Universitat Autònoma de Barcelona

Projecte fi de Carrera

Enginyeria Tèc. de Telecomunicació

Implementació d'un demostrador wireless: Network coding i virtualització d'interfícies sense fils

Jordi Hidalgo Gómez

Director: Antoni Morell Pérez
Codirector: José López Vicario
Departament de Telecomunicació i Enginyeria de Sistemes

Escola Tècnica Superior d'Enginyeria (ETSE)
Universitat Autònoma de Barcelona (UAB)

Universitat Autònoma de Barcelona

El sotasignant, Antoni Morell Pérez, Professor de l'Escola Tècnica Superior d'Enginyeria (ETSE) de la Universitat Autònoma de Barcelona (UAB),

CERTIFICA:

Que el projecte presentat en aquesta memòria de Projecte Fi de Carrera ha estat realitzat sota la seva direcció per l'alumne Jordi Hidalgo Gómez.

I, perquè consti a tots els efectes, signa el present certificat.

Bellaterra, 22 de septembre de 2009.

Signatura: Antoni Morell Pérez

Índex

Índex de figures

Índex de taules

Capítol 1

Introducció

En els darrers anys, la popularitat de les xarxes sense fils ha crescut considerablament. Aquestes són presents a la majoria de domicilis particulars i en multitud d'espais públics com ara biblioteques, escoles o fins i tot parcs. Tots els ordinadors portàtils d'avui en dia ja incorporen els mitjans necessaris per a connectar-se a la xarxa, i són molts els telèfons mòbils actuals que també gaudeixen d'aquest servei.

L'increment de l'us d'aquesta tecnologia és degut a la comoditat que suposa el fet de no haver-se de connectar físicament a la xarxa. Un altre factor important per el qual cada vegada les xarxes wireless són més populars, és la millora en qualitat. Aquestes millores son possibles gràcies als estudis de recerca en camps com les xarxes de comunicació o de la teoria de l'informació. Tècniques com el network coding, que encara no s'implementa a les xarxes actuals, o la virtualització, que si està en pràctica des de fa temps, són exemples del fruit d'aquests estudis, i que fan possible millorar i avançar en els respectius camps d'investigació.

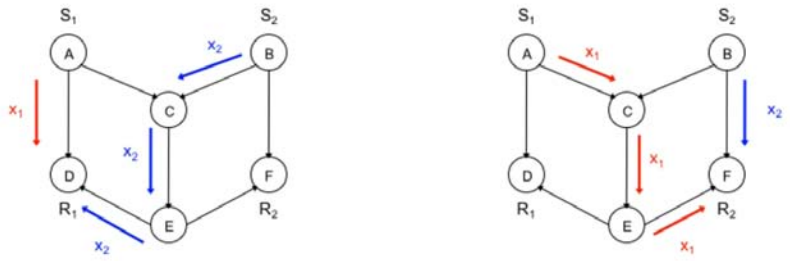
Els objectius principals d'aquest projecte són l'estudi i implementació d'un sistema capaç de posar en pràctica els trets característics de network coding. A partir d'aquí es decantarà la línia de treball i ens centrarem en la virtualització a les xarxes sense fils.

1.1 Network Coding i virtualització

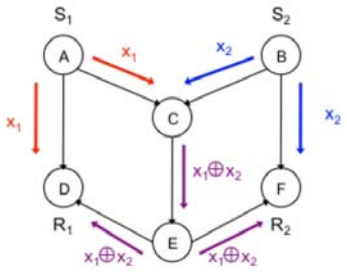
El concepte de ***Network coding*** sorgeix a partir d'un seguit d'observacions realitzades a uns estudis [8], on es mostrava que el thoughput d'una xarxa podia millorar considerablement, permetent als nodes de la xarxa combinar els paquets entrants en comptes de retransmetre'ls directament. Un exemple típic del funcionament d'aquesta tècnica, el podem veure a la figura 1.1.

A la figura 1.1 es representen les dues tècniques, convencional i Network Coding. L'objectiu és transmetre dos bits d'informació($x1$ i $x2$) a dos nodes(R1 i R2). Per dur

a terme aquesta tasca, les transmissions es divideixen en fraccions de temps, i només es pot transmetre un bit per cada fracció de temps.



a)Mètode convencional



a)Network Coding

Figura 1.1: *Mètode convencional vs. Network coding*

Com es veu a la figura 1.1.a al mètode convencional, son necessaris dos processos de transmissió per a que els nodes R1 i R2, rebin els dos bits correctament. En canvi, amb network coding, com que es fan transmissions simultàniament amb un sol procés n'hi ha prou per transmetre els dos bits *x1* i *x2* als nodes R1 i R2.

L'avantatge de network coding esdevé gràcies a la combinació dels dos bits, de manera que podrem transmetre'ls com si només fos un, i un cop rebut, descodificar el paquet.

L'implementació de network coding no implica l'ús de la virtualització, però si que és una molt bona opció per a treballar amb diferents usuaris transmetent informació simultàniament. És per aquesta raó que part de l'estudi del projecte es centra en l'estudi de la virtualització.

La virtualització és una tècnica que s'usa en molts àmbits. El seu objectiu principal és el de compartir els recursos d'un sistema per a que diferents usuaris virtuals puguin utilitzar-los d'una manera més eficient. En aquest projecte, ens centrarem en la virtualització de interfícies sense fils. Així doncs, del que es tractarà, és de crear diferents usuaris virtuals transmetent i rebent dades en una sola interfície, independentment i a temps real.

1.2 Objectius

Els principals objectius sorgits del desenvolupament del demostrador wireless mitjançant network coding i posant èmfasi en la virtualització, són els següents:

- Conèixer les principals eines i utilitats que ens ofereix **LINUX** i **MadWifi** per a la creació i manipulació de xarxes Wireless.
- Estudiar les diferents tècniques de transmissió de dades i comunicació entre equips mitjançant l'ús de **sockets**.
- Aprendre i dominar els aspectes bàsics de la programació **concurrent**.
- Estudiar els models bàsics de la virtualització a les xarxes wireless i implementar un programa multi-procés per a simular la virtualització.
- Realitzar les **simulacions, proves i càlculs** necessaris per veure els avantatges i inconvenients que puguin sorgir de la virtualització.

1.3 Estructura de la memòria

Podríem separar en dues parts les tasques dutes a terme, una primera part en la que es fa recerca i s'estudien les eines necessàries per al desenvolupament del programa final, i evidentment una segona part on es dissenya i s'implementa el programa on s'aplicaran les tècniques de virtualització i network coding.

Primerament es mostrarà l'estructura del demostrador desenvolupat i les seves corresponents parts al capítol 2. Als capítols 3 i 4, hi trobarem els passos necessaris per a la correcta configuració d'una tarja Atheros amb el driver MadWifi i les diferents comandes LINUX necessàries. Als capítols 5 i 6, es farà una recerca dels conceptes bàsics sobre programació amb sockets i amb threads respectivament. Al capítol 7, s'hi tractarà la virtualització, tan teòricament, com una implementació pràctica. Finalment, als capítols 8 i 9, podrem veure els resultats i conclusions obtingudes.

Capítol 2

Arquitectura del demostrador wireless

2.1 Estructura de la xarxa

L'estructura de la xarxa amb la que es volia treballar inicialment es la mostrada a la figura 2.1. Aquesta xarxa, disposa de tres nodes, amb els quals és suficient per a la implementació de network coding. Igual que als exemples vistos a la secció 1.1 del capítol 1, l'objectiu és transmetre dos missatges a dos nodes diferents. A la nostra xarxa l'intenció era transmetre dos missatges des del node 1 i el 2, simultàniament al node 3, on aquest fa la codificació i es retransmet als nodes 1 i 2 respectivament.

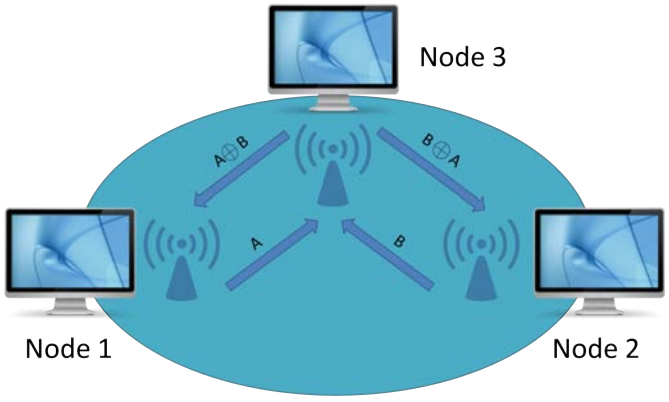


Figura 2.1: *Estructura de la xarxa inicial*

Com ja hem dit, a partir de les necessitats que sorgeixen de la tècnica network coding, es va decidir per aplicar la virtualització al nostre sistema. Així doncs, partint de la xarxa inicial de la figura 2.1 es va optar en prescindir d'un ordinador i realitzar les mateixes

operacions amb només dos nodes. Això implica que a l'ordinador on s'implementaran els nodes virtuals, tindrem dos usuaris treballant independentment que simularan l'existència de dos interfícies físiques. A la figura 2.2 podem veure un exemple d'una xarxa realitzant les mateixes operacions que a la xarxa inicial amb només dos ordinadors.

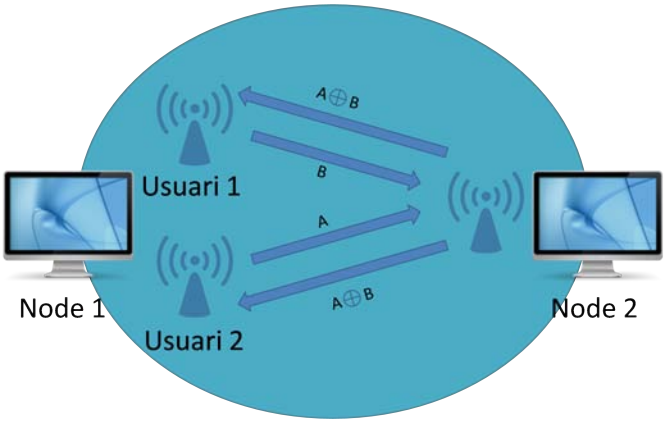


Figura 2.2: *Xarxa amb nodes virtuals*

En aquesta nova distribució de la xarxa, el node que retransmet els paquets farà la mateixa funció. A l'altre node hi funcionaran els dos usuaris que transmetran i rebran els missatges independentment. A més cadascun, guardarà les dades rebudes a la seva part de memòria.

2.2 Format del paquet

Un cop vista l'estructura que tindrà la nostra xarxa, haurem de decidir quin tipus de dades transmetem i com les transmetem. Per a simular el tràfic de dades d'una xarxa sense fils normal, el que es farà és generar el nostre propi tràfic de dades. El tràfic generat, consistirà en una sèrie de paquets, que seran diferents per als dos usuaris. Per diferenciar-los, a més dels paràmetres de control, aquests paquets hauran de tenir les corresponents capçaleres. Però abans de definir el format del paquet haurem de determinar quin tamany tindrà.

2.2.1 Tamany del paquet

Per evitar que el nostre paquet pateixi modificacions o pugui ser fragmentat, a l'hora de determinar el tamany del paquet, haurem de tenir en compte el format de les capes del model *OSI*[6] que es vegin involucrades. Des d'un principi es va decidir per treballar a nivell de la capa de transport ja que a nivells inferiors com la capa d'enllaç, no existeixen tantes eines per a la transmissió de dades com a la de transport.

Per determinar el tamany del paquet que volem implementar, primer haurem de veure quines són les capes que s'hi veuen involucrades. En el nostre cas, com que usarem el protocol UDP per a realitzar les transmissions, haurem de tenir en compte les capes que quedin per sota de la de transport. Un cop sapiguem el tamany de les capçaleres de cada capa, haurem d'anar restant-les partint des de la inferior fins a la de transport. Així el nostre paquet podrà ser encapsulat dins de les dades de la capa de transport.

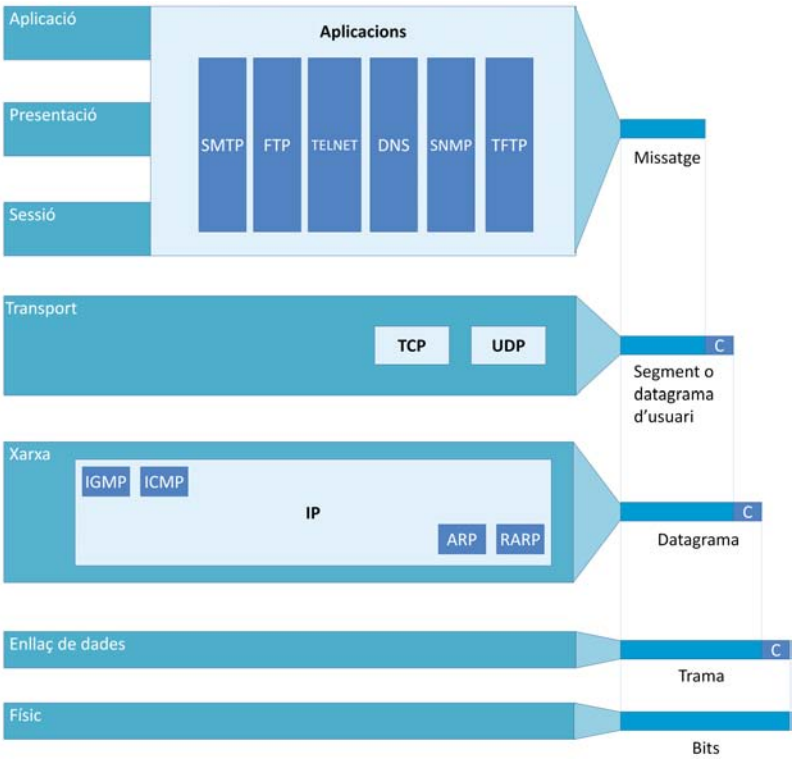


Figura 2.3: *Model TCP/IP d'OSI*

Si mirem la figura 2.3 podem veure l'estructura del model *TCP/IP* d'*OSI*. Com es pot observar, la capa d'enllaç està per sota de totes, deixant de banda la capa física. Un cop sapiguem el tamany de la capçalera MAC, podrem anar restant les capçaleres de les capes superiors.

Per trobar el format de trama MAC adequat, haurem de buscar el corresponent a una xarxa Wireless amb un estàndard determinat. En el nostre cas, estarem treballant amb l'estàndard *IEEE 802.11*. El format de la trama MAC *IEEE 802.11* serà el següent:

2	2	6	6	6	2	6	0-2312	4
Control de trama	Duració/ID connexió	Adreça	Adreça	Adreça	Control de seqüència	Adreça	Dades	CRC

Figura 2.4: *Format de la trama MAC IEEE 802.11*

El format es el mateix per a *IEEE 802.11b* que per al *IEEE 802.11g*, que són els utilitzats a les simulacions. Com podem veure a la figura 2.4 el tamany total de la trama serà de:

$$\textbf{\textit{Capçalera MAC}} + \textbf{\textit{Dades MAC}} = 2346 \ \textbf{\textit{bytes}}$$

Ara pujarem un nivell i ens trobarem la capa de xarxa o IP. El format del datagrama IP és fixa per a totes les xarxes, així que agafarem el format de la figura 2.5.

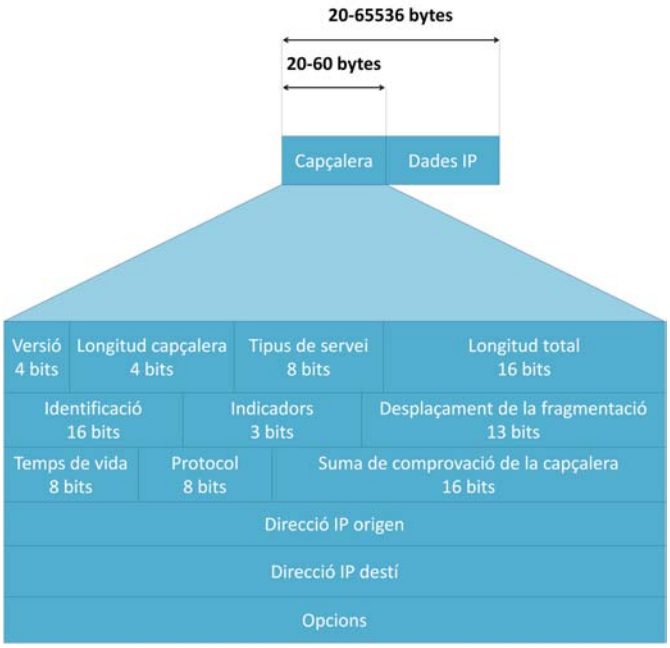


Figura 2.5: *Format del datagrama IP*

El datagrama IP te camps que són variables, per tenir el màxim de marge, sempre agafarem el pitjor cas. En aquest cas les capçaleres poden tenir entre 20 i 60 bytes, i les dades entre 20 i 65476 bytes.

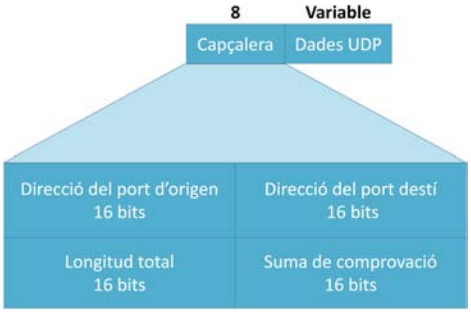


Figura 2.6: *Format del datagrama UDP*

Finalment arribem a la capa de transport. Tindrem segment o datagrama d'usuari depenent del protocol que utilitzem. Com ja hem dit, utilitzarem el protocol UDP. El format del datagrama d'usuari es el següent:

Com podem veure a la figura 2.6, el datagrama UDP té una capçalera fixa de 8 bytes i el camp de dades variable.

Vistos els formats de les capçaleres de les capes necessàries, podem començar calcular el tamany final del paquet. El tamany de la capa MAC, al ser el de abaix de tot, serà el que ens limitarà els altres. Com hem vist el tamany total de la trama MAC és de 2346 bytes, però ara només ens interessen els 2312 bytes de dades, on anirà encapsulat el datagrama IP.

$$DatagramaIp <= DadesMAC = 2312bytes$$

$$DatagramaIP = CapcaleraIP + DadesIP = 2312bytes$$

$$DadesIP = 2312 - 60 = 2252bytes$$

$$DatagramaUDP <= DadesIP = 2252bytes$$

$$DatagramaUDP = CapcaleraUDP + DadesUDP = 2252bytes$$

$$DadesUDP = 2252 - 8 = 2244bytes$$

Després de restar les capçaleres UDP i IP a la trama MAC, obtenim que les dades UDP podran ser com a molt de 2244 bytes. El nostre paquet haurà d'anar encapsulat dintre d'aquestes dades, és a dir, el tamany del nostre paquet serà de **2244 bytes**.

