Programació de serveis i processos

CFGS.DAM.M09/0.13

Desenvolupament d'aplicacions multiplataforma

Generalitat de Catalunya
Departament d'Educació

ioc
institut obert de catalunya

Aquesta col·lecció ha estat dissenyada i coordinada des de l'Institut Obert de Catalunya.

Coordinació de continguts
Josep Cañellas Bornas

Redacció de continguts
Joan Arnedo Moreno, Josep Cañellas Bornas i José Antonio Leo Megías

Agraïments
Als editors, correctors i col·laboradors que han fet possible que aquest projecte vegi la llum. Gràcies per la paciència.

Primera edició: setembre 2014
© Departament d'Ensenyament

Dipòsit legal: DL B 13814-2015

Llicenciat Creative Commons BY-NC-SA. (Reconeixement-No comercial-Compartir amb la mateixa llicència 3.0 Espanya).

Podeu veure el text legal complet a

http://creativecommons.org/licenses/by-nc-sa/3.0/es/legalcode.ca

Introducció

Les noves tendències tecnològiques situen Internet al bell mig de qualsevol avenç. Així, cada cop sentim més a parlar de l'Internet de les coses, de l'Internet de les persones o de l'Internet dels serveis per denotar l'enorme presència prevista d'aquest sistema en la nostra vida quotidiana.

Quines repercussions tindran aquests canvis en el desenvolupament de la nova generació d'aplicacions adreçades a l'Internet de les coses, els serveis i les persones? Sembla clar que els programadors hauran de tenir un gran coneixement de com funciona la xarxa, com es comuniquen el programes i com col·laboren entre sí.

També caldrà que sàpiguen com es poden optimitzar les aplicacions per tal que funcionin àgilment, amb independència del nombre d'usuaris que les facin servir. A Internet tot es magnifica. La demanda que una aplicació de la xarxa pot suportar avui dia ha crescut exponencialment i de moment la tendència segueix a l'alça. Serà important, doncs, desenvolupar programes capaços de gestionar múltiples ordres a la vegada i de treballar amb un nombre molt elevat de dades sense que el seu rendiment en resulti afectat.

Per descomptat, una altra peça important en l'Internet de les coses és la seguretat, perquè cada cop hi haurà més dades circulant per la xarxa i això suposa un risc. El nombre de delictes informàtics incrementa de forma similar a l'increment d'usuaris i a la informació que circula. Cal assegurar que les dades sensibles siguin confidencials i no puguin anar a parar a mans desconegudes. És per això que el desenvolupador informàtic haurà de contemplar la seguretat informàtica com un component rellevant en qualsevol aplicació futura.

El cicle formatiu de Desenvolupament d'Aplicacions Multiplataforma compacta tots aquest coneixements en el present mòdul, anomenat *Programació de serveis i processos*. El mòdul s'estructura en 3 unitats. La unitat anomenada "Processos i fils" ofereix una visió del que anomenem programació multifil, que està orientada a garantir l'execució de diverses ordres a la vegada treballant amb les mateixes dades de forma sincronitzada. En aquest mòdul aprendreu tècniques avançades de sincronització i procés paral·lel que us ajudaran a resoldre problemes complexos i de gran volum de forma eficient i robusta.

La unitat anomenada "Sòcols i serveis" centra la temàtica en les aplicacions distribuïdes i el desenvolupament de serveis d'Internet. Hi aprendreu com funcionen les aplicacions d'Internet, així com els principals protocols de comunicació que permeten el seu funcionament. També hi trobareu les claus per desenvolupar aplicacions que funcionin de forma distribuïda.

La unitat anomenada "Seguretat i criptografia" us donarà les claus de les principals tècniques criptogràfiques i us mostrarà com aplicar-les al servei de la seguretat informàtica, específicament a la seguretat dins la xarxa. Aquí aprendrem a identificar els usuaris que volen accedir a les aplicacions, a protegir les seves dades,

fer-ne enviaments segurs o evitar les suplantacions d'identitats, entre d'altres.

Resultats d'aprenentatge

En finalitzar aquest mòdul, l'estudiant:

Processos i fils

1. Desenvolupa aplicacions compostes per diversos processos reconeixent i aplicant principis de programació paral·lela.
2. Desenvolupa aplicacions compostes per diversos fils d'execució analitzant i aplicant llibreries específiques del llenguatge de programació.

Sòcols i serveis

1. Programa mecanismes de comunicació en xarxa emprant sòcols i analitzant l'escenari d'execució.
2. Desenvolupa aplicacions que ofereixin serveis en xarxa, utilitzant llibreries de classes i aplicant criteris d'eficiència i disponibilitat.

Seguretat i criptografia

1. Protegeix les aplicacions i les dades definint i aplicant criteris de seguretat en l'accés, emmagatzematge i transmissió de la informació.

Continguts

Processos i fils

Unitat 1

Processos i fils

1. Programació multiprocés i paral·lela
2. Programació multifil

Sòcols i serveis

Unitat 2

Sòcols i serveis

1. Programació de comunicacions en xarxa
2. Generació de serveis en xarxa

Seguretat i criptografia

Unitat 3

Seguretat i criptografia

1. Criptografia i Java
2. Aplicacions Java segures a Internet

Processos i fils

José Antonio Leo Megías

Índex

Introducció 5

Resultats d'aprenentatge 7

1 Programació multiprocés i paral·lela 9
1.1 Programació multiprocés . . . 9
1.1.1 Procés, sistemes monoprocessador-multiprocessador . . . 9
1.1.2 Programació concurrent, paral·lela i distribuïda . . . 13
1.1.3 Avantatges i desavantatges del multiprocés . . . 16
1.2 Processos i serveis . . . 16
1.2.1 Fils i processos . . . 18
1.3 Gestió de processos i desenvolupament d'aplicacions amb finalitat de computació paral·lela . . . 20
1.3.1 Estats d'un procés . . . 20
1.3.2 Planificació de processos . . . 22
1.3.3 Programació paral·lela transparent . . . 23
1.4 Sincronització i comunicació entre processos . . . 33
1.4.1 Competència de recursos i sincronització . . . 33
1.4.2 Sincronització i comunicació . . . 34
1.4.3 Solucions a problemes de sincronització i comunicació . . . 39

