static allocation of high amounts of data or big structures;

- **Concurrency and race conditions**: This is one of the major concerns for a kernel developer. Concurrency issues are bigger than the ones a developer must face when programming a user application, even if multi-threaded: *symmetric multi-processing* (*SMP*), *kernel-preemption* and *asynchronous interrupt delivery* make protecting shared data an issue that requires a lot of care. We will discuss all this issues in the following sections.

2.2 Kernel-space and User-space

The kernel is the innermost component of the operating system and it is responsible for managing system resources, dealing with the hardware, sharing processor time among the different processes, and a lot of other

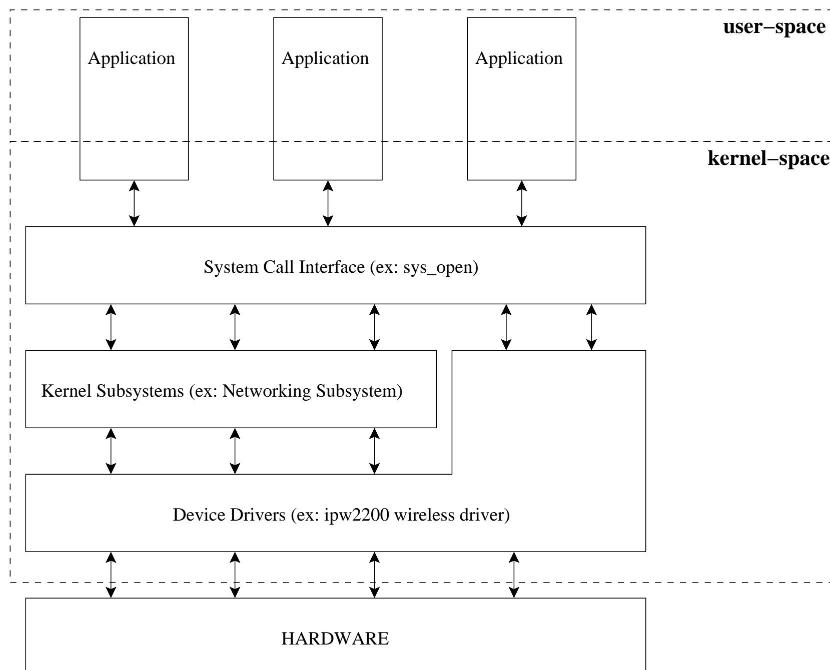

Figure 2.1: User-space, kernel-space and hardware interaction.

low-level activities like *Inter Process Communication*. The Linux Kernel, as the majority of the modern operating systems, runs in a privileged state in comparison with user applications: it has its own protected memory space and a full access to the hardware. This particular state is referred to as *kernel-space* while the user applications run in *user-space*: they can not directly access the hardware or system resources for example and they must often interact with the kernel which does the low level system operations for them. When the operating system is executing the kernel it is in kernel-space and it is executing in *kernel-mode*; during normal application execution, the system is in user-space and is executing in *user-mode*.

Each application, or better each *process* (in the rest of the document processes will be called also *tasks*, as it is the name that is used inside the kernel), does not live only in user-space: as it can be seen in Figure 2.1, processes can enter the kernel-space, thus interacting with the kernel and asking for doing some work, and they can do that through the *system call interface*. Usually applications call some library function, like `open`, which in turn invokes a system call (`sys_open`, for example): the system call invocation instructs the kernel to do some work for the process and the kernel is said to be *executing on behalf of the application* and in *process context*.