2.2.2 Format del paquet

Un cop trobat el tamany adequat del paquet, ens podem disposar a definir quin serà el seu format. El seu format, igual que les trames i els datagrames, tindrà una part de capçaleres, i una altre de dades.

Figura 2.7: *Format del paquet*

La part de capçaleres estarà formada per diferents camps que portaran informació de control d'un usuari en concret. Mentre que les dades serà l'espai destinat a guardar-hi el missatge, fitxer o informació que vulguem transmetre.

Els camps d'informació que portarà la capçalera, seran: **identificador d'usuari**, **Xor** i **Identificador de paquet**.

- **Identificador d'usuari.**
  El camp identificador d'usuari ens servirà per especificar quin usuari envia el paquet. Aquest camp ocuparà 1 byte de longitud, i com que en el nostre cas només treballarem amb 2 usuaris, només haurem de diferenciar entre l'usuari 1 i 2. Encara que en aquest cas no sigui necessari, amb aquest camp, podríem arribar a tenir 2^8 usuaris diferents .

  Per a identificar a l'usuari 1 utilitzarem la seqüència de bits 00000000, mentre que per a l'usuari 2 11111111.

- **Xor.**
  Aquest camp no serà molt important per a a la part de virtualització, ja que està destinat a especificar si el paquet està codificat o no en el demostrador wireless amb network coding. De totes maneres, com que pot ser útil per a implementacions futures, s'hi ha inclòs.

  Com que només tenim dues possibilitats, utilitzarem la mateixa codificació que al camp anterior. Si s'ha fet la suma XOR tindrem 11111111, sinó s'ha fet res 00000000.

- **Identificador de paquet.**
  L'identificador de paquet serà igual que el camp identificador d'usuari. Tindrà 1 byte de longitud, amb el qual podrem arribar a comptar fins a 2^8 paquets. Aquest camp també estava destinat per a l'us amb *Network Coding*, per fer correspondre els missatges d'un usuari i l'altre, i així poder descodificar-los sense cap problema.

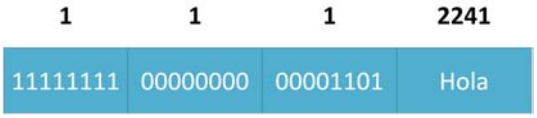


Figura 2.8: *Exemple de paquet*

A la figura 2.8 podem veure l'exemple d'un paquet. En aquest cas, anirà destinat a l'usuari 2, no estarà codificat, el seu número de seqüència serà el 13 i el missatge a transmetre serà la cadena hola.

Capítol 3

MadWifi

El fet que des de l'inici del projecte es decidís per utilitzar LINUX, ens va fer acotar una mica la recerca a l'hora d'escollir la tarja wireless LAN, ja que no totes les tarjes son compatibles amb el sistema operatiu o no tenen el mateix rendiment.

Així doncs, després de buscar per diferents llocs web de fabricants de tarjes i fòrums d'usuaris de LINUX, ens vam decantar per escollir la tarja amb chip Atheros. Els chips Atheros, són els que tenen més bon rendiment avui en dia, a més de ser els que millor treballen amb LINUX.

Un cop decidit i comprada la tarja, el següent pas va ser trobar quin eren els millors *drivers* per a la tarja. Aquest pas no ens va costar gaire, ja que a diversos projectes relacionats amb *Network Coding* que vam consultar, tots utilitzaven el *driver* MadWifi. A més a més, també és el més recomanat a molts fòrums d'usuaris amb tarjes Atheros.

3.1 MadWifi

El nom de MadWifi ve de les sigles ***M**ultiband **A**theros **D**river for **Wi**reless **Fi**delity*[1]. En altres paraules, és tracta d'un controlador del kernel de LINUX per a dispositius Wireless LAN basats en chips Atheros. És, com ja hem dit, el *driver* més estès en l'ús de tarjes Atheros.

3.1.1 Característiques

Modes d'operació

Els següents modes d'operació son compatibles:

- **STA**.
  Mode estació, infraestructura o *managed*. Aquest és el mode que ve sempre per

defecte. Només podrà connectar-se a un AP (*Access point*) i no es podrà connectar amb altres STA.

- **AP**.
  Mode *Access point* o *master*. Aquest es l'encarregat de crear una xarxa a la que els altres usuaris(STA) s'hi connectaran. Al contrari del STA, aquest no es podrà connectar a altres AP, només permet que s'hi connectin.

- **AD-HOC**.
  Mode AD-HOC o mode IBSS. Amb aquest mode, creem una xarxa *peer-to-peer* WLAN a la que s'hi poden connectar altres usuaris AD-HOC o STA sense necessitat de crear cap AP.

- **Monitor**.
  Mode monitor. El mode monitor és utilitzat bàsicament per alguns *sniffers* per a capturar trames d'informació i analitzar el tràfic de la xarxa.

- **WDS**.
  Sistema de distribució inalàmbrica. El mode WDS ens permet crear grans xarxes WLAN ja que el que fa es connectar entre si diferents AP com si només en fos un.

Xifratge

El controlador suporta els diferents tipus de xifratge:

- **WEP**
  Wired Equivalent Privacy, amb claus de 40/64 bits o 104/128 bits. Aquest és compatible amb els modes STA, AP i AD-HOC.

- **WPA**
  WiFi Protected Access, compatible als modes STA i AP.

- **WPA2/IEEE 802.11i**
  WiFi Protected Access 2, igual que el WPA, suportat per els modes STA i AP.

- **IEEE 802.1X**
  Port-based Network Access Control, aquest només és compatible amb el mode AP.

Multi-BSSID

EL *driver* MadWifi ens permet crear diferents AP en una mateixa tarja, els quals anomenem *Virutal Access Points*(VAP). Els VAPs es creen sobre el dispositiu base (normalment anomenat *wifi0*), representant diferents dispositius WLAN que poden ser usats en diferents modes cadascun. Per exemple, de la mateixa manera que ens connectem a un

AP amb un VAP, amb un altre podem crear un AP i que altres usuaris s'hi connectin. A més d'altres modes d'operació que son possibles.

Per manipular els VAPs, MadWifi posseeix una eina anomenada *wlanconfig* que s'usa per crear i destruir VAPs amb diferents modes d'operació.

3.1.2 Funcionament

Un cop instal·lat el driver, podem configurar els paràmetres del dispositiu amb les comandes habituals (*ifconfig, iwconfig*). Per exemple si volem veure informació de la nostra interfície:

```
$ iwconfig ath0
```

El resutltat serà, el següent, on podem veure els paràmetres de l'interfície:

```
$ ath0    IEEE 802.11  Nickname:""
          Access Point: Not-Associated
          Link Quality:2  Signal level:178  Noise level:166
          Rx invalid nwid:0  invalid crypt:0  invalid misc:0
```

Ara bé, si el que volem es treballar amb els VAPs, el que haurem de d'utilitzar es l'eina ja esmentada a l'apartat anterior, *wlanconfig*.

```
$ wlanconfig ath0 create wlandev wifi0 wlanmode managed
```

A l'exemple anterior, podem veure com es crea un VAP. Li especifiquem el nom del VAP, *ath0*, li diem `ath0` perquè es el primer, però podríem anar creant-ne diferents, `ath1,ath2,ath3`... Seguidament li especifiquem el nom del nostre dispositiu, com ja hem dit, amb atheros pren el nom de `wlan0`. I finalment li diem quin mode volem, en aquest cas *managed*, o STA.
Igualment podem crear els altres VAPs amb diferents modes d'operació, ara bé, per destruïr-ne ho farem de la següent manera:

```
$ wlanconfig ath0 destroy
```

Capítol 4

Comandes LINUX

Parlar de xarxes de computadors sempre implica parlar de **UNIX**. Per suposat, UNIX no és l'únic sistema operatiu amb capacitat per connectar-se a xarxes, però ho ha estat l'opció escollida durant molts anys i segurament ho seguira sent.

GNU/LINUX es un clon de UNIX, lliurement distribuït per a l'ús personal. És una plataforma molt adequada per accedir a Internet i té un gran suport de *hardware*. LINUX té *software* i protocols que altres sistemes no solen incloure. És potent i ràpid.

Per les raons aquí esmentades, s'ha treballat des del principi amb LINUX en aquest projecte. Abans de començar a implementar el programa de l'aplicació final, i després d'haver instal·lat correctament la tarja inal·làmbrica, vam començar la tasca de configurar correctament la tarja i la xarxa amb els paràmetres adequats, per tal de, posteriorment, poder treballar en millors condicions. És aquí on entren en joc un seguit d'instruccions i comandes que ens facilitaran l'administració de la xarxa i la configuració del nostre dispositiu.

4.1 Paràmetres de xarxa i Comandes

Per tal d'aconseguir un bon funcionament de la xarxa i de la taraja inal·làmbrica, haurem de configurar una sèrie de paràmetres que podran ser modificats amb les comandes que ens proporciona LINUX. Alguns dels més importants són:

- **Adreça MAC.**
  Podem establir l'adreça MAC de la nostra interfície.
- **Adreça IP.**
  Podem assignar una adreça IP determinada al nostre host, així com als diferents VAPs que tinguem creats.

- **Màscara IP.**
  Es pot també definir la màscara IP.

- **ESSID.**
  En el cas de que la nostra interfície estigui en mode AP podem definir el nom de la xarxa.

- **Nom de host.**
  Podem especificar el nom dels diferents hosts que es connecten a una xarxa local.

- **Freqüència/canal.**
  Hi ha la possibilitat de variar la freqüència i el canal amb els quals treballa la tarja.

- **Fragmentació.**
  És possible definir la mida amb que es fragmentaran els paquets.

- **Xifratge.**
  Podem afegir seguretat a la xarxa afegint-hi un xifrat dels diferents tipus vistos a l'apartat anterior.

- **Bit rate.**
  Es pot variar la velocitat amb que transmet el nostre dispositiu.

- **Potència.**
  També es possible canviar la potència amb que es transmet.

- **Modulació.**
  El dispositiu te varis tipus de modulació especificats a l'estàndard IEEE 802.11. Podem variar a 11a, 11b i 11g, per defecte IEEE 802.11b.

Així doncs, tots aquests paràmetres podran ser modificats mitjançant un seguit de comandes LINUX. Algunes de les més utilitzades són les següents:

- **ifconfig.**
  Configuració dels paràmetres de l'interfície local.

```
$ ifconfig ath0 essid jordi
```

  A l'exemple, el que fem, assignar-un nom de la xarxa local en la que estem. Li especificarem el nom de l'interfície(ath0) i el nom de la xarxa, en aquest cas "jordi". Aquesta operació només es podrà fer en el cas de que s'hagi creat l'interfície en mode AP.

- **iwconfig.**
  Configuració dels paràmetres de l'interfície inal·làmbrica local.

- **iwpriv.**
  Configuració de paràmetres opcionals de l'interfície inal·làmbrica local. Amb aquesta comanda podrem per exemple canviar la modulació amb que transmet el nostre dispositiu. Com ja hem dit hi ha varis modes de modulació, si volem per exemple passar al mode 11a, haurem de fer:

```
$ iwpriv ath0 mode 11a
```

- **wlanconfig.**
  Manipulació, creació i destrucció de VAPs(*driver* MadWifi).

- **route.**
  Mostra/manipula taules d'enroutament IP.

- **iwlist.**
  Aconsegueix informació detallada de l'interfície inal·làmbrica, tals com la potència, freqüència, velocitat, etc.

```
$ iwlist ath0 rate
eth0      12 available bit-rates :
          1 Mb/s
          2 Mb/s
          5.5 Mb/s
          6 Mb/s
          9 Mb/s
          11 Mb/s
          12 Mb/s
          18 Mb/s
          24 Mb/s
          36 Mb/s
          48 Mb/s
          54 Mb/s
          Current Bit Rate:54 Mb/s
```

  Veiem com, al cridar a la comanda *iwlist* amb l'opció *rate*, obtenim les diferents velocitats disponibles i l'actual.

- **iwevent.**
  Mostra esdeveniments i canvis provocats per el el nostre dispositiu inal·làmbric.

4.2 Exemple de creació d'una xarxa Wireless

Un cop vistos els paràmetres de xarxa bàsics, i les comandes necessàries per a manipular-los, ens podem disposar a fer un petit exemple de com crear una petita xarxa on dos equips estan connectats inal·làmbricament i poden intercanviar informació.
La xarxa estarà formada per un equip que farà de node (AP) on s'hi podrà connectar l'altre que estarà en mode estació (STA).

Primer de tot configurarem l'equip que farà de node. Començarem per crear-lo en mode *master*, seguidament li assignarem adreça IP i nom.

```
$ wlanconfig ath0 create wlandev wifi0 wlanmode master
$ ifconfig ath0 192.168.2.1
$ iwconfig ath0 essid jordi
```

En aquest cas es tractarà d'una xarxa sense xifratge, ja que per aquest exemple no es necessari. Seguidament procedirem a configurar l'altre equip. En aquest cas haurem de crear-lo en mode estació, assignar-li una adreça IP i finalment vincular-lo amb el *Access point*.

```
$ wlanconfig ath0 create wlandev wifi0 wlanmode managed
$ ifconfig ath0 192.168.2.2
$ iwconfig ath0 essid jordi
```

La comanda *iwconfig* te moltes opcions, però per a donar-li nom a una xarxa i per vincular-se a la xarxa es fa amb la mateixa opció *essid*. Per això en els dos equips, l'ultima instrucció que executem es la mateixa.

Finalment per comprovar que ens em connectat bé al AP, fem un ping des de l'equip estació.

```
$ ping 192.168.2.1
PING 192.168.2.1 (192.168.2.1) 56(84) bytes of data.
64 bytes from 192.168.2.1: icmp_seq=1 ttl=64 time=3.57 ms
64 bytes from 192.168.2.1: icmp_seq=2 ttl=64 time=3.55 ms
64 bytes from 192.168.2.1: icmp_seq=3 ttl=64 time=3.54 ms
```

Capítol 5

Sockets

Arribat al punt en que tenim dos equips correctament configurats de manera que podem connectar els dos equips entre sí, ens centrarem en la transmissió de dades.

Com que a l'aplicació final haurem de poder intercanviar dades, hem de buscar la manera de poder transmetre informació a nivell de la capa de transport. Podríem haver utilitzat programes específics per aquestes operacions, com per exemple el *netcat*, que es una utilitat que incorpora LINUX amb la qual podem transmetre paquets amb el protocol desitjat i especificant la fragmentació, però aquest tipus de programes no acaben de ser fiables en el sentit de que, moltes vegades no sabem realment com el programa està transmetent les dades.

Una forma d'aconseguir que dos programes es transmetin dades, basant-se en la família de protocols TCP/IP, és la programació de sockets. Un socket no es més que un "canal de comunicació"entre dos programes que funcionen sobre dos ordinadors diferents o fins i tot en el mateix.

Des del punt de vista de programació, un socket no és més que un "fitxer"que s'obre d'una manera especial. Un cop obert s'hi pot escriure i llegir dades amb les funcions habituals de `read()` i `write()` del llenguatje C.

Existeixen dos tipus de sockets, TCP i UDP. Els sockets TCP, es diu que estan orientats a connexió, ja que fins que no es connecten no es poden transmetre dades. Amb els sockets TCP garantitzem que totes les dades arriben correctament. Els sockets UDP son els no orientats a connexió, perquè en qualsevol moment poden intercanviar dades. Aquests no garantitzen que totes les dades que arribin siguin correctes.

5.1 Client/Servidor

A l'hora de comunicar dos programes[2], existeixen varies possibilitats per establir la connexió inicialment. Una d'elles es l'utilitzada aquí. Un dels programes deu estar

funcionant i en espera de que un altre vulgui connectar-s'hi. Mai farà el primer pas a la connexió. A aquest programa se'l coneix com a **servidor**.

L'altre programa s'anomena **client**. El seu nom es deu a que es el que sol·licita l'informació al servidor.

En resum, el servidor es el programa que roman passiu a l'espera de que algú sol·liciti una connexió seva, normalment per a demanar-li alguna dada. El client és el programa que sol·licita la connexió, normalment per a demanar dades al servidor.

Un exemple ben clar de l'estructura client/servidor és la forma en que accedim a Internet. Un servidor web espera passiu a que un navegador s'hi connecti per consultar el seu contingut.

5.2 Connexió

Per poder realitzar la connexió entre ambdós programes, són necessaries un seguit de paràmetres:

- **Direcció IP del servidor.**
  El client no necessita la direcció de cap dels equips que es connectaran entre sí. El client, en canvi, si que necessita saber la direcció IP del servidor per poder-s'hi connectar

- **Port.**
  Es pot donar perfectament la situació que en un mateix ordinador estiguin funcionant varis programes servidors a la vegada, en aquest cas, hauran de tenir ports diferents. Quan el client intenti connectar-s'hi haurà d'especificar el port determinat.

  Per això, cada port dins de l'ordinador ha de tenir un número únic que l'identifiqui. Aquests números són enters normals i van de l'1 al 65535. Els números baixos, des de 1 a 1023 estan reservats per a serveis habituals dels sistemes operatius (ftp,mail,ping). La resta estan a disposició del programador.

5.3 Servidor

Per a la implementació del programa servidor necessitarem un sèrie de funcions C que ens permetran la comunicació entre els dos programes. Com ja hem dit, hi han dos tipus de sockets, TCP i UDP, doncs cadascun tindrà un programa servidor diferent, encara que molt semblant.