2 Programació multifil 53
2.1 Els fils . . . 53
2.1.1 Definició de fil . . . 53
2.1.2 Relació procés i fil . . . 54
2.1.3 El fils a Java . . . 55
2.2 Gestió de fils . . . 59
2.2.1 Creació i execució d'un fil . . . 59
2.2.2 Estats i canvis d'estat d'un fil . . . 62
2.2.3 Mètode thread.join() . . . 64
2.3 Sincronització de fils . . . 67
2.3.1 Mètodes synchronized . . . 68
2.3.2 Objectes synchronized . . . 73
2.3.3 Sincronitzar mètodes estàtics . . . 74
2.3.4 Variables volatile . . . 75
2.3.5 Interbloqueig . . . 76
2.4 Compartir informació entre fils . . . 78
2.4.1 Mètodes wait(), notify() i notifyAll() . . . 79
2.4.2 Productor-consumidor . . . 82
2.5 Agrupació de fils . . . 86
2.5.1 Agrupació per defecte i creació de grups . . . 86
2.5.2 Classe ThreadGroup . . . 88
2.6 Gestió de fils per part del sistema operatiu . . . 89
2.6.1 Planificació de fils . . . 89

2.6.2 Prioritat de fils 90
2.6.3 Fils dimonis 91
2.7 Documentació d'aplicacions i depuració d'aplicacions multifils 93
2.7.1 Depuració d'aplicacions multifils 94

Introducció

Contínuament veiem persones i màquines fent accions de forma simultàniament. Els ordinadors intenten simular aquesta realitat, on moltes coses passen a la vegada.

Un dels avantatges de realitzar diverses coses a la vegada és que es podem acabar més ràpid. Aquest seria el paradigma que s'intenta seguir quan a un ordinador executen diverses tasques alhora.

El multiprocés, ha estat molt lligat al sistema operatiu. De fet, els primers programes concurrents van ser els sistemes operatius, on ordinadors amb un únic processador en el sistema operatiu s'encarregava de repartir el temps del processador entre les diferents tasques.

Amb l'avanç i l'accessibilitat de la tecnologia, els ordinadors s'estan dotant de més d'un processador i per tant la programació en paral·lel és un fet. Al millorar les comunicacions també és més viable implementar sistemes distribuïts, on cada ordinador executa part d'una aplicació.

Durant els darrers temps, la programació concurrent s'està implementant a una gran part dels sistemes informàtics. L'aparició del fil o *thread* i de llenguatges de programació que donen suport a la programació concurrent amb primitives pròpies del llenguatge, com Java, han fet que l'execució d'una aplicació sigui més ràpida i una mica més simple de programar. Malgrat tot la programació multifil, on diversos fils estan en execució continua sent una mica més complexa.

Un altre element que ha propiciat el desenvolupament d'aplicacions concurrents és Internet, ja que qualsevol aplicació que executem sobre Internet (programes de missatgeria, navegadors, etc.) estarà lligada a la programació concurrent.

La unitat està dividida en dues activitats, la primera "Programació multiprocés" veurem les característiques de la programació concurrent, paral·lela i distribuïda, veurem que són els processos, quins són els seus estats, els problemes i les solucions que es plantegen quan s'apliquen tècniques de concurrència, com ara la comunicació i la sincronització.

La segona unitat "programació multifil", veurem com s'implementen aplicacions amb diversos fils en execució. Veurem els diferents estats d'un fil, els problemes que pot plantejar la seva programació i com es poden solucionar.

És molt important seguir la teoria i entendre els exemples que es proposen, anar fent els exercicis d'autoavaluació, i realitzar les activitats per a una bona compressió de la unitat.

Resultats d'aprenentatge

1. Desenvolupa aplicacions compostes per diversos processos reconeixent i aplicant principis de programació paral·lela.

- Reconeix les característiques de la programació concurrent i els seus àmbits d'aplicació.
- Identifica les diferències entre programació paral·lela i programació distribuïda, els seus avantatges i inconvenients.
- Analitza les característiques dels processos i de la seva execució a càrrec del sistema operatiu.
- Caracteritza els fils d'execució i descriu la seva relació amb els processos.
- Utilitza classes per programar aplicacions que creïn subprocessos.
- Utilitza mecanismes per sincronitzar i obtenir el valor retornat per als subprocessos iniciats.
- Desenvolupa aplicacions que gestionin i utilitzin processos per a l'execució de diverses tasques en paral·lel.
- Depura i documenta les aplicacions desenvolupades.

2. Desenvolupa aplicacions compostes per diversos fils d'execució analitzant i aplicant llibreries específiques del llenguatge de programació.

- Identifica situacions en què sigui útil la utilització de diversos fils en un programa.
- Reconeix els mecanismes per a crear, iniciar i finalitzar fils.
- Programa aplicacions que implementen diversos fils.
- Identifica els possibles estats d'execució d'un fil i programa aplicacions que els gestionen.
- Utilitza mecanismes per compartir informació entre diversos fils d'un mateix procés.
- Desenvolupa programes formats per diversos fils sincronitzats mitjançant tècniques específiques.
- Estableix i controla la prioritat de cadascun dels fils d'execució.
- Depura i documenta els programes desenvolupats.