5.3.1 TCP

Ela passos que haurem de seguir per implementar un programa servidor TCP són els següents:

1. **Apertura d'un socket**, mitjançant la funció `socket()`. Aquesta funció retorna un descriptor de fitxer normal, com pot retornar-lo la funció `open()`. La funció `socket()` ens retornarà i prepararà un descriptor de fitxer que el sistema posteriorment associarà a una connexió de xarxa.

```
int socket(int domain, int type, int protocol);
```

   El primer argument, *domain*, pot rebre dos valors, `AF_INET` o `AF_UNIX`. Posarem un o l'altre depenent de si el servidor i el client estan al mateix ordinador o no. El valor `AF_INET` valdrà per als dos casos, en canvi `AF_UNIX` només el podrem fer servir en el cas de que treballem al mateix equip. Normalment s'usa `AF_INET`, ja que funciona en els dos casos, però `AF_UNIX` implementa millor el programa en el cas de que només utilitzem un ordinador.

   El segon paràmetre indica si el socket està orientat a connexió(`SOCK_STREAM`) o no ho és(`SOCK_DGRAM`).

   El tercer argument és el protocol a utilitzar, normalment a 0.

2. **Avisar al sistema operatiu** de que hem obert un socket i volem que associï el nostre programa al socket. S'aconsegueix mitjançant la funció `bind()`. Es en aquesta crida a la funció `bind()` quan s'ha d'indicar el número de port.

```
int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

   `sockfd` és el descriptor[**?**] que hem creat al pas anterior, `*my_addr` és un punter a una estructura que conté informació amb la pròpia direcció IP i el port utilitzat. El tercer argument és la longitud de l'estructura `sockaddr`.

3. Avisar al sistema de que **comenci a atendre una connexió** de xarxa. S'aconsegueix mitjançant la funció `listen()`. A partir d'aquest moment el sistema operatiu anotarà la connexió de qualsevol client per passar-li al servidor quan aquest li demani. Si arriben clients més ràpid del que el servidor és capaç d'atendre, el sistema operatiu fa una cua i els anirà atenent ordenadament.

```
 int listen(int sockfd, int backlog);
```

   Com ja sabem, `sockfd` és el descriptor, i `backlog` es el número de connexions permeses a la cua d'entrada.

4. Demanar i **acceptar les connexions** de clients al sistema. Per això, cridarem a la funció `accept()`. Aquesta funció li indica al sistema que li doni al servidor el següent client de la cua. En cas de que no n'hi hagi s'espera fins que n'hi hagi algun.

```
int accept(int sockfd, void *addr, int *addrlen);
```

   A part del descriptor, la funció `accept()` rep l'argument `*addr`, un punter a una estructura on guardarem la informació de la connexió, i un altre punter amb la longitud d'aquesta estructura.

5. **Escriure i rebre dades** del client, mitjançant les funcions `write()` i `read()`, que son exactament les mateixes que utilitzem per escriure o llegir a un fitxer.

```
int write(int sockfd, const void *msg, int len);
```

   Aquesta funció rebrà com a arguments el descriptor, el missatge que volem enviar, que en aquest cas serà en forma de punter `*msg`, i la longitud del missatge.

   La funció `read()`, rebrà exactament els mateixos arguments.

6. **Tancar la comunicació** i el socket, mitjançant la funció `close()`, que és la mateixa que serveix per tancar un fitxer.

```
close(sockfd);
```

5.3.2 UDP

Per implementar el servidor UDP, haurem de seguir uns passos molt semblants al cas TCP. Com que UDP no està orientat a connexió, la diferència amb TCP és que no haurem d'esperar connexions i acceptar-les sinó que un cop haguem creat el socket i associat al sistema ja podrem intercanviar informació amb els clients i finalment tancar el socket.

En aquest programa, doncs, els passos a seguir seran els següents:

1. **Obrir** el socket amb la funció `socket()`
2. **Associar** el socket al sistema operatiu `bind()`
3. **Enviar** dades al client amb la funció `sendto()`.

```
int sendto(int sockfd, const void *msg, int len, unsigned
   int flags, const struct sockaddr *to, int tolen);
```

A la funció `sendto()`, els primers arguments són els mateixos que al `write()` o `read()` que a TCP: descriptor, missatge i longitud del missatge. El quart argument `flags`, es posa a 0, el cinquè `*to` es un punter a l'estructura que conté l'adreça IP i el port, finalment l'ultim es la longitud de l'estructura.

4. **Rebre** dades, mitjançant la funció `recvfrom()`, que rebrà els mateixos arguments que `sendto()`, la diferència és que en aquest cas en comptes de enviar-los, hi guardara l'informació rebuda.

```
int recvfrom(int sockfd, void *buf, int len, unsigned int
   flags, struct sockaddr *from, int *fromlen);
```

5. **Tancar** el socket amb la funció `close()`.

5.4 Client

L'estructura del programa del client, tant TCP com UDP, és molt semblant a l'estructura del servidor, per tant els passos seran pràcticament els mateixos.

5.4.1 TCP

Per a la implementació del client TCP seguirem els següents passos:

1. **Apertura** del socket, igualment que al servidor, mitjançant la funció `socket()`.
2. **Sol·licitar connexió** amb el servidor, amb la funció `connect()`. El programa quedarà bloquejat fins que el servidor accepti la connexió o bé si no hi ha servidor, retornarà un error. A aquesta crida, s'hi ha de facilitar la direcció IP del servidor i el número de port desitjat.

```
int connect(int sockfd, struct sockaddr *serv_addr, int
   addrlen);
```

   Li passarem, el descriptor, un punter a l'estructura que conté el port i adreça del servidor i la longitud de l'estructura.

3. **Escriure i rebre dades** del servidor mitjançant les funcions `write()` i `read()`.
4. **Tancar** la comunicació amb la funció `close()`.

5.4.2 UDP

Finalment, per a la implementació del client UDP, els passos necessaris son els mateixos que al servidor:

1. **Obrir** el socket amb la funció `socket()`
2. **Associar** el socket al sistema operatiu `bind()`
3. **Enviar** dades al client amb la funció `sendto()`.
4. **Rebre** dades, mitjançant la funció `recvfrom()`
5. **Tancar** el socket amb la funció `close()`.

Capítol 6

POSIX threads

Igual que en molts altres àmbits, a les xarxes de comunicacions és molt habitual que s'hagi de treballar amb diferents processos simultàniament per al bon funcionament d'un sistema determinat. Es per això que l'ús de threads està molt present en aquest àmbit i és una eina molt útil a l'hora de programar amb multi-processos.

Un thread[4], és un fil o un procés que s'executarà al nostre ordinador de manera simultània a altres processos que tinguem en un programa. Aquest procés podrà accedir a variables compartides per altres processos i manipular-les de manera segura sincronitzant-se amb els altres. Dins d'un programa, es crearà, s'executarà i finalment morirà com qualsevol altre procés.

La programació amb threads ha resultat ser pràcticament la base del programa que s'implementa en aquest projecte. Ja que sempre estem treballant amb diferents usuaris que s'intercanvien informació a temps real, no es pot tractar els processos de forma seqüencial. Ja que que perdríem molt de temps i rendiment de la xarxa. De la mateixa manera que un usuari mitjançant sockets espera per un port les diferents peticions d'altres usuaris, al mateix temps també en va enviant. Només amb aquest exemple podem veure ja que a un sistema amb aquesta implementació tindrem un millor rendiment que en un sistema seqüencial.

6.1 Conceptes bàsics de la programació concurrent

6.1.1 Definicions

- **Asincronisme** significa que les activitats passen independentment al temps, és a dir, que segueixen una certa ordre determinada. Per exemple imaginem que un procés vagi incrementant una variable i que en un cert moment la volem consultar. Si volguéssim que aquesta variable no deixés d'incrementar en el moment en que la consultem, haurem de tenir dos processos asíncrons.

- Ens referim a **concurrència** quan volem executar més d'un procés simultàniament en el mateix interval de temps. La concurrència descriu el comportament dels threads o múltiples processos en un ordinador. Les operacions concurrents poden executar-se arbitràriament: no necessitaran que una acabi per que comenci l'altre. Un programa concurrent no té perquè executar totes els seus processos simultàniament, encara que permet que executem funcions mentre les altres es van executant independentment.
- El **paral·lelisme** descriu processos concurrents que procedeixen simultàniament. Es refereix a coses que van en la mateixa direcció però independentment.

6.1.2 Funcions de control de concurrència

Un sistema concurrent haurà de proporcionar una sèrie de funcions que necessitarem per controlar com els nostres processos s'executen.

1. El context d'execució és l'estat d'una entitat concurrent. El sistema concurrent ha des saber l'estat dels processos en cada moment, com per exemple, quan hem d'esperar a un procés per començar un altre.
2. La planificació (scheduling) determinarà quin context executar en un moment determinat, i canviarà a altres quan sigui necessari.
3. La sincronització proporciona la coordinació entre els diferents processos a l'hora d'utilitzar dades compartides.

6.1.3 Avantatges i inconvenients de l'ús de threads

Avantatges	**Inconvenients**
Explotació del paral·lelisme	Overhead per creació de threads
Explotació de la concurrència	Sincronització:mes bloquejos al haver-hi mes threads
Model de programa modular	Col·lisions a l'accés a memòria

Taula 6.1: *Avantatges i inconvenients de l'us de threads*

6.1.4 POSIX Threads

Linux utilitza un estàndard POSIX per a l'us de threads que anomenem POSIX threads o Pthreads. Això vol dir que l'API (aplication progamming interfaces) està especificat a l'estàndard POSIX 1003.1c-1995. L'estàndard va ser aprovat per la IEEE al juny del 1995. POSIX son les sigles de "Portable Operating System Interface for Unix".

6.2 Threads

6.2.1 Creant i usant threads

Dintre del nostre programa un thread està representat per un identificador, en aquest cas del tipus `pthread_t`. Per crear un thread, haurem de declarar una variable del tipus `pthread_t` en algun lloc del programa.

```
pthread_t thread;
```

Un thread comença per cridar alguna funció que nosaltres definim. Per exemple definim la funció:

```
void *proces_thread (void *arg)
```

Aquesta funció esperarà un argument del tipus `void *`, i hauria de retornar un valor del mateix tipus. A l'hora de crear el thread haurem de fer servir la funció `pthread_create()`. Per al nostre cas seria:

```
pthread_create(&thread, NULL,&proces_thread, NULL);
```

A aquesta funció li haurem de passar quatre arguments, dels quals, el segon i el quart són opcionals i en podem prescindir, així que li passarem el valor `NULL`. El valor del primer argument serà l'adreça de la variable on hem guardat el thread, i el tercer serà l'adreça de la funció que es cridarà al crear el thread. Finalment, un cop s'hagi executat la funció `pthread_create()` retornarà un valor de tipus int. Aquest serà 0 si el thread s'ha creat sense cap problema o un valor diferent de 0 en cas contrari.

Quan executem un programa en C, comença en una funció especial anomenada main. En un programa amb threads, aquesta funció es anomenada "thread inicial" o "thread principal". Al thread principal podem fer qualsevol cosa que es pugui fer als demés threads, des de determinar l'identificador cridant a `pthread_self` com acabar cridant a la funció `pthread_exit`.

El thread inicial és especial perquè Pthreads conserva el comportament UNIX dels processos, amb el qual, el procés principal acaba sense esperar a que els altres finalitzin. Quan el procés principal acaba, els altres, sigui l'estat que sigui en el que es trobin, simplement s'evaporen, no ens hem de preocupar per ells.

Un cop tenim els threads creats i en funcionament, tenim dos opcions: esperar que els threads acabin, en el cas de que ens interessi obtenir algun resultat, o simplement dir-li a la llibreria phtreads que quan acabi l'execució de la funció del thread elimini totes les seves dades internes. Per això, disposem de dues funcions més de la llibreria: `pthread_join` i `pthread_detach`.

Si cridem a la funció `pthread_detach` el que estem fent és dir-li al sistema que pot utilitzar els recursos que se li havien assignat al thread. En la creació del thread es pot especificar el detach amb un atribut, que no es vol tenir control sobre el thread.

```
pthread_detach(thread);
```

Quan cridem a `pthread_join` el que farem és bloquejar el thread en el que es fa la crida de la funció fins que acabi el thread que li passem com a argument. Al segon argument li passarem `NULL`, ja que es opcional.

```
pthread_join(thread,NULL);
```

En el cas de l'exemple anterior, el thread on cridem a la funció `pthread_join()` quedarà bloquejat fins que no acabi el procés `thread`.

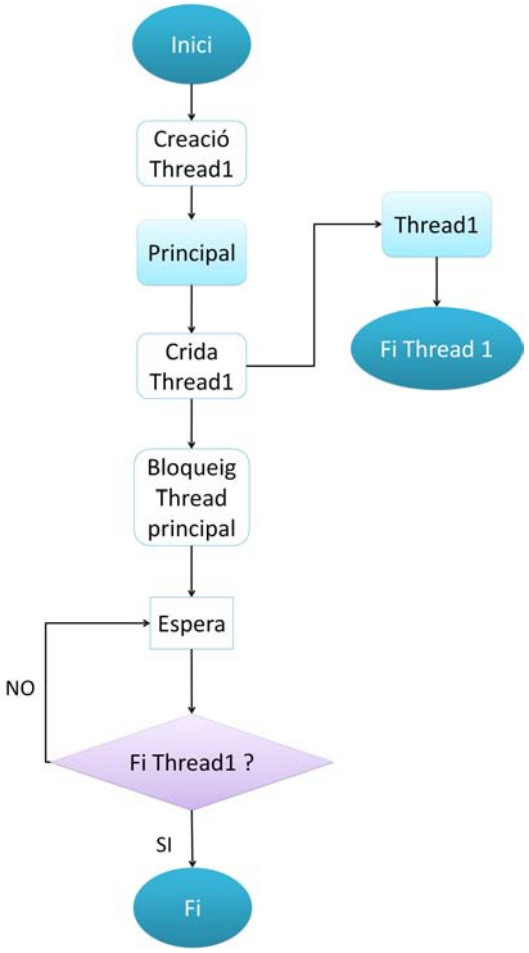


Figura 6.1: *Creació i blocatge d'un thread*

A la figura 6.1 podem veure un exemple molt bàsic, primer es crea el thread1 i es crida a la seva funció asociada, seguidament es bloqueja el `main()` quan es crida a la funció `pthread_join()`. Fins que el thread1 no hagi acabat totes les seves operacions no acabarà el programa.

6.2.2 Vida d'un thread

Durant la seva vida, un thread passa pels quatre estats bàsics que es descriuen a la taula 6.2. Els threads començen a l'estat de "preparat". Si no està bloquejat ni esperant a cap variable de condició, executarà el seu codi fins que acabi. En el cas que estigui bloquejat, aquest anirà esperant a mesura que els altres processos s'executen(si es que n'hi ha).

Estat	Significat
Preparat	El thread està preparat per funcionar, però està esperant un procés.
En funcionament	El thread està funcionant
Bloquejat	El thread està esperant alguna senyal, com una variable de condició
Acabat	El thread ha acabat, un cop s'ha executat la funció `pthread_exit`.

Taula 6.2: *Estats d'un thread*

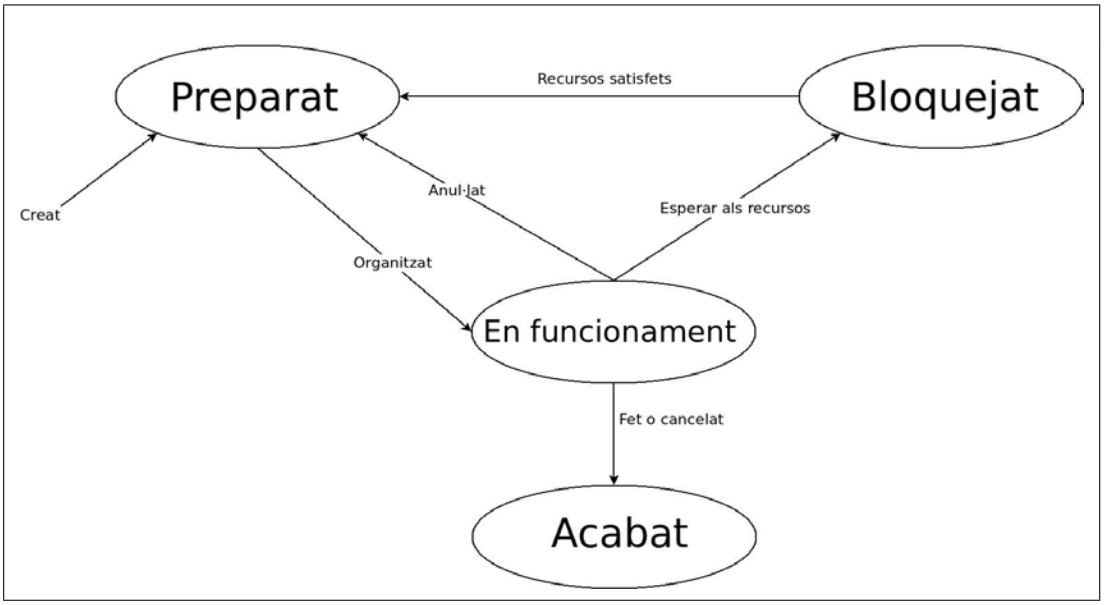


Figura 6.2: *Estats de transició d'un thread*

Creació

El "thread inicial", `main()` o funció principal, com a qualsevol altre programa, es crea al iniciar-se el programa. En canvi els demés threads, hauran de ser cridats des d'algun punt del programa, sigui des del `main()` o des d'altres threads. Així doncs, almenys un thread haurà de ser cridat des del `main()` i a partir d'aquest anar cridant la resta.

Quan un nou thread és creat, comença amb l'estat de preparat. El temps que tardi en passar a l'estat de funcionament dependrà del temps que tardi el programa a organitzar els diferents threads i per això dependrà del número de threads que tinguem. Un fet a tenir en compte sempre en la creació de threads és que no existeix sincronització entre la devolució que fa la funció `pthread_create()` i la organització amb els nous threads. Això vol dir que el thread pot començar abans que la funció retorni un valor.

Inici

Un cop el thread ha sigut creat, el primer que farà és executar un seguit d'instruccions que ens portaran a la funció d'inici del thread que hem especificat com a argument a la funció `pthread_create()`. Quan la funció del thread retrona un valor, aquest haurà acabat, mentre que els altres seguiran en funcionament. En canvi al `main()`, un cop acaba, el programa finalitzarà immediatament.

Haurem de tenir en compte també que el main funciona amb una pila per defecte, que pot créixer considerablement. Aquesta pila pot estar limitada en algunes implementacions, i el programa pot fallar si s'emplena del tot.

Funcionament i bloqueig

La majoria dels threads ocasionalment queden "adormits". Un thread pot quedar "adormit"perquè necessita un recurs que no està disponible en un moment determinat (esdevé bloquejat) o perquè és anul·lat per un altre thread. Un thread dedica la major part del seu temps en tres estats: *preparat, en funcionament* i *bloquejat.*

- El thread estarà *preparat* just després de ser creat, i també quan deixa d'estar bloquejat. Els threads que estan *preparats*, es queden esperant a posar-se en *funcionament*
- Un thread passa a estar en *funcionament* quan, estant *preparat*, el programa el selecciona per ser executat.
- Quan un thread queda *bloquejat* pot ser per vàries raons, la crida d'un mutex que esta bloquejat, l'espera a una variable de condició o l'esperar d'una operació d'entrada/sortida que no pot ser realitzada immediatament.

Acabat

Normalment un thread acaba retornant un valor des de la seva funció associada. El valor que retornarà la funció del thread, dependrà del que nosaltres li especifiquem, però pot donar-se el cas que aquesta retorni el valor `NULL` com per exemple si cridem a la funció `pthread_exit()`.

Es pot donar el cas també, que un cop acabat el thread, aquest doni pas a l'execució d'un altre, tal com passa quan un thread ha cridat a la funció `pthread_join()` i està esperant a que acabi un thread determinat.

6.3 Sincronització

Segurament quan fem un programa amb threads, en algun moment o altre necessitarem compartir dades entre els diferents threads. Per a fer això sense tenir cap mena de problema amb les dades, necessitarem sincronitzar els nostres threads. Per a la sincronització tenim dos eines bàsiques, els mutex, i les variables de condició. Una altra opció que s'usa per a la protecció de dades en sistemes multiprocessos, són els semàfors, aquesta però, no ha estat contemplada en aquest treball.

6.3.1 Mútex(exclusió mútua)

Com ja hem dit, l'ús de threads implica que haurem de compartir dades entre els threads. Per evitar tenir problemes amb l'accés a les dades haurem d'assegurar-nos que aquestes estan "mútuament excloses". Això significa que només a un thread li està permès escriure a la vegada. La bilioteca Pthreads proporciona una forma especial del semàfor de Edsger Dijkstra que anomenem mutex. El mutex serà la solució al nostre problema. Quan un thread bloqueja un mutex que està protegint dades que comparteixen diferents threads, en aquest precís moment, només pot escriure el thread que ha bloquejat el mutex. En el cas de que altres threads intentin escriure a les dades protegides, aquests no podran.

Creació i destrucció d'un mútex

Igual que un thread, un mutex és representat en el nostre programa per una variable, en aquest cas del tipus `pthread_mutex_t`. No podem fer còpies d'un mutex. El que podem fer és definir-ne varis. Si volem definir un mutex estàticament, haurem d'utilitzar el macro `PTHREAD_MUTEX_INITIALIZER`:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

La diferència entre crear un mutex estàticament i dinàmicament recau en que en els mutex estàtics no cal tenir les dades que protegeixen associades. En canvi, en els dinàmics sí.

```
struct estructura{
pthread_mutex_t mutex;
int variable;
}
data = malloc(sizeof(estructura));
pthread_mutex_init(&data->mutex,NULL);
```

Si creem un mutex estàticament, amb els atributs per defecte, no caldrà que un cop acabem d'usar-lo el destruïm. En canvi si el creem dinàmicament, caldrà que el destruïm.

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Un cop creat el mutex, ja ens podem disposar a usar-lo. Està bé crear el mutex a nivell global, ja que d'aquesta manera podrem fer-lo servir des de qualsevol lloc del programa. Si el declaréssim a la funció principal, només podríem protegir dades d'aquesta funció. O en el cas que el declaréssim dins d'un thread només ens serviria per aquest thread determinat. Així que si el declarem globalment, podem utilitzar els mutex des de la funció principal o des de qualsevol thread.

Bloquejar i desbloquejar un mútex

Per fer ús d'un mutex, haurem de bloquejar-lo. Un cop bloquegem el mutex en algun lloc del codi, a partir d'aquí les dades que hi hagin al codi quedaran protegides.

```
Int pthread_mutex_lock (&mutex);
```

Un cop bloquegem un mutex, no podem tornar a bloquejar-lo, en el cas de que ho féssim, el programa retonaria un error. Per desbloquejar un mutex utilitzarem la comanda:

```
Int pthread_mutex_unlock (&mutex);
```

A la taula 6.3 es veu clarament com el mutex protegeix les dades. Com que el thread A comença primer, bloqueja el mutex, i seguidament el thread B intenta bloquejar el mutex però no pot i s'ha d'esperar fins que l'altre el desbloquegi. D'aquesta manera impedim que el thread B accedeixi a les dades compartides fins que el thread A no hagi acabat de manipular-les.

Thread A	Thread B
`pthread_mutex_lock(&mutex);`	
`VariableA = 1;`	`pthread_mutex_lock(&mutex);`
`VariableB = 2;`	
`pthread_mutex_unlock(&mutex);`	
	`Variable1 = VariableA;`
	`Variable2 = VariableB;`
	`pthread_mutex_unlock(&mutex);`

Taula 6.3: *Protecció de dades amb mutex*

Així doncs, si per exemple tenim una funció en la que un bucle va incrementant una variable i volem protegir-la, per a cada iteració del bucle haurem de bloquejar el mutex i desbloquejar-lo, en cas contrari se'ns bloquejaria el thread. Per a casos com aquest, podem fer servir la funció `pthread_mutex_trylock` de la mateixa manera que les anteriors.

```
Int pthread_mutex_trylock (&mutex);
```

La diferència de fer trylock i lock està en que quan fem trylock, bloquegem el mutex només en el cas en que no estigui bloquejat. D'aquesta manera, impedim que la funció ens retorni un missatge d'error a l'hora de intentar tornar bloquejar el mutex.

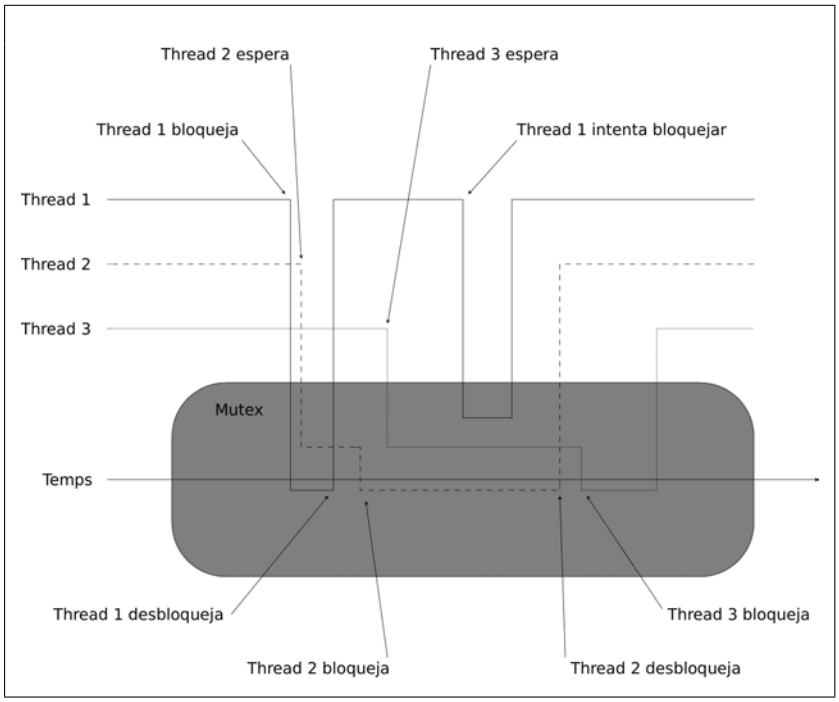


Figura 6.3: *Modes d'operació d'un mutex*

A la figura 6.3 tenim un exemple de com diferents threads bloquegen un mutex. En aquest cas tenim tres threads, els quals en diferents moments intenten bloquejar el mutex. El thread 1 és el primer que el bloqueja, de manera que quan el thread 2 l'intenta bloquejar s'ha d'esperar fins que el thread 1 el desbloquegi. Igualment passa quan ho intenta el 3, s'haurà d'esperar al 2. En un moment donat el thread 1 torna a intentar bloquejar el mutex (*trylock*), però el té bloquejat el 3 i segueix sense bloquejar el mutex.

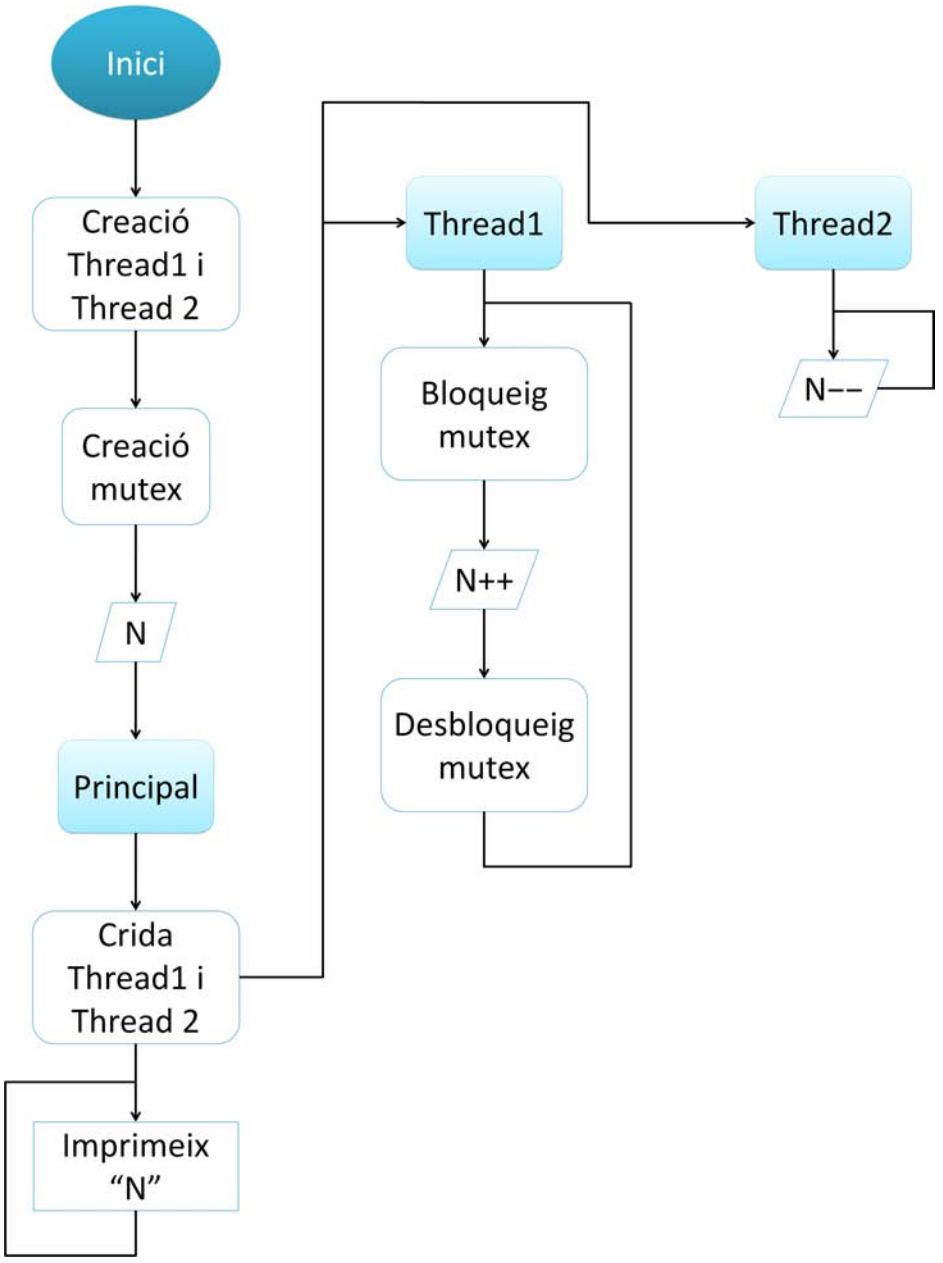


Figura 6.4: *Blocatge i desblocatge d'un mutex*

A la figura 6.4, podem veure un exemple d'un programa en el que es protegeix la variable N. Els tres processos principal, thread1 i thread2 són bucles infinits, en els que, al principal anirem mostrant el contingut de la variable, mentre que en els threads anirem modificant el contingut de la memòria. En el thread1 incrementarem N, mentre que al thread2 la decrementarem, però com que només al thread1 protegim la variable, el thread2 només podrà accedir a N en els moments en que es desbloqueja el mutex. El resultat serà doncs que la variable anirà augmentant, sense pràcticament poder decrementar.

6.3.2 Variables de condició

Usem les variables de condició per a controlar com els diferents processos accedeixen a la memòria. Les fem servir per sincronitzar els mutex. Per exemple si tinguéssim dos processos que van bloquejant i desbloquejant els mutex a la vegada, un cop un thread ha bloquejat un mutex, l'altre no pot bloquejar aquest mateix mutex. Així doncs o bé es bloqueja o bé utilitzem la funció `pthread_mutex_trylock` per a que no se'ns bloquegi el thread. Ara bé, de cap de les maneres protegirem les dades. La solució serà doncs utilitzar variables de condició de manera que podrem triar quin thread bloqueja el mutex en cada moment.

Creant i usant variables de condició

Com en els altres casos, representarem una variable de condició declarant-la del tipus `pthread_cond_t`. Igual que en els mutex, no podem fer una copia d'una variable de condició, encara que si que podem crear-ne vàries. Normalment declararem les variables de condició amb els atributs per defecte, i ho especificarem en el moment en que creem la variable.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Definirem també la nostra variable de condició globalment de manera que un cop creada poguem disposar d'ella en qualsevol moment i lloc del programa.

En el cas en que definíssim una variable de condició dinàmica, un cop acabi l'haurem d'eliminar amb la funció `pthread_cond_destroy`.

Esperant i despertant una variable de condició

Cada variable de condició haurà d'estar associada amb un mutex específic. Quan un thread està esperant a una variable de condició, aquest ja ha d'haver bloquejat el mutex associat. Un cop tenim tenim el mutex associat bloquejat, podrem esperar a la variable de condició.

```
pthread_cond_wait (&cond,&mutex);
```

Quan cridem a la funció `pthread_cond_wait`,li haurem d'especificar l'identificador de la variable de condició i l'identificador del mutex associat. Si ho fem d'aquesta manera, el thread es quedarà esperant indefinidament fins que en algun moment donat "despertem" la variable de condició.

```
pthread_cond_signal (&cond);
```

Quan "despertem" la variable de condició, passant-li l'identificador corresponent, el codi que segueixi s'executarà i es podrà escriure a la memòria de manera segura. El thread s'anirà executant fins que s'acabi o fins que es torni a bloquejar el mutex. Ara bé amb la funció `pthread_cond_signal` despertem un sol thread que estigui esperant a la variable de condició. En el cas en que tinguéssim varis threads esperant a la mateixa variable de condició quan cridem a la funció ens activarà el thread que es decideixi per scheduling. Si volguéssim despertar tots el threads que esperin a una mateixa variable de condició haurem de cridar una funció `pthread_cond_broadcast`.

```
pthread_cond_broadcast (&cond);
```

Una altra opció que tenim amb les variables de condició es de fer esperar al thread un temps determinat.