1. Programació multiprocés i paral·lela

El dia a dia demana cada cop més potència de càlcul a les aplicacions informàtiques, més i més càlculs numèrics en enginyeria, ciència, astrofísica, meteorologia... Volem que els ordinadors reaccionin amb un raonament humà. Així tenim ordinadors que juguen partides d'escacs, que parlen com humans o que prenen decisions a partir de dades poc precises...

Malgrat que no és l'únic factor determinant, els processadors juguen un paper fonamental a l'hora d'assolir aquesta potència. Cada cop són més ràpids i eficients i els sistemes operatius permeten coordinar diversos processadors per incrementar el nombre de càlculs per unitat de temps. L'ús de diversos processadors dins d'un sistema informàtic s'anomena multiprocés. Ara bé, la coordinació de diversos processadors pot provocar situacions d'error que haurem d'evitar.

1.1 Programació multiprocés

Un cuiner que tingui intenció de fer un plat molt elaborat, amb salses i guarnicions diferents, probablement farà servir diferents paelles per cuinar cada element que necessiti utilitzar per elaborar el plat. Tindrà una paella amb la salsa, una altra preparant la guarnició i en una altra cuinarà el plat principal. Tot això ho farà alhora i, finalment, quan acabi totes aquestes tasques les ajuntarà en un únic plat.

Aquesta analogia ens serveix per introduir el terme **multiprocés**. El cuiner està executant diferents processos a la vegada: cuina la salsa, la guarnició i el plat principal, seguint un ordre d'execució per arribar a un resultat que esperava, un bon plat.

1.1.1 Procés, sistemes monoprocessador-multiprocessador

Seguint amb l'analogia del cuiner, un procés seria l'activitat realitzada pel cuiner (processador) d'elaborar algun dels complements o el plat principal (resultats) a partir d'una recepta (programa). Cada procés de creació requereix agafar els ingredients (les dades), posar-los a la paella (recurs associat) i cuinar-los amb independència de la resta de processos.

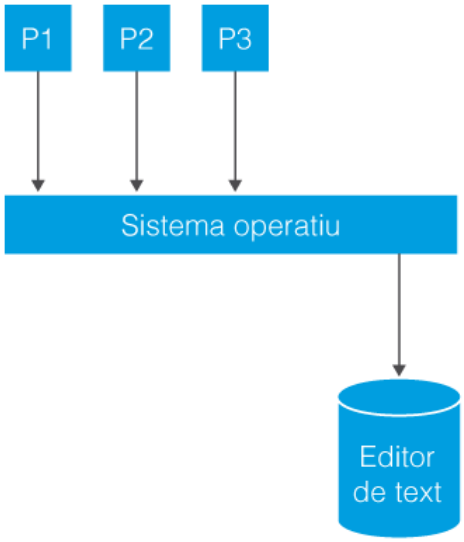
Un programa és un element estàtic, un conjunt d'instruccions, unes línies de codi escrites en un llenguatge de programació, que descriuen el tractament que cal donar a unes dades inicials (d'entrada) per aconseguir el resultat esperat (una sortida concreta). En canvi, **un procés** és dinàmic, és una instància d'un programa en execució, que realitza els canvis indicats pel programa a les dades inicials i obté una sortida concreta. El procés, a més de les instruccions, requerirà també de recursos específics per a l'execució com ara el comptador d'instruccions del programa, el contingut dels registres o les dades.

El sistema operatiu és l'encarregat de gestionar els processos.

El sistema operatiu és l'encarregat de la gestió de processos. Els crea, els elimina i els proveeix d'instruments que en permetin l'execució i també la comunicació entre ells. Quan s'executa un programa, el sistema operatiu crea una instància del programa: el procés. Si el programa es tornés a executar es crearia una nova instància totalment independent a l'anterior (un nou procés) amb les seves pròpies variables, piles i registres.

Imaginem un servidor d'aplicacions en el qual tenim instal·lat un programa d'edició de text. Posem per cas que existeixen diversos usuaris que volen escriure els seus textos executant l'editor. Cada instància del programa és un procés totalment independent als altres. Cada procés té unes dades d'entrada i per tant diferents sortides. Cada usuari escriurà a la seva execució de l'editor (el procés), el seu text. A la figura 1.1 es reflecteix aquest exemple.

Figura 1.1. Execució de processos



Quan un procés es troba en execució es troba completament en memòria i té assignats els recursos que necessita. Un procés no pot escriure en zones de memòria assignada a altres processos, la memòria no és compartida. Cada procés té una estructura de dades en la qual es guarda la informació associada a l'execució del procés. Aquesta zona s'anomena Bloc de Control de Procés (BCP). Els processos competeixen amb altres processos executats a la vegada pels recursos

del sistema. Opcionalment el sistema operatiu permet també que els processos puguin comunicar-se i col·laborar entre ells.

Fins ara hem parlat d'elements de programari, com ara programes i processos, però no del maquinari utilitzat per executar-los. L'element de maquinari en el qual s'executen els processos és el processador. Un **processador** és el component de maquinari d'un sistema informàtic encarregat d'executar les instruccions i processar les dades. Quan un dispositiu informàtic està format per un únic processador parlarem de sistemes monoprocessadors, per contra, si està format per més d'un processador parlarem de sistemes multiprocessador.

Un **sistema monoprocessador** és aquell que està format únicament per un processador.
Un **sistema multiprocessador** està format per més d'un processador.

Actualment, la majoria dels sistemes operatius aprofiten els temps de repòs dels processos, quan esperen, per exemple, alguna dada de l'usuari o es troben pendents d'alguna operació d'entrada/sortida, per introduir al processador un altre procés, simulant així una execució paral·lela.