```
pthread_cond_timedwait (&cond, &mutex,&expiration);
```

D'aquesta manera, no ens hem de preocupar per activar el thread, sinó que li passarem un temps específic com a argument.

A l'exemple de la figura 6.5, podem veure el funcionament de les variables de condició. En aquest cas, es creen 2 variables de condició per a que poguem escollir quin procés executar. Així doncs, el main esperarà a que se li entri 1 o 2 i en funció del que li entrem, el programa simplement permetrà escriure a un thread o a l'altre, a la variable que comparteixen, i amb el tercer s'anirà mostrant el contingut.

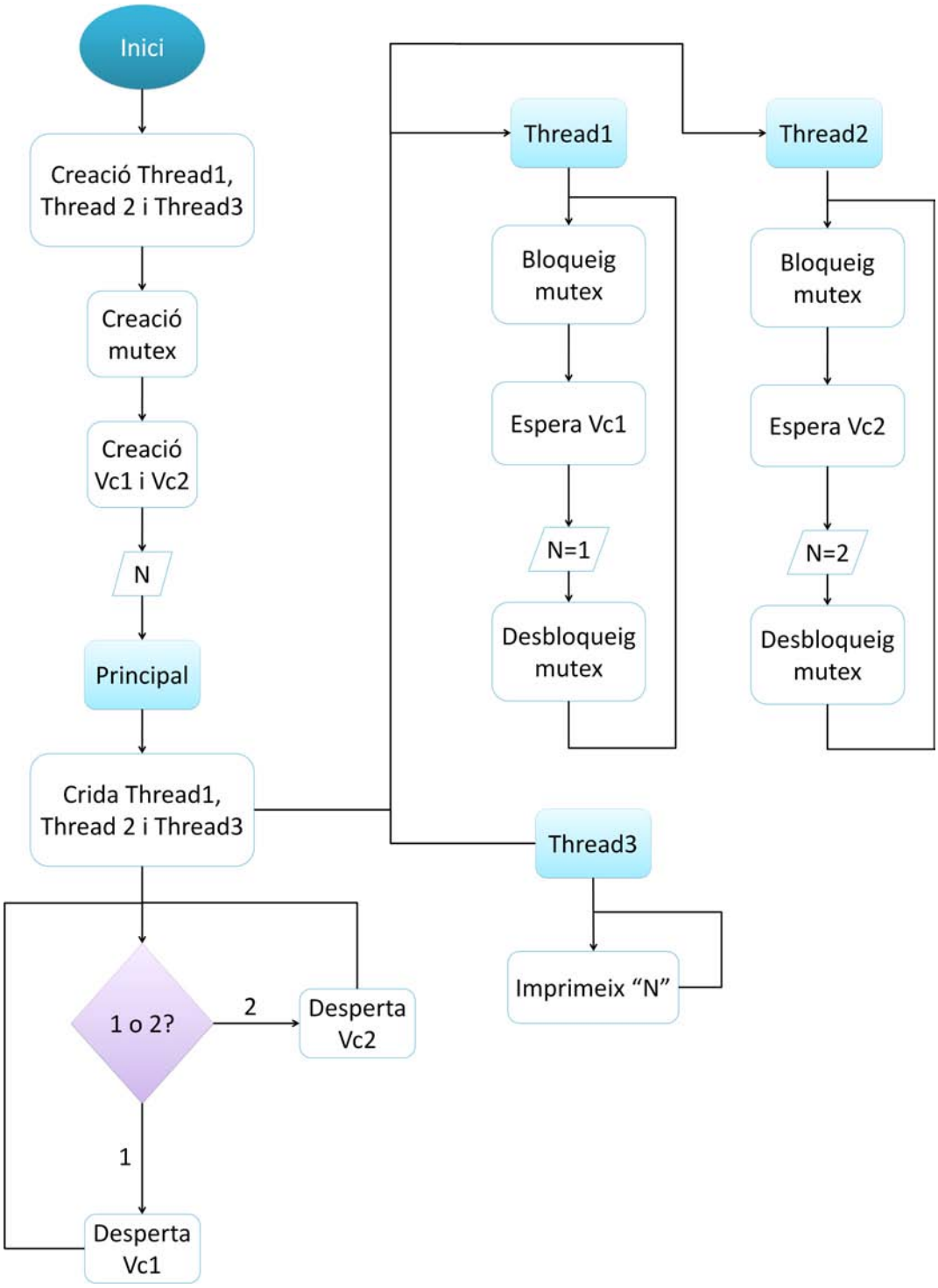


Figura 6.5: *Us de variables de condició*

Capítol 7

Virtualització

7.1 Virtualització a xarxes Wireless

A xarxes de comunicacions, virtualització[9][7] es un terme molt ampli que en la majoria de casos involucra l'intercanvi de recursos físics d'un sistema. Quasi tot es pot virtualitzar en una xarxa, des de nodes físics(CPU, memòria) a tots els enllaços que els connecten. Les xarxes sense fils, són un recurs especial perquè poden ser considerades com un recurs comú a tota la xarxa física, ja que no pertany exclusivament a un node específic, sinó a tota la xarxa en si.

En general, les tècniques de virtualització del medi wireless estan enfocades per a l'ús de transmissions sense fils de manera coordinada, és a dir cada operador virtual té a la seva disposició la xarxa durant intervals de temps específics.

La virtualització a les xarxes wireless, té dos propòsits principals:

- **Compartir la mateixa infraestructura física**. Un exemple ben clar l'hem vist al capítol 1 amb els VAPs.
- **Aprofitar els recursos de la xarxa** d'una manera més eficient.

7.2 Tècniques d'accés múltiple

A les xarxes wireless, algunes de les tècniques[5] bàsiques d'accés múltiple o virtualització són les següents:

1. ***Frequency Division Multiple Access* (FDMA).**
   Es virtualitza cada node fent particions en freqüència. Per exemple, en un node 802.11a amb 12 freqüències ortogonals, es pot fraccionar en 4 nodes virtuals dividits en 3 freqüències per node virtual.

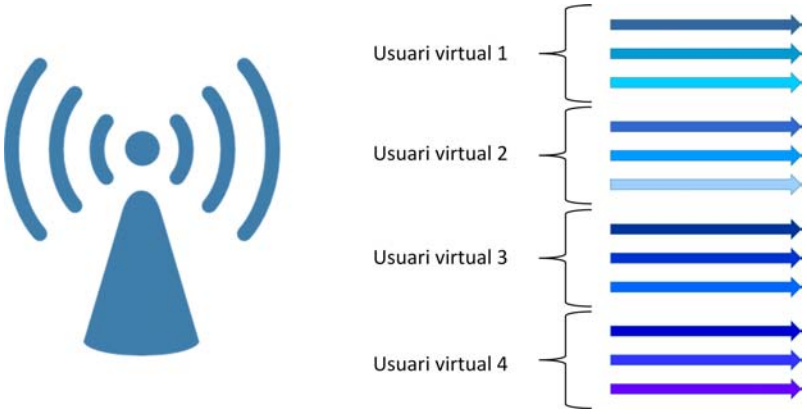


Figura 7.1: *Virtualització FDMA*

Encara que cada node pot ser dividit en diferents nodes virtuals, al llarg de la freqüència, canviar de un node virtual a un altre no es instantani. La commutació de canals per a les tarjes Atheros és de 5 ms, mentre que per tarjes Intel es de 20 ms.

A causa d'aquests temps de commutació (prenem els 5ms de les tarjes Atheros), cada node virtual haurà d'aconseguir el seu torn per transmetre amb la política *round robin* un temps d'espera de $t_e = (5 + x) * n$ ms on x es el temps que cada node virtual està actiu, i n el número de nodes virtuals.

2. ***Time Division Multiple Access* (TDMA).**
La virtualització es realitza mitjançant divisions de temps. Això vol dir que, diferents usuaris usen una mateixa divisió de freqüència, en diferents "*slots*"o porcions de temps.

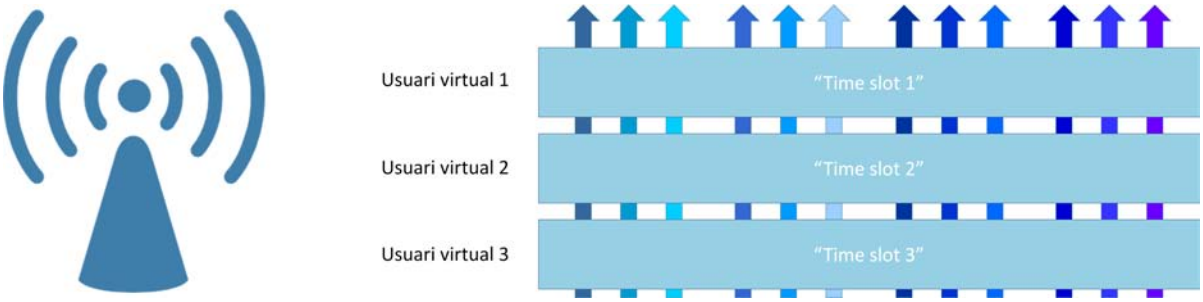


Figura 7.2: *Virtualització TDMA*

Per exemple, un node 802.11a pot ser dividit en 3 nodes virtuals, separats en 3 porcions de temps. A mesura que el nombre de nodes virtuals va incrementant, cada node virtual s'haurà d'esperar més per aconseguir el seu torn.

Si tenim en compte el temps de commutació(1-10 ms a LINUX), el temps que hauran d'esperar el seu torn per transmetre de forma round robin, serà $t_e = (10 + x) * n$ on x es el temps que cada node virtual està actiu, i n el número de nodes virtuals.

3. ***Code Division Multiple Access* (CDMA).**
Es fa la virtualització d'un node fent particions de codi. Això s'aplica a estacions base que operin usant codis ortogonals.

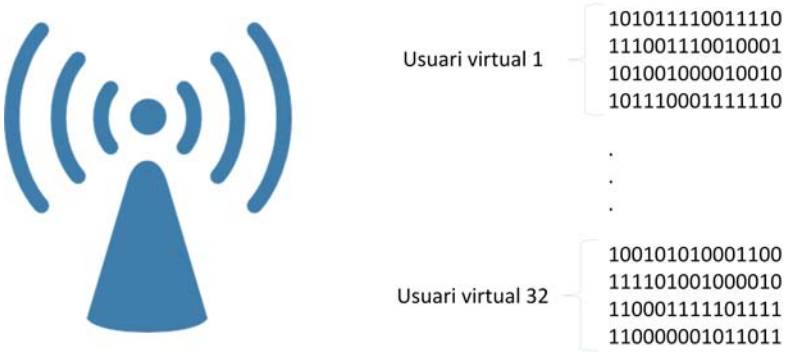


Figura 7.3: *Virtualització CDMA*

Per exemple, un node que utilitzi l'estàndard CDMA2000 1x(mòbils de tercera generació **3G**)[3] podrà com a molt abarcar 128 codis ortogonals. Aquests podran ser dividits en N nodes, assignant-los a M codis ortogonals per node virtual de la forma $N * M = 128$ com es mostra a la figura 7.3 $M = 4$ i $N = 32$

7.3 Estructura de l'entorn de virtualització

Virtualització significa bàsicament compartir recursos. La situació que tractarem correspon a un node determinat, rebent tràfic generat per diferents nodes virtuals.

L'organització del tràfic generat és molt simple, i consisteix en anar guardant en una cua el tràfic de paquets rebuts dels diferents usuaris virtuals, i depenent de la tècnica de virtualització escollida, els paquets d'un usuari o un altre seran enviats a la interfície sense fils per ser transmesos.

L'estructura de l'entorn de virtualització consistirà en tres parts diferenciades però ben complementades.

- **Generador de tràfic.**
El generador de tràfic, és l'encarregat de crear un tràfic de paquets per a cada usuari virtual, que serà enviat a la cua de transmissió. Per a la generació de tràfics,

es poden usar un seguit d'estadístiques o distribucions aleatòries de probabilitat, com per exemple la distribució Poisson. D'aquesta manera podem aconseguir que els tràfics de dades s'acostin més a la realitat.

- **Cua.**
  Cada usuari virtual anirà emplenant una cua on s'anira guardant el tràfic generat. Cada usuari tindrà un tipus de tràfic diferent, per tant, s'anirà emplenant la cua tal i com arribi del generador de tràfic, sense cap mena d'ordre. En el mateix ordre que els paquets van arribant a la cua, després seran transmesos per l'organitzador.

- **Organitzador.**
  L'organitzador és l'encarregat de controlar la cua i preparar els paquets per a ser transmesos. Depenent de la tècnica de virtualització escollida, l'organitzador seguira una estratègia o una altra per gestionar la cua i transmetre els paquets. Per exemple a TDMA, l'organitzador assignarà porcions de temps a cada usuari de igual (*Round Robin* (RR)) o diferent (*Weighted Round Robin* (WRR)) duració.

  El número de paquets que es transmetran, dependrà doncs de la duració de la fracció de temps que se li hagi assignat al usuari virtual.

Per a que tots aquests elements funcionin correctament entre sí, diferenciarem tres etapes:

1. **Configuració.**
   Abans de començar a generar tràfic, haurem de definir una sèrie de paràmetres, com: el número de nodes virtuals, el tipus de tràfic que generarem, número de paquets, tamany de la cua...

2. **Generació de tràfic.**
   La generació de paquets vindrà determinada per els paràmetres definits al pas anterior i pel tipus de tràfic desitjat.

3. **Algorisme d'organització**
   Finalment l'organitzador serà l'encarregat, depenent del tipus de política escollida, de transmetre els paquets generats.

7.4 Implementació d'un entorn de virtualització a una xarxa Wireless

En aquesta part del projecte, s'hi veuen involucrats tots els aspectes tractats en els capítols anteriors. Des de la configuració de la tarja Wifi amb els *drivers* i les comandes LINUX fins a la implementació del programa amb l'ajut dels sockets i threads.

L'estructura del programa a realitzar serà molt semblant al vist a la secció 7.3. Tindrem 2 usuaris virtuals que aniran generant un tràfic de paquets i l'aniran guardant a una cua.
Finalment tindrem una sèrie de processos que faran la funció de l'organitzador. Agruparem aquesta sèrie de processos i l'anomenarem "**Master**". Aquest s'encarregarà d'ordenar la cua i fer les transmissions.

Així doncs, dividirem les tasques següents en dos parts: una primera part en la que es configurarà la xarxa i els paràmetres adequats per a la virtualització i una segona part on implementarem el programa en sí.

7.4.1 Configuració de xarxa i paràmetres de virtualització

Configuració de xarxa

La configuració de la xarxa serà molt simple, ja que només necessitarem dos ordinadors, a un dels quals hi farem la virtualització. És a dir, tindrem dos ordinadors connectats, simulant que tenim varis nodes connectats a un sol ordinador.

Com ja hem dit, no serà molt difícil d'aconseguir aquesta configuració de xarxa. Per fer-ho, només haurem de seguir els passos realitzats a la secció 4.2 del capítol 4, on tindrem dos nodes, un configurat com a AP i l'altre STA connectat a l'AP.

Paràmetres de virtualització

Per al correcte funcionament del programa, serà molt important escollir bé els valors dels paràmetres de virtualització que després utilitzarem. Per a cada part del programa haurem de tenir en compte diferents aspectes específics, que tractarem en parts diferents: **tràfic generat**, **temps de generació i transmissió**, **cua de transmissió** i **política d'organització**.

- **Tràfic generat.**
  Com hem vist a la secció 7.3 per a la generació de tràfic és molt normal l'ús d'estadístiques i distribucions de probabilitat conegudes. Per realitzar les simulacions, implementarem el programa amb dos variants, un que ens transmeti paquets amb un temps fix i un altre que ens transmeti paquets amb un temps aleatori.

  En comptes de que cada usuari enviï un paquet a la cua, el que faran aquests, serà enviar una petició a una cua comuna de transmissions. Això vol dir que a la cua no s'aniran guardant els paquets dels diferents usuaris, sinó un identificador que després el Master, al accedir a la cua en funció de l'identificador que es trobi, enviarà un paquet d'un usuari o un altre.

Per a la generació del tràfic amb temps aleatoris, hem escollit la distribució de probabilitat exponencial. La funció densitat de probabilitat de la distribució exponencial es la següent:

$$f(x) = \lambda e^{-\lambda x} \qquad x \leq 0 \tag{7.1}$$

La variable $\lambda > 0$ és el paràmetre de la distribució on $\frac{1}{\lambda}$ es la mitja. Podem generar variables aleatòries amb distribució exponencial a partir de una variable amb distribució uniforme. Donada una variable U a l'interval (0,1), obtindrem la variable aleatòria amb distribució exponencial mitjançant l'expressió 7.2.

$$x = \frac{-ln(U)}{\lambda} \tag{7.2}$$

A la figura 7.4 podem veure l'histograma d'una variable aleatòria x amb una mitja $\frac{1}{\lambda} = 1$.

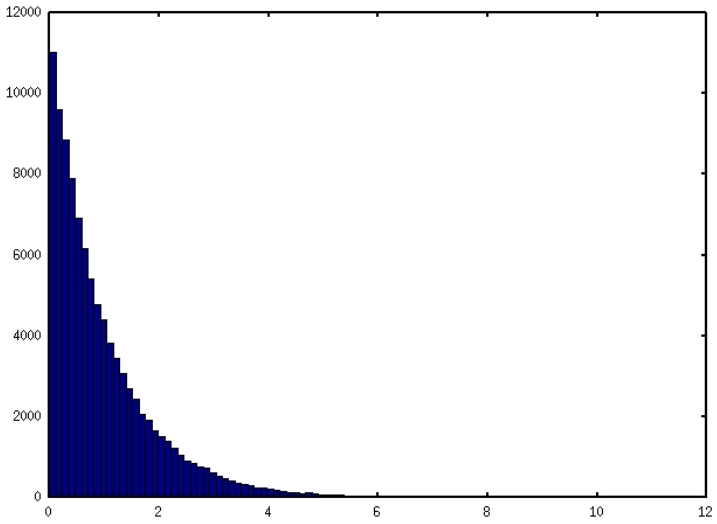


Figura 7.4: *Funció densitat de probabilitat de la distribució exponencial,* $\lambda = 1$

- **Temps de generació i de transmissió.**
  Definirem dos temps que caracteritzaran l'entorn de virtualització. El temps de generació T_g, serà el temps que tarda cada usuari a posar una petició a la cua de transmissió. El temps de transmissió T_t, serà el temps que tarda el Master a enviar un paquet, sigui l'usuari que sigui.

  Es podran donar diferents situacions a estudiar en les que els usuaris tinguin temps de generació mes grans, mes petits o iguals al temps de transmissió.

- **Cua de transmissió.**
  La cua de transmissió serà un vector de tamany finit en el que anirem guardant com ja hem dit, les diferents peticions que faci cada usuari. Aquest vector haurà de tenir un tamany finit perquè cada vegada que es faci una transmissió el Master eliminarà la petició del usuari que s'hagi transmès i farà córrer una posició la cua. Per aquesta raó no pot ser una cua infinita, ja que si tenim un usuari amb un T_g molt petit, emplenaria la cua massa ràpid i en pocs segons tindríem una cua tan gran que el programa no podria gestionar.

  Serà molt important escollir bé el tamany de la cua de transmissió, ja que com més gran sigui més tardarà el programa en recórrer-la. D'altra banda, tampoc serà bo que sigui massa petita ja que llavors els usuaris ràpids es veuran limitats a fer poques peticions. La sol·lució doncs, serà trobar un tamany intermig que ens permeti treballar adequadament.

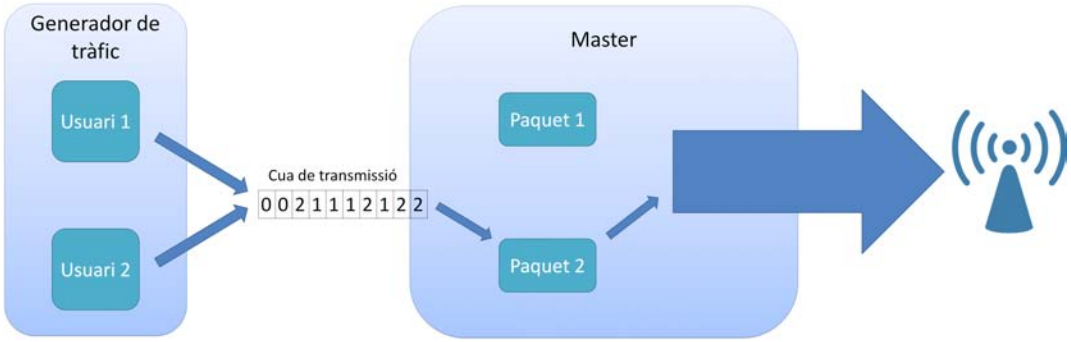


Figura 7.5: *Esquema del sistema de virtualització*

Haurem d'anar molt en compte a l'hora d'escollir el tamany adequat de la cua. Perquè a més dels aspectes que acabem d'explicar, també haurem de tenir en compte els T_g i T_t. El Master fa córrer una posició de la cua immediatament després de fer una transmissió, per a que a la següent transmissió el Master agafi la nova petició. El temps que es tardi a recórrer tota la cua per moure les posicions haurà de ser més petit que el T_t. Per tant, el tamany de la cua haurà de ser suficientment petit perquè el programa el pugui recórrer en menys temps que el T_t.

- **Política d'organització.**
  Igual que fa l'organitzador vist a la secció 7.3, el Master serà l'encarregat d'organitzar la cua de transmissió i transmetre els paquets depenent la política escollida. Les polítiques d'organització que seguirà el Master seran dos, **FIFO (*First In First Out*)** i **RR (*Round Robin*)**.

  Quan escollim la política FIFO, l'única feina del Master serà anar desplaçant la cua de transmissió i transmetre el paquet corresponent a la petició que estigui a la

primera posició de la cua. Es a dir, en el mateix ordre que la petició d'un usuari entra a la cua, s'enviara el seu corresponent paquet.

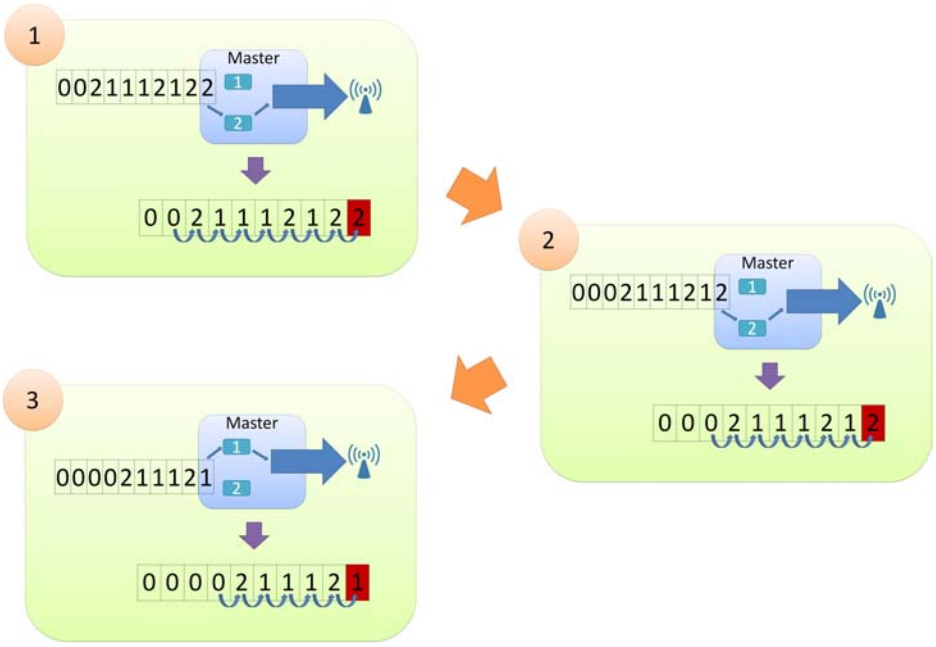


Figura 7.6: *Organització de la cua FIFO*

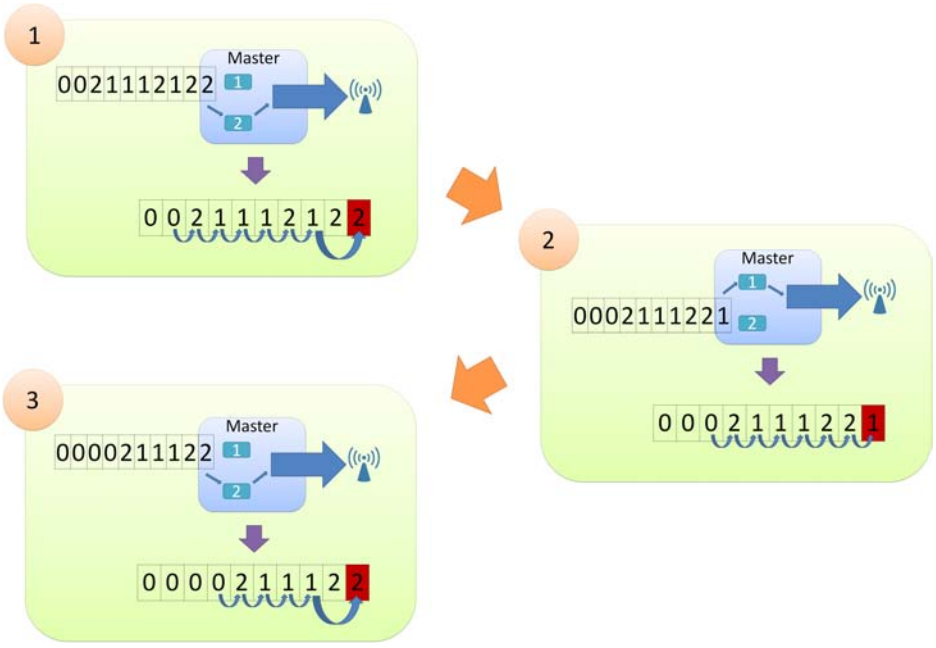


Figura 7.7: *Organització de la cua RR*

En el cas RR, serà diferent, ja que el Master anirà buidant la cua d'una forma diferent. Independentment de la petició que toqui, el Master transmetrà el paquet d'un usuari diferent cada vegada. Això vol dir que el Master haurà de recórrer la cua cada vegada en busca d'una petició de l'usuari al que li toqui transmetre, eliminarà la petició i desplaçara la cua a partir de la posició actual.

7.4.2 Implementació del programa

L'implementació del programa final consistirà en desenvolupar el sistema vist al capítol 2 on s'apliquen les tècniques network coding i virtualització conjuntament. La part de virtualització serà en la que ens centrarem per realitzar els càlculs i posteriorment obtenir resultats.

A la figura 7.8 podem veure un esquema total del programa, on es realitzen tots els passos necessaris per a la tècnica network coding i implícitament, la virtualització. En aquest esquema, s'hi mostra la part dels usuaris i el master transmetent i rebent paquets simultàniament. A més també s'hi pot observar la funció del node 2, que serà la de rebre els paquets dels usuaris, codificar-los i reenviar-los.

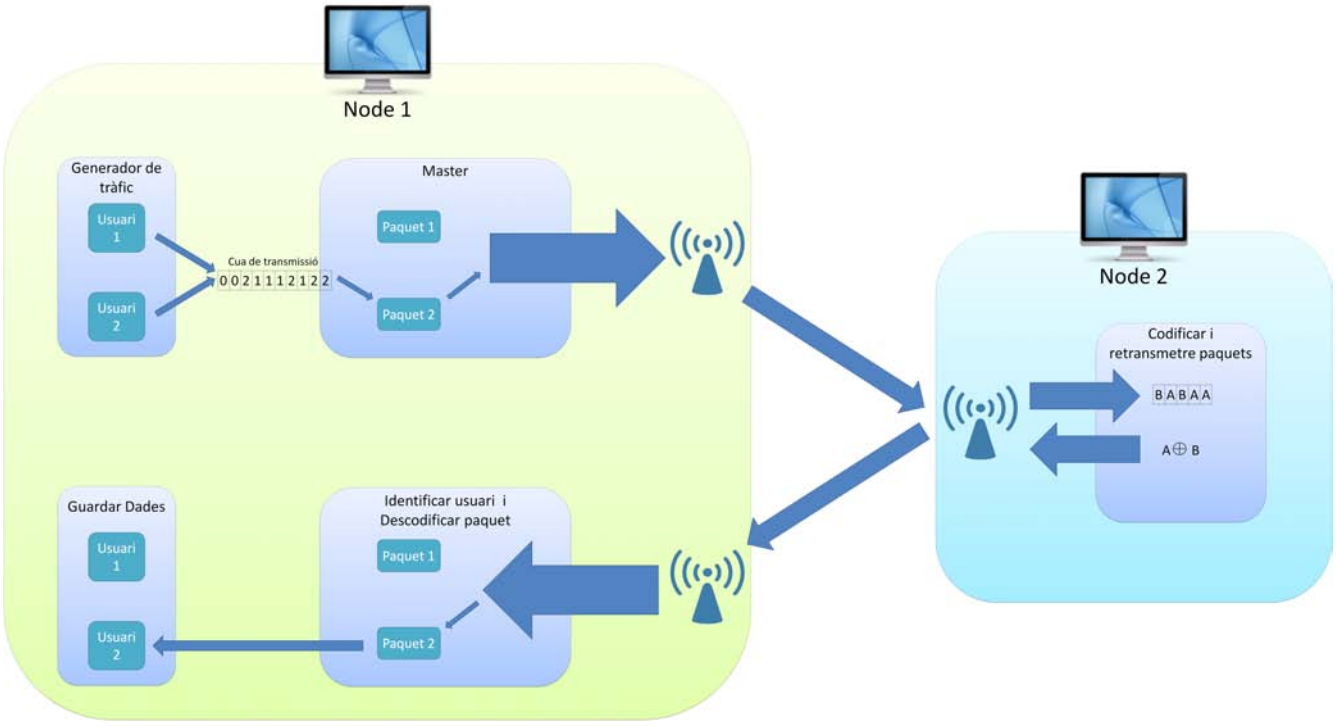


Figura 7.8: *Esquema del programa final*

Tant per a les parts de virtualització com per a la resta, el programa està basat en l'ús de threads per a fer funcionar diferents processos a l'hora. Un dels aspectes més

importants del programa, serà la protecció de dades amb els mutex, sobretot perquè els diferents processos comparteixen variables globals, com per exemple la cua de transmissió, que l'aniran manipulant els usuaris i el Master simultàniament .

Els diferents threads seran cridats des del `main()` al iniciar-se el programa tal i com es veu a l'exemple següent.

```
int main(void)
{
    thr_id = pthread_create(&p_thread, NULL, &cua, NULL );
    thr_id3 = pthread_create(&p_thread3, NULL, &usuari1, NULL );
    thr_id4 = pthread_create(&p_thread4, NULL, &usuari2, NULL );
    thr_id7 = pthread_create(&p_thread7,NULL,&trans,NULL);

    pthread_join(p_thread3,NULL);
}
```

Gràcies a la funció `pthread_join()` el programa no acabarà fins que el thread3 no acabi d'executar-se(podria ser qualsevol thread, per que cap acaba mai d'executar-se).

Cada procés farà operacions diferents a l'hora, on cadascun formarà part dels usuaris virtuals o del Master. Per tant, separarem els processos que formin part dels usuaris i del Master.

Usuaris

Per a la simulació dels dos usuaris virtuals, s'utilitzaran només dos processos, simulant cada procés un usuari virtual. El seu funcionament serà molt simple: cada procés es basarà en la generació de peticions cada cert temps. Aquest temps serà el temps de generació de cada usuari T_g.

Per a fer les peticions, cada usuari anirà posant a l'última posició de la cua de transmissió un número que serà el que els identificarà a l'hora de transmetre.

```
void *usuari1(void *arg)
{
    while(1){
                usleep(500000); //Temps de generacio
                pthread_mutex_lock (&mutex); // Bloquejem mutex
                if(p<=10000){
                cuatrans[p]=1; //Cua de transmissio
                buffer1[m]=1;
                p++;
                m++;
}
```

```
12                  pthread_mutex_unlock (&mutex);
13                  }
14  }
```

A l'exemple podem veure el thread complet d'un usuari, en aquest cas el de l'usuari 1. Com ja hem dit el funcionament és molt bàsic, es tracta d'un bucle infinit, en el que mitjançant la funció `usleep()` podem controlar la freqüència amb que es fan les peticions. Aquesta funció el que fa es retardar l'execució del programa, el temps passat com a argument en microsegons. El temps que li especifiquem a la funció `usleep()` serà el temps de generació de l'usuari. En aquest cas tindrem $T_g = 0,5s$.

Un cop passat el T_g, es bloquejarà el thread amb un mutex, de manera que les dades que es van a modificar a les línies següents, quedin protegides. Protegir la cua de transmissió (`cuatrans[p]`) serà molt important, ja que si no estigués protegida, en qualsevol moment pot accedir-hi l'altre usuari i alterar l'ordre. A més de la cua de transmissió, tindrem un altre vector (`buffer1[m]`) que ens servirà per, més endavant, portar el compte dels paquets que s'envien i es perden.
Com s'ha vist a l'apartat 7.4.1, la cua de transmissió haurà de tenir un tamany finit per no tenir problemes. Després de provar diferents tanyanys de la cua, el que hem trobat més adequat és el d'un vector de 10000 posicions. Encara que per a usuaris amb T_g grans pot resultar una mica gran, per a T_g petits, la cua s'emplena molt fàcilment.

A la línia 6, ens trobem un `if()`, que no permetrà que s'empleni més la cua si ja s'ha emplenat fins a la posició 10000. D'aquesta manera limitem als usuaris a fer peticions si la cua esta plena, i evitem que es perdin paquets innecessàriament.

Finalment per acabar el procés, cada vegada que afegim una petició a la cua, incrementem la variable global `p`, per moure una posició de la cua de transmissió.

Per al cas de temps de generació aleatòris, els threads de cada usuari cridaran a una funció que ens generarà una variable aleatòria i li passarem com a argument a la funció `usleep()`.

```
1  float aleatori(){
2                           numero = (int)( rand()*(1000/(RAND_MAX
                               +1.0)) + 0.5 );
3                           return numero;
4                    };
```

A la funció aleatori l'únic que fem és generar un número aleatori entre 0 i 1000, que després dividirem per 1000 per poder-nos quedar amb els decimals. Seguidament, al thread de l'usuari fem les operacions per obtenir la variable exponencial i el multipliquem per 1000000 per passar-ho a microsegons.

```
n1=((-log(aleatori()/1000))/1);
n1*=1000000;
```

Master

La part del Master engloba les tasques de transmissió de paquets i gestió de la cua de transmissió. Per a cada una d'aquestes tasques, tindrem un procés específic, que realitzaran les operacions necessàries.

Per a la transmissió de dades, el punt més important a tenir en compte, és com transmetem els paquets. És aquí on utilitzarem els sockets. Més concretament, utilitzarem el programa del client UDP.

En comptes d'utilitzar programes diferents, es va crear un fitxer amb una funció que ens fes tots els passos de la comunicació amb sockets, i es va afegir al programa en forma de llibreria. D'aquesta forma ens és més còmode treballar, ja que podem cridar a la funció `client()` des de qualsevol lloc del programa.

L'estructura de la funció `client()` es molt semblant als passos vistos a l'apartat 5.4.2 del capítol 5. Aquesta funció rebrà els arguments `char ip[]`, `char port[]`, `char miss[]`.

```
int client(char ip[], char port[], char miss[])
```

Els arguments, els hi passarem des del thread trans, on `ip[]` serà un vector amb la direcció IP de l'equip on transmetrem les dades, `port[]` el port que utilitzarem i `miss[]` el paquet en si.

El que fa la resta d'operacions del thread és molt simple, es tracta d'un bucle infinit que cada T_t envia el paquet d'un usuari o d'un altre. Dependrà doncs, de l'identificador que es trobi a la primera posició de la cua.

```
void *trans(void *arg)
{
    while(1)
    {
        usleep(100000);// temps de transmissio
        if(cuatrans[0]==1)
        {
            client("127.0.0.1","1235",mis1);//transmissio del
               paquet
            w=1;
            pthread_cond_signal (&cond);//Despertem variable de
               condicio
```

```
11          }
```

En aquest cas només podrà ser 1 o 2, així que si tenim un 0 degut a que la cua està buida, no es transmetrà cap paquet. Cada vegada que es transmeti, es posarà la variable `w` a 1 o 2, que utilitzarem més endavant. Finalment Despertarem la variable de condició, que ens desbloquejarà el thread que gestiona la cua.

La implementació del thread que manipula la cua, tindrà dos variants, per FIFO i per a RR. En els dos casos, del que es tractarà es d'organitzar la cua un cop s'ha fet una transmissió. Es a dir s'eliminarà la petició que s'hagi transmés i es desplaçarà la cua en funció de la política escollida.

Al principi, el thread es bloquejarà amb un mutex, i es farà esperar a la variable de condició del thread de transmissió. Daquesta manera aconseguim que s'ordeni la cua cada vegada que es fagi una transmissió

Quan utilitzem la política FIFO, només ens haurem de preocupar d'anar desplaçant la cua cada vegada, independentment de quines peticions hi hagin.