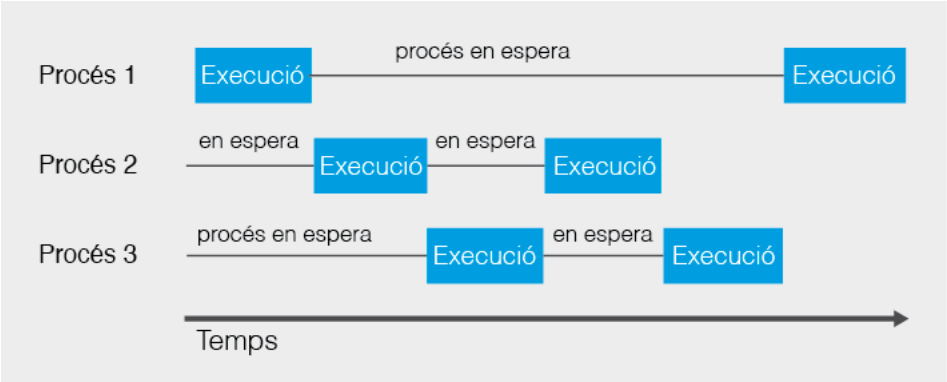
Placa Base Multiprocessador

De forma genèrica anomenarem els processos que s'executin a la vegada, ja sigui de forma real o simulada, **processos concurrents**. L'execució de processos concurrents, encara que sigui de forma simulada, fa augmentar el rendiment del sistema informàtic ja que aprofita més el temps del processador. És el sistema operatiu l'encarregat de gestionar l'execució concurrent de diferents processos contra un mateix processador i sovint ens hi referim amb el nom de **multiprogramació**.

Parlem de **multiprogramació** quan el sistema operatiu gestiona l'execució de processos concurrents a un sistema monoprocessador.

La figura 1.2 correspon a una imatge de processos concurrents en la qual hi ha tres processos en execució i es van intercalant repartint-se el temps del processador.

Figura 1.2. Execució de processos concurrents



El sistema operatiu Windows 95 únicament permet treballar amb un únic processador. Els sistemes operatius a partir de Win2000/NT i Linux/Unix són multiprocés, poden treballar amb més d'un processador

Els sistemes informàtics monoprocessadors actuals tenen una gran velocitat i si han d'executar més d'un procés a la vegada aniran alternant la seva execució simulant un multiprocés. Quan en un ordinador monoprocessador s'apliquen tècniques de multiprogramació els beneficis en rendiment no són tan evidents com en sistemes informàtics multiprocessadors.

Processador Intel amb dos nuclis

En un sistema informàtic **multiprocessador** existeixen dos o més processadors, per tant es poden executar simultàniament diversos processos. També es poden qualificar de multiprocessadors aquells dispositius el processador dels quals té més d'un nucli (*multicore*) , és a dir, tenen més d'una CPU al mateix circuit integrat del processador. Evidentment l'arquitectura és diferent, però aquest sistema, de la mateixa manera que un equip multiprocessador, ens permet executar de manera simultània processos diferents.

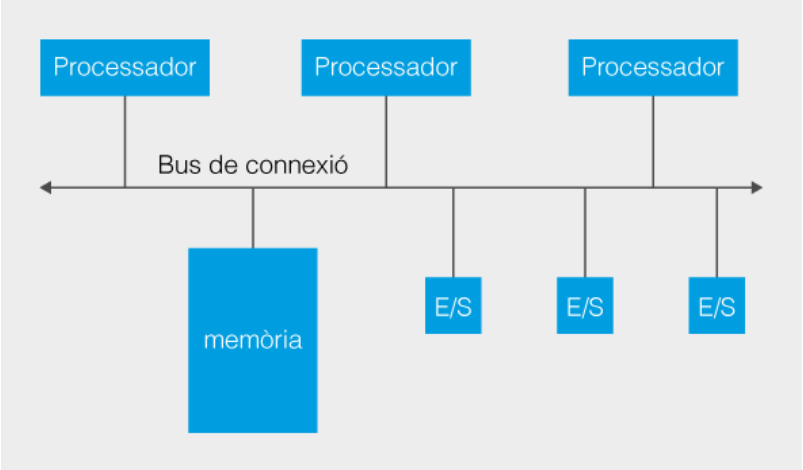
Els sistemes multiprocessadors es poden classificar depenent de la seva arquitectura en *sistemes multiprocessador fortament acoblats* i *sistemes multiprocessador dèbilment acoblats*.

Sistemes multiprocessadors fortament acoblats: en aquesta arquitectura els diferents processadors comparteixen una mateixa memòria i estan interconnectats a ella a través d'un bus. També poden tenir una petita memòria cau a cada processador. Hi ha una forta col·laboració entre processadors compartint variables a la memòria comuna a mode de resultats parcials o també per comunicar-se entre ells. Actualment, la majoria de sistemes operatius suporten aquest tipus de sistemes multiprocessador.

Els sistemes fortament acoblats depenen del pes que té cada processador a l'hora d'executar els processos. Es poden dividir en **sistemes multiprocés simètrics**, en els quals els processadors del sistema són de característiques similars i competeixen entre iguals per executar processos, i **sistemes multiprocés asimètrics** en els quals un dels processadors del sistema, el *màster*, controla la resta de processadors. Aquest sistema també s'anomena *master-slave*.

A la figura 1.3 es mostra gràficament com és un sistema multiprocessador fortament acoblat.