```
1 for(n=0;n<=10000;n++)
2 {
3       cuatrans[n]=cuatrans[n+1];
4       if(cuatrans[n+1]==0)break;
5 }
6 p=n;
```

Per al cas de RR, serà una mica més complexe. Per a que es transmetin paquets d'un usuari i de l'altre alternativament, haurem d'organitzar la cua de manera diferent a FIFO. Aquí, al accedir a la cua, es buscarà la petició d'un usuari determinat, si no s'hi troba, s'anirà desplaçant fins a trobar-ne una. En el cas de que es recorri tota la cua i no es trobi, es transmetrà un paquet de l'altre usuari.

Capítol 8

Resultats

Les simulacions fetes amb el programa vist al capítol 7 representen una sèrie proves que ens ajudaran a millorar el funcionament de l'entorn de virtualització.

Encara que el programa està dissenyat per a funcionar amb dos equips, les nostres simulacions, només s'han realitzat amb un sol equip. Això es possible gràcies a que amb sockets, podem utilitzar un canal de comunicacions intern, i de la mateixa manera que enviem dades a una certa adreça IP, podem enviar dades a l'adreça pròpia de l'ordinador anomenada ***localhost***, que normalment és 127.0.0.1.

Donat el cas que només utilitzem un ordinador, necessitarem un programa que faci la funció de l'altre equip. Per això s'ha creat un programa especial, que l'únic que farà serà rebre el tràfic generat a l'altre programa i identificar l'usuari de cada paquet. D'aquesta manera podrem veure quants paquets arriben al destí correctament.

Com s'ha vist al capítol 7, s'ha implementat el programa de dues formes, FIFO i RR. Així que, determinar quin és el mode d'operació més adequat en cada cas serà un dels objectius.

Trobar els temps de generació T_g de paquet òptim, també serà un dels objectius de l'estudi. Per a cada simulació doncs, anirem variant els temps de generació dels dos usuaris. Haurem de tenir en compte també el temps de transmissió T_t, que influirà molt a l'hora de veure quants paquets es transmeten o es perden.

8.1 Temps de generació fix

Els diferents temps de generació de cada usuari els trobarem seguint les següents expressions:

$$T_{g1} = \alpha * T_{mig-generacio}$$
$$T_{g2} = (1 - \alpha) * T_{mig-generacio}$$

On T_{g1} i T_{g2} representen els temps de generació de cada usuari. Per tant, per a cada $T_{mig-generacio}$ tindrem varies combinacions dels temps de generació depenent del valor α.

Com ja hem dit el T_g està molt relacionat amb el T_t, és per això que les simulacions es faràn en funció d'aquest temps. És a dir, es faran simulacions amb $T_{mig-generacio}$ diferents, de la seguent manera:

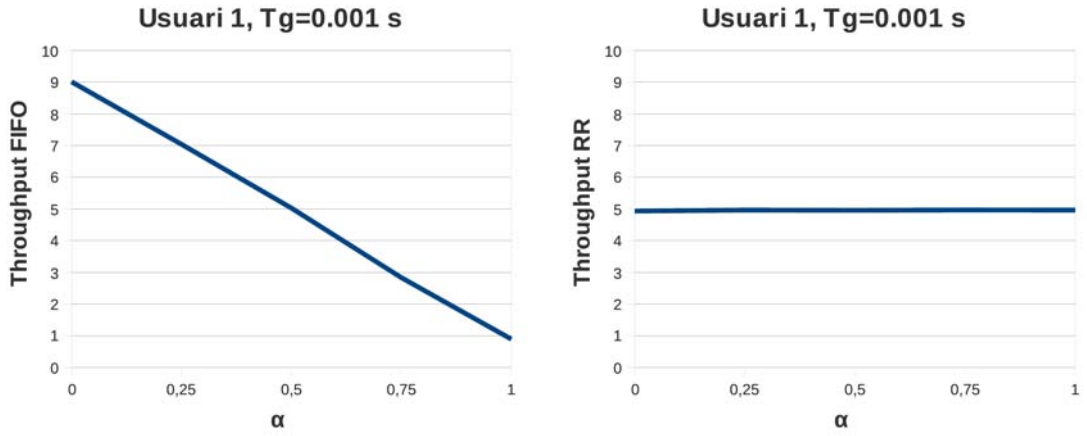
$$T_{mig-generacio} < T_t$$
$$T_{mig-generacio} = T_t$$
$$T_{mig-generacio} > T_t$$

Un cop fetes totes les simulacions, ens disposarem a analitzar els resultats obtinguts. Els paràmetres que determinarem amb les simulacions seran, el ***Throughput***, ***Eficiència*** i ***Fairness***

El primer paràmetre que analitzarem serà el **throughput**. El throughput és una mesura molt comú a xarxes de comunicacions, que trobarem mitjançant (8.1).

$$Throughput = \frac{Numero\ de\ paquets\ rebuts}{Temps\ simulacio} \tag{8.1}$$

El throughput, doncs, ens permetrà veure quants paquets per segon arriben al seu destí. A l'exemple de la figura 8.1 podem veure dos exemples del throughput de l'usuari 1. Els resultats obtinguts per al throughput seran iguals per als dos usuaris, però invertits. És a dir, quan un usuari transmet el màxim de paquets possible, l'altre farà el mínim i viceversa.

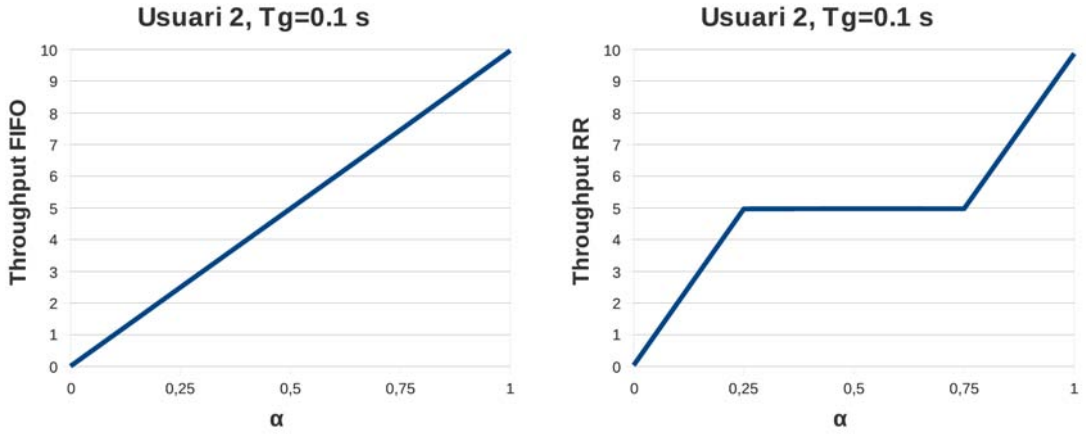


a)Thourghput FIFO usuari 1 *b)Thourghput RR usuari 1*

Figura 8.1: *Throughput*, $T_{mig-generacio} = 0,001s$

Als exemples de la figura 8.1 s'hi pot veure el throughput amb un $T_{mig-generacio} = 0,001$ mesurat per als diferents valors de α, i per a les dues tècniques, FIFO i RR. Com

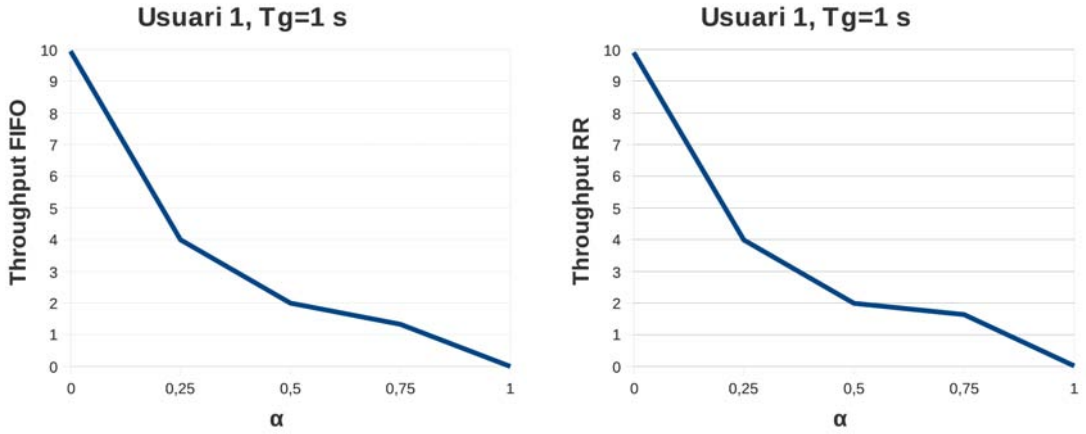
es pot veure, el throughput FIFO, és màxim per a $\alpha = 0$ i mínim per a $\alpha = 1$. En canvi per a RR, es manté constant tot el temps per a qualsevol valor de α. Per al cas del throughput FIFO és evident, l'usuari que més peticions fa, més paquets pot transmetre. Però per al cas de RR, és diferent, com que la diferència de temps entre un i l'altre es tan petita, els dos usuaris tenen temps de posar peticions a la cua, per això, per a qualsevol valor de *alpha* tindrem el mateix throughput.



a)Thourghput FIFO usuari 2 *b)Thourghput RR usuari 2*

Figura 8.2: *Throughput* , $T_{mig-generacio} = 0,1s$

Veiem ara a la figura 8.2, el throughput de l'usuari 2. Al augmentar el $T_{mig-generacio}$ fins a 0,1 segons, per a RR, el throughput canvia considerablement. Entre $\alpha = 0,25$ i $\alpha = 0,75$ es manté igualment constant, però per a $\alpha = 1$ arriba a un *throughput* $= 10$, igual que per a FIFO. Als màxims hi arribem en punts en que envia pràcticament tots els paquets només un usuari, ja que és molt més ràpid i l'altre no te temps de posar cap petició a la cua.



a)Thourghput FIFO usuari 1 *b)Thourghput RR usuari 1*

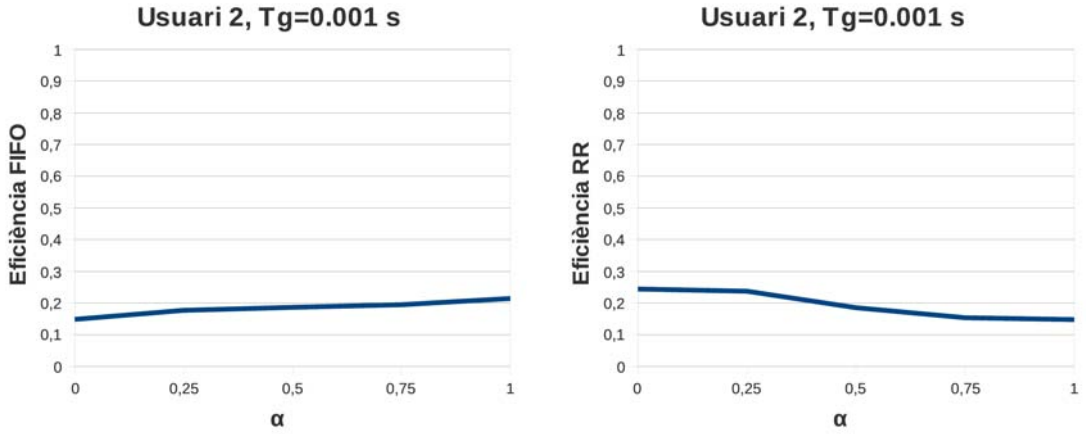
Figura 8.3: *Throughput* , $T_{mig-generacio} = 1s$

Finalment, a la figura 8.3 tenim el throughput per al cas de $T_{mig-generacio} > T_t$ amb $T_{mig-generacio} = 1segon$. Els resultats obtinguts són els mateixos tan per FIFO com per a RR. Als dos casos, el throughput disminueix significativament, degut a que els temps de generació, són més grans que el temps de transmissió. D'aquesta manera, estarem limitant el tràfic de paquets, ja que la cua de transmissió estarà en molts instants buida.

L'**eficiència**, és una mesura que utilitzarem per expressar el numero de paquets rebuts correctament en funció del total. Ho podrem expressar mitjançant (8.2).

$$Eficiencia = \frac{Numero\ de\ paquets\ rebuts}{Numero\ total\ de\ paquets} \tag{8.2}$$

Encara que en realitat no es perdran els paquets, sinó que es deixaran de transmetre. És a dir, per a calcular el número de paquets total, sumarem el número de paquets rebuts, més el número de peticions que no s'han transmès.

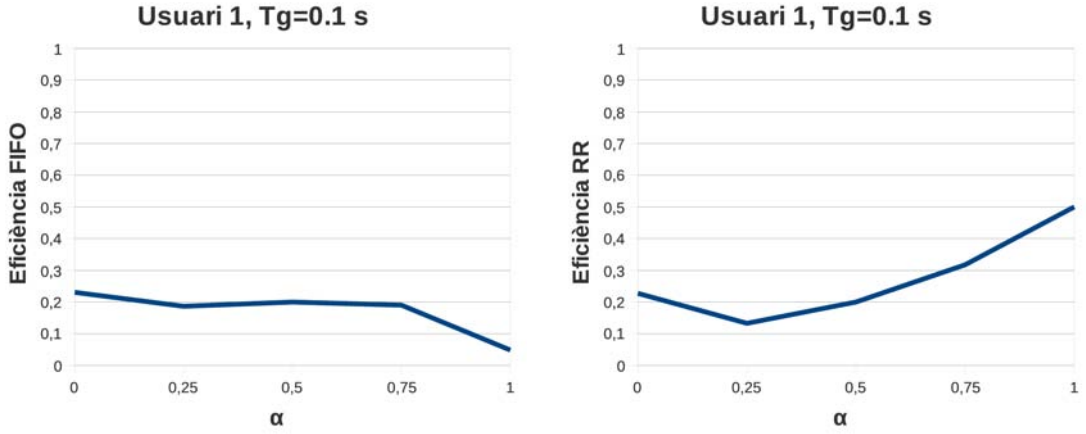


a)Eficiència FIFO usuari 2 *b)Eficiència RR usuari 2*

Figura 8.4: *Eficiència* , $T_{mig-generacio} = 0,001s$

A la figura 8.4 tenim l'eficiència mesurada per a un $T_{mig-generacio} = 0,001s$. Com es pot apreciar, per a temps de generació tan petits, es fan moltes peticions, de manera que es perden molts paquets, tan amb FIFO com amb RR. Per als dos casos doncs, obtenim valors baixos d'eficiència que es mantenen força constants, al voltant de 0,2.

A mesura que anem augmentant el $T_{mig-generacio}$ tindrem millor eficiència, ja que, no es generen tantes peticions, i no s'en perden tantes. A la figura 8.5, encara que per a FIFO no hi ha millora, per a RR sí que millora l'eficiència, sobretot per a valors de α més grans de 0,5. Això es degut a que es perden menys paquets.

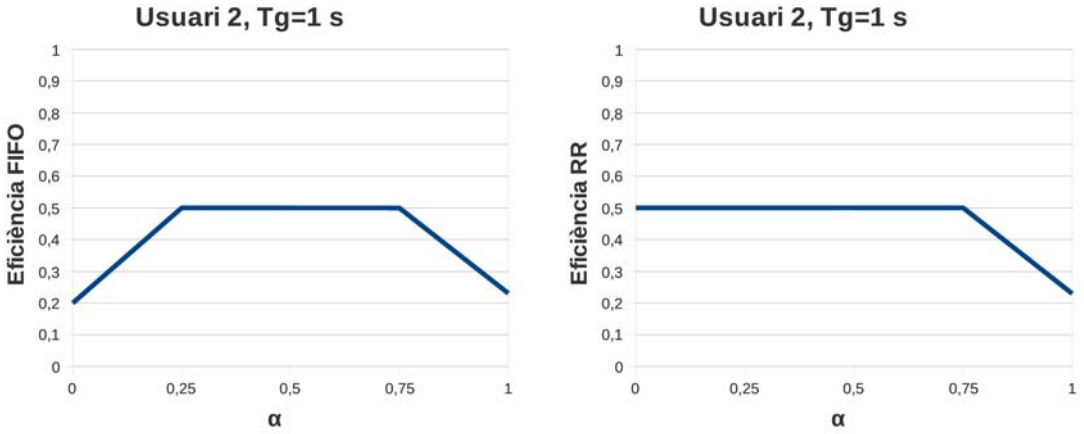


a)Eficiència FIFO usuari 1 *b)Eficiència RR usuari 1*

Figura 8.5: *Eficiència* , $T_{mig-generacio} = 0,1s$

A la figura 8.6 tenim ja per als dos casos FIFO i RR una eficiència de 0,5. L'eficiència FIFO es manté constant entre 0,25 i 0,75, mentre que per a RR es manté constant des de

0 a 0,75. Per als dos casos, l'eficiència disminueix molt als extrems, quan un usuari fa el màxim de peticions que pot fer. Als extrems, tindrem un throughput alt, però limitarem molt l'eficiència ja que al generar-se tantes peticions, també se'n perdran moltes.

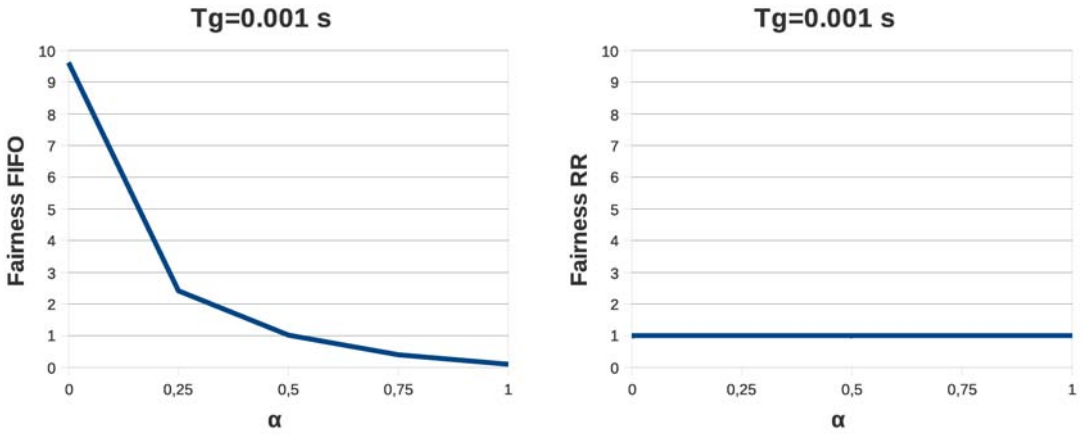


a)Eficiència FIFO usuari 2 *b)Eficiència RR usuari 2*

Figura 8.6: *Eficiència*, $T_{mig-generacio} = 1s$

Finalment, l'altre mesura a estudiar serà el **Fairness**. El fairness, és una mesura que ens permet veure com diferents usuaris estan compartint els recursos d'un sistema. En el nostre cas, el càlcul del fairness el determinarem en funció dels paquets que transmeti cada usuari (8.3).

$$Fairness = \frac{Throughput\ usuari1}{Throughpupt\ usuari2} \tag{8.3}$$

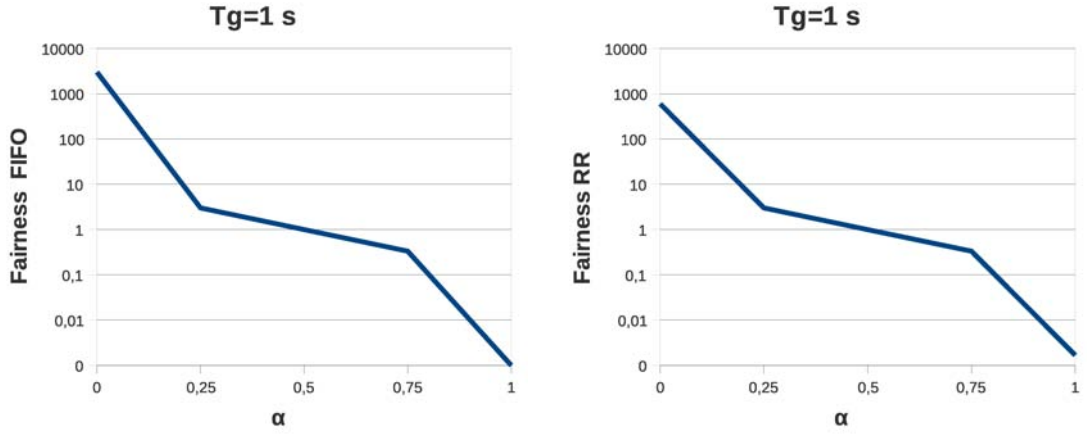


a)Fairness FIFO *b)Fairness RR*

Figura 8.7: *Fairness*, $T_{mig-generacio} = 0,001s$

A la figura 8.7, veiem que, amb la tècnica RR, el fairness es queda estancat al 1, mentre que per a FIFO arriba fins a 10, però a partir de $\alpha = 0,5$ està per sota del 1. Com ja hem anat veient, per a temps de generació petits, i sobretot amb RR, on es reparteixen els recursos amb igualtat de condicions, aconseguim que sense tenir en compte el valor de α, cada usuari pugui transmetre pràcticament el mateix número de paquets.

Al augmentar el $T_{mig-generacio}$, el fairness del dos tipus d'operació cada vegada s'assemblen més, ja que encara que a RR s'intenti repartir equitativament, amb temps de generació alts, no seria eficient esperar cada vegada a l'usuari més lent. A la figura 8.8 veiem com el fairness dels dos casos són molt semblants, encara que per a RR no arriba a valors tan alts.



a)Fairness FIFO *b)Fairness RR*

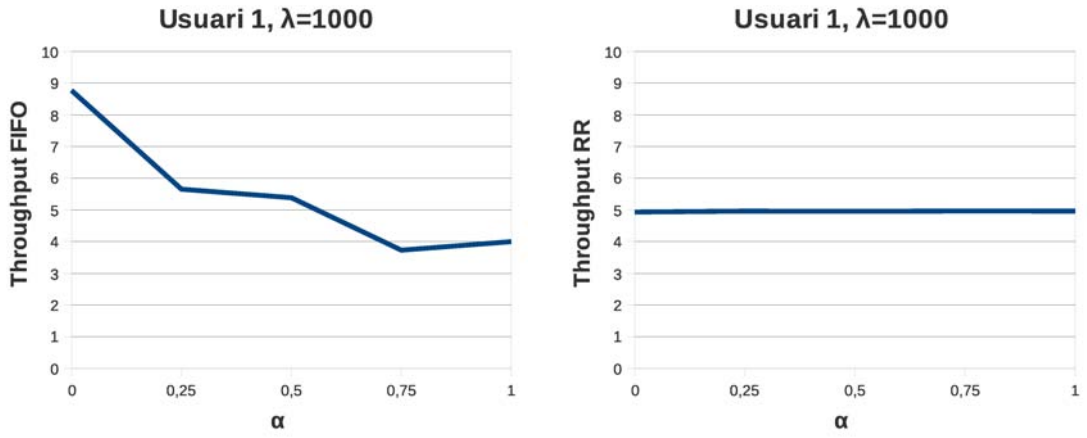
Figura 8.8: *Fairness*, $T_{mig-generacio} = 1s$

8.2 Temps de generació aleatoris

Per a les simulacions amb aquest sistema de generació de paquets, s'ha seguit la mateixa metodologia que a l'apartat anterior. Un cop generada la variable aleatòria que definirà el temps de generació, es variarà aquest temps en funció de α.

Encara que es generin temps aleatoris, els resultats obtinguts en aquest cas són molt semblants als vistos amb les simulacions de temps fix. El problema és degut a que al final de les simulacions pràcticament no s'aprecia l'efecte d'usar temps de generació aleatoris. Per tant, per intentar observar l'efecte causat per els temps aleatoris, reduirem el temps de les simulacions.

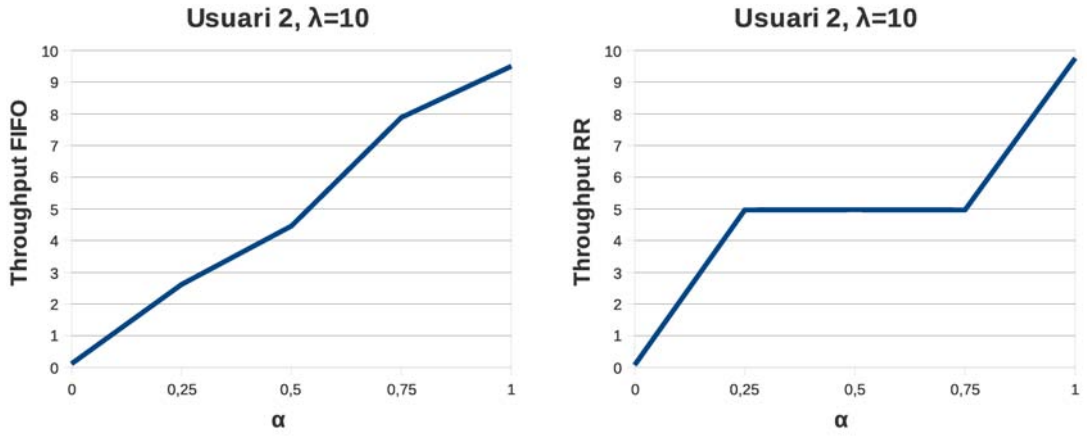
De les mesures vistes a l'apartat anterior, aquí només veurem el throughput, ja que els resultats obtinguts amb les demés mesures són pràcticament iguals.



a)Throughput FIFO usuari 1 *b)Throughput RR usuari 1*

Figura 8.9: *Throughput*, $\lambda = 1000$

A la figura 8.9 s'hi pot veure el throughput de l'usuari 1 per al cas de $\lambda = 1000$. Amb aquest valor de λ obtenim un temps de generació mig de 0,001 segons. Aquest cas, al ser en el que obtenim els temps més petits, serà en el que veiem més canvis envers als temps de generació fix. Com es pot veure, el throughput FIFO deixa de ser tan lineal com era amb el temps de generació fix. Aquest comportament és degut a que la diferència entre el temps de generació i el de transmissió es molt gran i fa que les diferències provocades per els temps aleatoris es notin més. Al throughput RR en canvi, segueix igual.

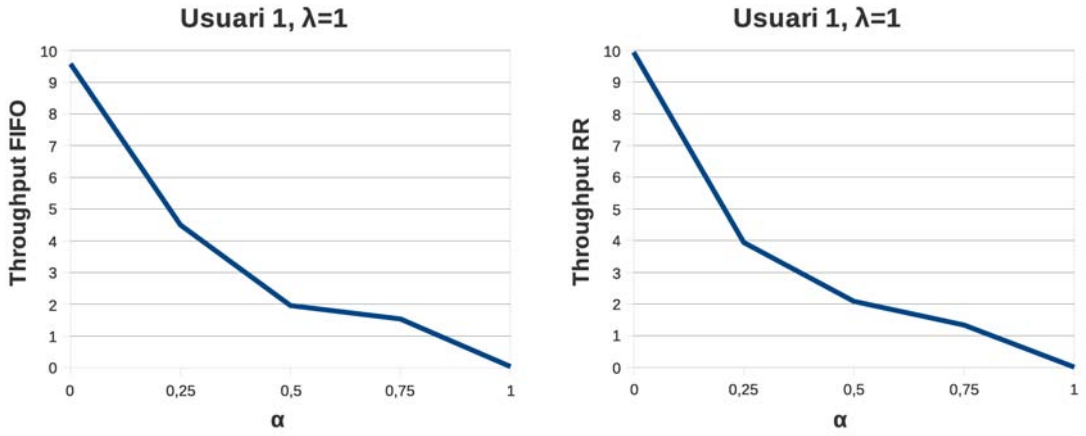


a)Throughput FIFO usuari 2 *b)Throughput RR usuari 2*

Figura 8.10: *Throughput*, $\lambda = 10$

A mesura que disminuïm el valor de λ, és a dir, augmentem el temps de generació mig, els efectes dels temps de generació aleatoris cada cop són menys presents. A la figura 8.10 per exemple tenim el cas de $\lambda = 10$ per tant el temps mig de generació serà

igual al de transmissió. En aquest cas el throughput FIFO comença a ser més lineal, i igual que abans, a RR segueix igual.



a) Throughput FIFO usuari 1 *b) Throughput RR usuari 1*

Figura 8.11: *Throughput*, $\lambda = 1$