Figura 1.3. Sistema informàtic multiprocessador fortament acoblat

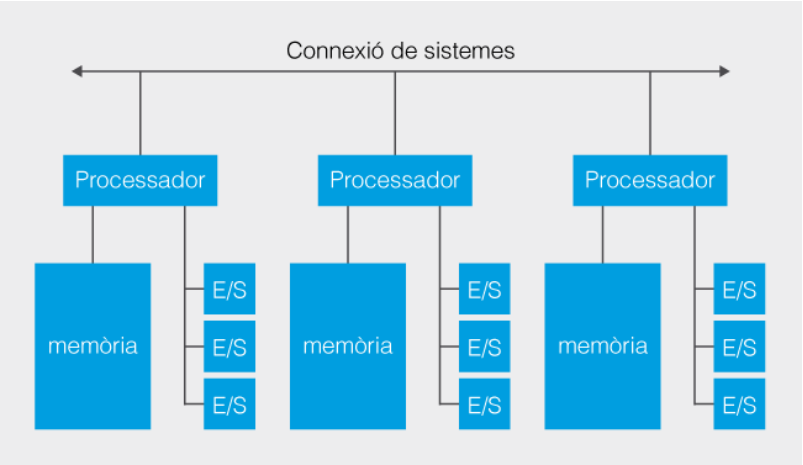


Sistemes multiprocessadors dèbilment acoblats: aquests sistemes no comparteixen memòria, cada processador té una memòria associada. A les execucions que necessiten col·laboració entre processos els cal l'intercanvi de missatges a través d'enllaços de comunicacions, per habilitar la comunicació entre processos. Un tipus d'aquests sistemes poc acoblats són els sistemes distribuïts, en els quals

cada dispositiu monoprocessador (o multiprocessador) podria estar situat a llocs físicament distants. Una xarxa d'ordinadors o Internet podria ser un exemple d'un sistema dèbilment acoblat distribuït.

A la figura 1.4 es mostra gràficament com és un sistema multiprocessador dèbilment acoblat.

Figura 1.4. Sistema informàtic multiprocessador dèbilment acoblat



1.1.2 Programació concurrent, paral·lela i distribuïda

Simula, creat al 1966 va ser el primer llenguatge de programació orientat a objectes concurrent.

Actualment, una gran quantitat d'aplicacions utilitzen la programació concurrent. En els sistemes operatius actuals existeixen moltes aplicacions executant-se al mateix temps i compartint informació. Per exemple, podem escriure en un editor de text, tenir posat un reproductor de música i descarregar un vídeo al mateix temps. També podem tenir en execució un navegador capaç de carregar en diverses pestanyes diferents pàgines web. Són exemples que il·lustren la idea de programació concurrent.

Gràcies a l'evolució que ha sofert el maquinari durant els últims anys s'han creat sistemes operatius que poden optimitzar els recursos dels processadors i poden executar diferents processos de forma simultània. L'execució simultània de processos s'anomena també **concurrència**. No vol dir que s'hagin d'executar exactament al mateix moment, l'intercalat de processos també es considera execució concurrent. La majoria dels llenguatges de programació actuals poden fer ús d'aquest recurs i dissenyar aplicacions en les quals els processos s'executin de manera concurrent.

Deep Blue. Supercomputador d'IBM amb 256 processadors treballant en paral·lel

> Parlem de **programació concurrent** quan s'executen en un dispositiu informàtic de forma simultània diferents tasques (processos).

L'execució concurrent pot comportar certs problemes a l'hora d'accedir a les dades que no trobarem mai a l'execució seqüencial i que caldrà tenir en compte. Bàsicament usarem dues tècniques per evitar-los: el boqueig i la comunicació.

Si la programació concurrent es dóna en un computador amb un únic processador parlarem de multiprogramació. En canvi, si el computador té més d'un processador i, per tant, els processos es poden executar de forma realment simultània, parlarem de programació paral·lela.

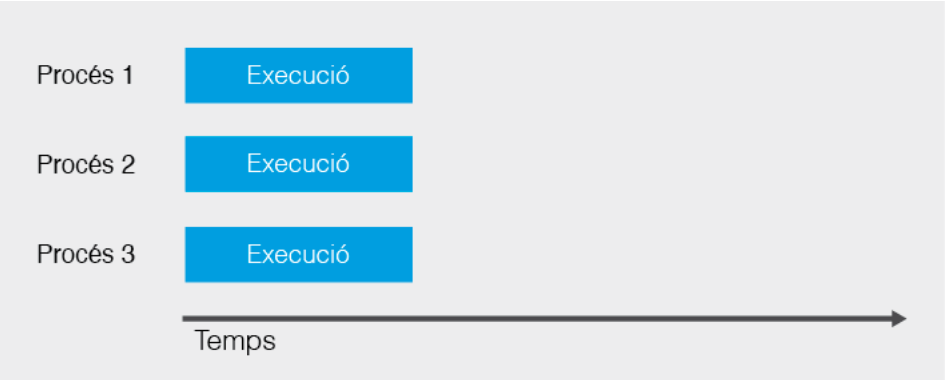
En un dispositiu multiprocessador la concurrència és real, els processos són executats de forma simultània en diferents processadors del sistema. És habitual que el número de processos que s'estigui executant de forma concurrent sigui major que el número de processadors. Per tant serà obligatori que alguns processos s'executin sobre el mateix processador.

Quan la programació concurrent es realitza en un sistema multiprocessador parlem de **programació paral·lela**.

En els ordinadors multiprocessadors la concurrència és real, per tant el temps d'execució acostuma a ser menor que en dispositius monoprocessadors.

A la figura 1.5 veiem un esquema del resultat d'una execució de tres processos en paral·lel en un ordinador multiprocessador amb tres processadors. En contraposició tenim la figura 1.2 amb l'execució en un ordinador monoprocessador.

Figura 1.5. Execució en paral·lel



El principal desavantatge de la programació paral·lela són els controls que hem d'afegir per tal que la comunicació i sincronització dels processos que s'executen concurrentment siguin correctes. Dos processos s'han de sincronitzar o comunicar quan un procés necessita alguna dada que està processant l'altre o bé un d'ells ha d'esperar la finalització de l'altre per poder continuar la seva execució.

El llenguatge de programació Java té integrat el suport a la concurrència en el propi llenguatge i no a través de llibreries.

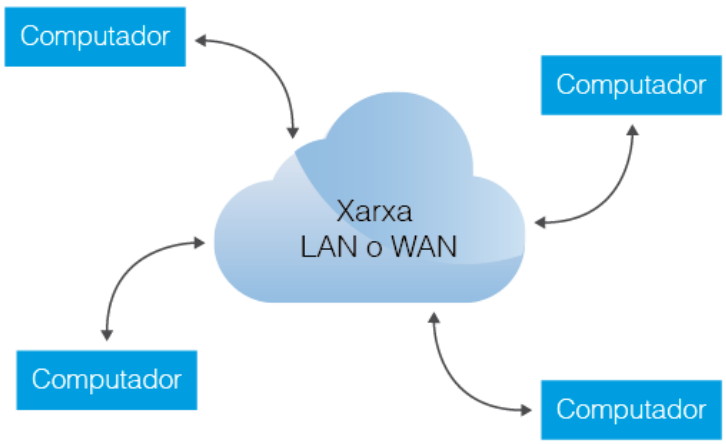
La programació paral·lela està molt lligada a l'arquitectura del sistema de computació. Però quan programem en llenguatges d'alt nivell ens hem d'abstreure de la plataforma en la qual s'executarà. Els llenguatges d'alt nivell ens proveeixen de llibreries que ens faciliten aquesta abstracció i ens permeten utilitzar les mateixes operacions per programar de forma paral·lela que acabaran implementades per usar-se un una plataforma concreta.

Un tipus especial de programació paral·lela és l'anomenada programació distribuïda. Aquesta programació es dóna en sistemes informàtics distribuïts. Un sistema distribuït està format per un conjunt d'ordinadors que poden estar situats en llocs

geogràfics diferents units entre ells a través d'una xarxa de comunicacions. Un exemple de sistema distribuït pot ser el d'un banc amb moltes oficines al món, amb un ordinador central per oficina per guardar els comptes locals i fer les transaccions locals. Aquest ordinador es pot comunicar amb els altres ordinadors centrals de la xarxa d'oficines. Quan es fa una transacció no importa on es troba la compte o el client.

En la figura 1.6 podem veure l'execució d'una programació distribuïda.

Figura 1.6. Execució de programació distribuïda



La **programació distribuïda** és un tipus de programació concurrent en la qual els processos són executats en una xarxa de processadors autònoms o en un sistema distribuït. És un sistema de computadors independents que des del punt de vista de l'usuari del sistema es veu com una sola computadora.

En els sistemes distribuïts, a diferència dels sistemes paral·lels, no existeix memòria compartida, per tant si necessiten la utilització de variables compartides es crearan rèpliques d'aquestes variables als diferents dispositius de la xarxa i caldrà controlar la coherència de les dades.

La comunicació entre processos per intercanviar dades o sincronitzar-se entre ells es fa amb missatges que s'envien a través de la xarxa de comunicacions que comparteixen.

Els principals avantatges són que es tracta de sistemes altament escalables i per tant reconfigurables fàcilment i d'alta disponibilitat. Com a desavantatges més importants, trobarem que són sistemes heterogenis, complexos de sincronitzar. Les comunicacions es fan a través de missatges que utilitzen la xarxa de comunicació compartida i això pot provocar-ne la saturació.

1.1.3 Avantatges i desavantatges del multiprocés

Hem definit programació concurrent o concurrència com la tècnica per la qual múltiples processos s'executen alhora i poden comunicar-se entre ells. La majoria dels sistemes intenten aprofitar aquesta concurrència per incrementar la capacitat d'execució.

Incrementar la potència de càlcul i el rendiment és un dels principals avantatges. Quan s'executen diversos processos a l'hora, la velocitat d'execució global es pot incrementar. Cal tenir en compte, però, que no sempre és així. A vegades, depenent de la complexitat de l'aplicació, les tècniques de sincronisme o comunicació són més costoses en temps que l'execució dels processos.

Un sistema multiprocessador és **flexible**, ja que, si augmenten el processos que s'estan executant és capaç de distribuir la càrrega de treball dels processadors, i pot també reassignar dinàmicament els recursos de memòria i els dispositius per ser més eficients. És de **fàcil creixement**. Si el sistema ho permet, es poden afegir nous processadors de forma senzilla i així augmenten la seva potència.

Els sistemes multiprocessador poden ser sistemes redundants. El fet de disposar de diversos processadors, pot permetre un augment de la disponibilitat dels recursos (més processadors al servei de l'usuari) o bé l'ús de processadors especialitzats en tasques de verificació i control. En el darrer cas parlarem de sistemes amb una alta **tolerància a fallades**. Una fallada d'un processador no fa que el sistema s'aturi.

Els sistemes multiprocés ens permeten diferenciar processos per la seva **especialització** i, per tant, reservar processadors per operacions complexes, aprofitar-ne d'altres per processaments paral·lels i per avançar l'execució.

Els inconvenients del multiprocessament vénen provocats sobretot pel control que s'ha de realitzar quan hi ha diversos processos en execució i han de compartir informació o comunicar-se. Això incrementa la complexitat de la programació i penalitza el temps d'execució.