Finalment, per a $\lambda = 1$ al obtenir temps de generació bastant més grans que el temps de transmissió, pràcticament ja no s'aprecien els efectes dels temps aleatoris. A la figura 8.11 podem veure els resultats obtinguts, i tan per FIFO com per RR no s'observen canvis comparant amb els temps de generació fix.

Capítol 9

Conclusions i treballs futurs

Arribats a aquest punt, farem un resum del treball fet al llarg de tot el projecte. La tasca inicial d'aquest projecte es va caracteritzar per buscar una tarja wireless LAN adequada. A part de ser una tarja que tingués bon rendiment, el que necessitàvem era que s'adaptés al màxim a les necessitats del projecte. Un cop escollida la tarja Atheros, es va procedir a instal·lar als dos ordinadors del laboratori el driver MadWifi. D'aquesta manera teníem dos ordinadors funcionant amb les tarjes Atheros i la xarxa es completaria amb un tercer ordinador per a poder implementar-hi el programa amb els conceptes de network coding.

Després de veure les possibilitats que ens oferien les tarjes , amb el driver i la seva característica principal, la creació d'interfícies virtuals, va sorgir l'idea de canviar la distribució de la xarxa inicial. La nova xarxa va passar a tenir dos nodes, en la que un dels nodes estava dividit en dos usuaris simulant dos ordinadors diferents, i l'altre node era l'encarregat de rebre les dades, combinar-les i retransmetre-les.

Fins arribar a aquest punt, totes les tasques del projecte es van realitzar conjuntament entre els dos membres del projecte. Des d'escollir la tarja i instal·lar el driver adequat fins a la recerca de possibilitats i modificacions de la xarxa que ens ofereix LINUX. A partir d'aquí doncs es van separar clarament les tasques, un membre es centraria en el funcionament i tasques que realitza el node encarregat de codificar els missatges i retransmetre'ls, mentres que l'altre es dedicaria a l'implementació dels nodes virtuals.

Un cop separades les tasques, es va començar a investigar quina era la millor manera d'implementar un programa que pogués gestionar la transmissió i recepció de dades de les dos interfícies virtuals simultàniament . Es va arribar a la conclusió de que la millor manera d'operar amb aquesta serie de processos simultanis era l'us de threads.

Es va procedir a fer el programa que implementés la virtualització dels nodes amb threads. Va ser aquí on vam veure que de la mateixa manera que Madwifi crea diferents interfícies virtuals, amb l'ajut dels threads, podíem implementar diferents processos que simulessin dos usuaris transmetent i rebent dades.

Finalment, ens vam centrar en el disseny dels diferents processos necessaris per a dur a terme la virtualització. Com que tan jo com la meva companya vam anar aprofundint en les diferents tasques ja esmentades, es van treballar parts del demostrador, com la recepció i descodificació de paquets, que va faltar per comprovar el seu funcionament.

9.1 Conclusions

Del treball realitzat al llarg d'aquest projecte, i tenint en compte els objectius proposats al principi, podem extreure les següents conclusions:

- Amb les tarjes Atheros i MadWifi, obtenim un alt rendiment per dur a terme estudis i investigacions a xarxes wireless; a més ens ofereixen característiques com la virtualització, que altres tarjes i drivers no proporcionen.
- Gràcies a les comandes, utilitats i programes que incorpora LINUX hem aconseguit manipular i modificar diferents paràmetres de la xarxa, amb un alt grau de llibertat, que ens han facilitat les tasques a l'hora de desenvolupar el projecte.
- L'estudi de les diferent tècniques de transmissió de dades que proporcionen els sockets, ens ha obert un gran ventall de possibilitats a l'hora de realitzar la comunicació entre els dos programes de la manera desitjada.
- La programació concurrent amb threads ens ha permès implementar un programa en el que varis processos interactuen i intercanvien dades simultàniament sense cap tipus de problema.
- Conèixer les tècniques bàsiques de virtualització ens a ajudat a implementar un programa que ens permet simular la virtualització d'una interfície, on els usuaris virtuals son capaços de transmetre i rebre paquets independentment com si de dos ordinadors diferents es tractés.
- A l'entorn de virtualització, escollir un bon temps transmissió en funció del temps de generació és una decisió molt important. Per a temps de generació més petits que el temps de transmissió i petits en general, aconseguirem que es transmetin una gran quantitat de paquets. Encara que, transmetre tans paquets, i haver de compartir l'interfície entre els dos usuaris, farà que es perdin molts paquets. Aquests temps de generació són ideals per *streaming* de veu o vídeo on no importa tant si es perden paquets sinó la velocitat i quantitat de paquets rebuts.

  Per a temps de generació mes grans que el temps de transmissió, i considerablement grans, evidentment es transmetran molts menys paquets. Al tenir un temps de generació més gran que el de transmissió, el que fem es limitar el transmissor, ja que molts cops no tindrà paquets per enviar. D'aquesta manera, encara que

tinguem una velocitat baixa, es perdran molt pocs paquets. Aquest cas serà el més recomanable per a la transmissió de fitxers, on és molt important no perdre informació.

- Serà determinant també l'elecció de la tècnica d'organització de les cues del virtualitzador, FIFO o RR. Amb l'elecció d'aquesta tècnica, podrem canviar la manera amb que els usuaris comparteixen l'interfície física. Gràcies a aquestes tècniques, si tenim un usuari que realitza transmissions més prioritàries que l'altre, amb FIFO podem assignar-li un temps de generació més petit que l'altre usuari, i tindrà pràcticament tot el temps l'interfície per transmetre. Amb RR en canvi, com que sempre es mira de repartir al màxim possible, encara que tinguem temps petits, els dos usuaris transmetran un número similar de paquets. Aquesta tècnica serà molt adequada si necessitem que els dos usuaris transmetin en igualtat de condicions.

  A més a més, la política RR també es eficient a l'hora de compartir l'interfície per al cas de la generació de temps aleatoris. Amb l'ús d'aquesta tècnica amb temps aleatoris amb distribució exponencial, aconseguim que els dos usuaris transmetin en igualtat de condicions, obtenint uns resultats pràcticament idèntics que als obtinguts amb temps de generació fixos.

- El principal avantatge que obtenim de l'ús de la virtualització és la comoditat que suposa treballar amb diferents interfícies virtuals a l'hora, ja que ens estalviem d'utilitzar altres ordinadors per realitzar operacions simultànies, com és el cas d'aquest projecte.

  També cal tenir en compte que aplicant la virtualització es pot fer un ús més eficient dels recursos que ens proporciona la interfície de xarxa.

- Un dels inconvenients que podem considerar de la tècnica de virtualització d'interfícies, és que, evidentment, un equip treballant amb varis usuaris virtuals paral·lelament, sempre consumirà més recursos i temps que un on tots els recursos de memòria i temps estan a la disposició d'un sol usuari.

9.2 Treballs futurs

Com ja hem anat veient al llarg de la memòria s'ha implementat un demostrador wireless amb la tècnica network coding i profunditzant en la virtualització. Com dèiem abans, han quedat parts i tasques del demostrador que falta acabar de comprovar i que es poden realitzar a partir d'aquest projecte.

- El principal treball que ha quedat pendent és l'unió dels programes implementats pels dos membres del projecte. I un cop feta l'unió del programa final, realitzar les proves, simulacions, i estudis necessaris per comprovar els avantatges d'aquesta tècnica.

- En la part de virtualització, una tasca que es podria realitzar, és ampliar el nombre d'usuaris virtuals que comparteixen l'interfície física i estudiar el nou comportament.
- Comparar els resultats obtinguts amb el nostre demostrador i la virtualització que fa MadWifi.
- També seria una bona opció posar en pràctica altres tècniques de virtualització que no s'han dut a terme en aquest projecte.
- Una altra opció molt interessant podria ser la de adaptar el programa per a poder realitzar simulacions amb tràfic de dades real com *VoIP* o *streaming* de vídeo.

Bibliografia

[1] `http://madwifi-project.org/`.

[2] LINUX sockets.
`http://www.chuidiang.com/clinux/sockets/sockets_simp.php`.

[3] `http://www.cdg.org/technology/3g.asp`.

[4] David R. Butenhof. *Programming with POSIX Threads*. Segona edició. 1997.

[5] GENI: Global Environment for Network Innovations. Technical document on wireless virtualization. September 2006.

[6] Behrouz A. Forouzan. *Transmisión De Datos y Redes De Computadores*. Segona edició. 2002.

[7] Victor Moreno i Kumar Reddy. *Network Virtualization*. Primera edició. 2006.

[8] N. Cai S.-Y. R. Li and R. W. Yeung R. Ahlswede. Network information flow. *IEEE Trans. Inform. Theory*, 46, July 2000.

[9] S. Perez, Eder Miguel, and J.M. Caballero. Virtualization of the wireless medium: a simulation-based study. 2009.

Resum:

El treball realitzat en aquest projecte, es basa en l'implementació d'un demostrador wireless, i més específicament l'estudi de les tècniques network coding i virtualització.

Network coding és un nou mètode de transmissió de dades que es basa en la codificació de paquets per incrementar el rendiment fins ara obtingut als mètodes de transmissió convencionals. La virtualització és una tècnica que consisteix en compartir de forma més eficient els recursos d'un sistema. En el nostre cas s'utilitzarà la virtualització per dividir una interfície sense fils en diferents usuaris virtuals transmetent i rebent dades simultàniament.

L'objectiu del projecte és realitzar un seguit de proves i estudis per veure els avantatges d'aquestes dues tècniques.

Resumen:

El trabajo realizado en este proyecto, se basa en la implementación de un demostrador wireless, i más específicamente en el estudio de las técnicas network coding i virtualización.

Network coding es un nuevo método de transmisión de datos que se basa en la codificación de paquetes para incrementar el rendimiento hasta ahora obtenido con los métodos de transmisión convencionales. La virtualización es una técnica que consiste en compartir de forma más eficiente los recursos de un sistema. En nuestro caso se utilizará la virtualización para dividir una interfaz inalámbrica en diferentes usuarios virtuales transmitiendo i recibiendo datos simultáneamente.

El objetivo del proyecto es realizar una serie de pruebas i estudios para observar las ventajas de estas dos técnicas.

Summary:

The work on this project, is based on the implementation of a wireless demonstrator , and more specifically in the study of techniques Network coding and virtualization.

Network coding is a new method of data transmission based on the coding of packages to improve performance so far obtained with conventional transmission methods. Virtualization is a technique that shares the resources of a system more efficiently. In our case we will use virtualization to divide a wireless interface into different virtual users to transmit and receive data simultaneously.

The project aims to conduct a series of studies to test and see the benefits of these two techniques.