Si el multiprocessament és en un entorn multiprocessador paral·lel o distribuït, el tràfic dels busos de comunicació s'incrementa paral·lel al número de processadors o computadors que incorporem al sistema, i això pot arribar a ser un crític coll d'ampolla.

1.2 Processos i serveis

La majoria de nosaltres tenim a l'ordinador un antivirus instal·lat. Quan posem en marxa el nostre ordinador no arranquem l'antivirus i no interactuem amb ell a no ser que trobi un virus i ens avisi. A l'iniciar el sistema, el programa d'antivirus s'executa. Llavors es crea un procés que es manté en execució fins que apaguem el nostre ordinador. Aquest procés que controla les infeccions de tots els fitxers que entren al nostre ordinador és un **servei**.

Un servei és un tipus de procés que no té interfície amb l'usuari. Poden ser, depenent de la seva configuració, inicialitzats pel sistema de forma automàtica, en el qual van realitzant les seves funcions sense que l'usuari se n'assabenti o bé es poden mantenir a l'espera que algú els faci una petició per fer una tasca en concret.

Depenent de com s'estan executant, els processos es classificaran com a processos en primer pla (*foreground*, en anglès) i processos en segon pla (*background*). Els processos en primer pla mantenen una comunicació amb l'usuari, ja sigui informant de les accions realitzades o esperant les seves peticions per mitjà d'alguna interfície d'usuari. En canvi, els que s'executen en segon pla no es mostren explícitament a l'usuari, bé perquè no els cal la seva intervenció o bé perquè les dades requerides no s'incorporen a través d'una interfície d'usuari.

El nom *deamon* prové de les sigles angleses DAEMON (*Disk And Execution Monitor*). Sovint a la literatura tècnica s'ha estès aquest concepte usant la paraula "dimoni", pel que podeu trobar aquest mot fent referencia als programes que s'executen en segon pla.

Els serveis són processos executats en segon pla. *Servei* és una nomenclatura utilitzada en Windows. En sistemes Linux s'anomenen també processos *deamon*.

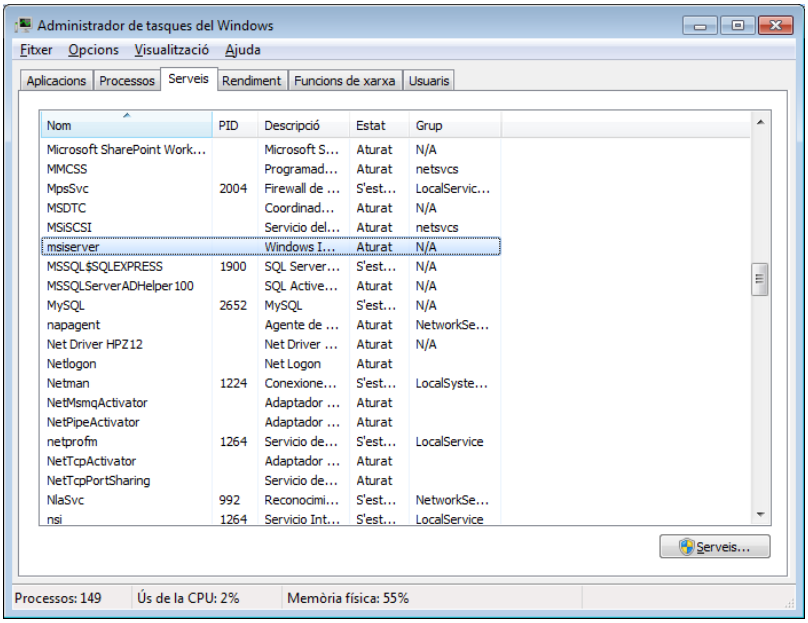
A Linux el nom dels *daemons* sempre acaben amb la lletra *d*, com ara httpd, el *daemon* de l'http.

Com que els serveis no disposen d'interfície d'usuari directa, emmagatzemen la informació generada o les possibles errades que vagin produint, en fitxers de registre habitualment coneguts com a *logs*.

Alguns serveis poden estar a l'espera de ser cridats per realitzar les seves funcions. Per exemple, un servidor web té un servei actiu. A Linux és un httpd el que està escoltant un port de la xarxa i quan l'usuari fa una petició a través de la xarxa per obtenir una plana web, és el dimoni el que s'encarrega de processar la petició i enviar la plana de retorn, si té accés autoritzat i la plana existeix, o bé fer arribar a través de la xarxa el missatge informatiu indicant la impossibilitat del lliurament.

A la figura 1.7 podem veure els serveis executats en un ordinador Windows.

Figura 1.7. Serveis Windows



Els serveis poden estar en execució local, al nostre ordinador, com ara el Firewall, l'antivirus, la connexió Wi-Fi, o els processos que controlen el nostre ordinador, o bé poden executar-se en algun ordinador d'un sistema informàtic distribuït o a Internet, etc. Les crides als serveis distribuïts es realitzen sobre protocols de comunicació. Per exemple, els serveis d'http, DNS (Domain Name System), FTP (File Transfer Protocol) són serveis que ens ofereixen servidors que són a Internet.

1.2.1 Fils i processos

Recordem el cuiner que treballa simultàniament preparant un plat per diferents comensals. El processador d'un sistema informàtic (cuiner) està executant diferents instàncies del mateix programa (la recepta). A banda, el cuiner va fent diferents parts del plat. Si la recepta és botifarra amb patates, dividirà la seva tasca en preparar la botifarra per una banda i les patates per l'altra. Ha dividit el procés en dos subprocessos. Cada un d'aquests subprocessos s'anomenen **fils**.

El sistema operatiu pot mantenir en execució diversos processos a la vegada fent servir concurrència o paral·lelisme. A més, dins de cada procés poden executar-se diversos fils, de manera que blocs d'instruccions que presentin certa independència puguin executar-se a la vegada. Els fils comparteixen els recursos del procés (dades, codi, memòria, etc). En conseqüència, els fils, a diferència dels processos, comparteixen memòria i si un fil modifica una variable del procés, la resta de fils podran veure la modificació quan accedeixin a la variable.

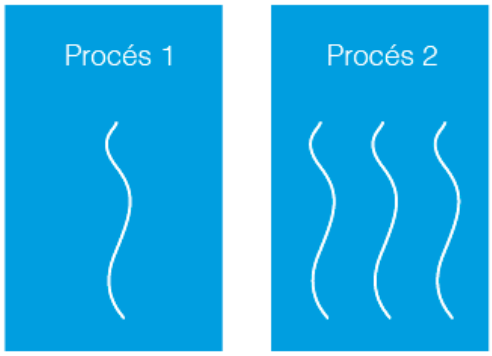
Els processos són anomenats **entitats pesades** perquè estan a espais de direccionament de memòria independents, de creació i de comunicació entre processos, cosa que consumeix molts recursos de processador. En canvi, ni la creació de fils ni la comunicació consumeixen gaire temps de processador. Per aquest motiu els fils s'anomenen **entitats lleugeres**.

Un procés estarà en execució mentre algun dels seus fils estigui actiu. Tot i així, també és cert que si finalitzem un procés de forma forçada, els seus fils també acabaran l'execució.

Podem parlar de dos nivells de fils: els fils de nivell d'usuari, que són els que creem quan programem amb un llenguatge de programació com ara Java; i els fils de segon nivell que són els que crea el sistema operatiu per donar suport als primers. Nosaltres programarem els nostres fils utilitzant unes llibreries que ens proporciona el llenguatge de programació. Són les llibreries amb les instruccions pertinents que indicaran al sistema els fils que han de crear i com gestionar-los.

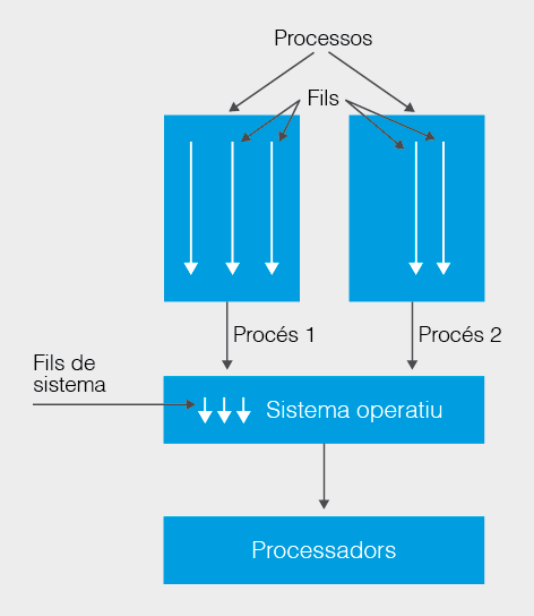
A la figura 2.8 es mostren dos processos un amb l'execució d'un fil i l'altre amb l'execució de tres fils concurrents.

Figura 1.8. Procés amb un únic fil en execució i procés amb diversos fils en execució



La figura 1.9 il·lustra el tractament que el sistema operatiu fa dels fils. En referència al processament, cada fil es processa de forma independent malgrat que pertanyi o no a un mateix procés. L'única diferència entre el tractament de fils i processos la trobem a nivell de memòria. Per a fils d'un mateix procés el sistema operatiu ha de mantenir les mateixes dades en memòria. Per a fils de diferents processos en canvi cal disposar de dades independents. En el cas que sigui necessari que un mateix processador alterni l'execució de diversos processos, caldrà restaurar la memòria específica del procés en cada canvi. D'això s'anomena també canvi de context. Si l'alternança de processament es fa entre fils d'un mateix procés, no caldrà restaurar la memòria i per tant el procés serà molt més eficient.

Figura 1.9. Dos processos amb diferents fils en execució. Cada fil s'executa de forma independent. Els fils del mateix procés comparteixen memòria.



1.3 Gestió de processos i desenvolupament d'aplicacions amb finalitat de computació paral·lela

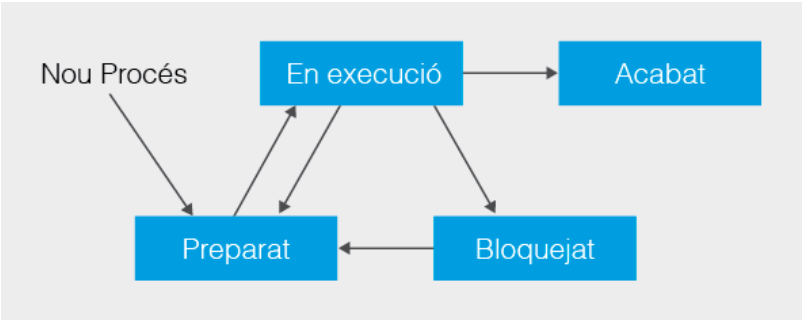
Un del objectius del sistema operatiu és proporcionar als processos els recursos necessaris per la seva execució. Ha d'aconseguir una política de planificació que determini quin procés farà ús del processador. El sistema operatiu és, doncs, l'encarregat de decidir quin procés cal executar i durant quant de temps cal fer-ho. Sortosament, els processos contemplen moltes estones de repòs en què no els cal fer servir el processador, per exemple esperant dades. Si en iniciar l'espera el procés avisa al sistema operatiu, aquest podrà reemplaçar el procés actiu per un altre en disposició de fer servir el processador. En acabar el temps de repòs, els processos hauran d'avisar de nou al sistema per tal que planifiqui una futura execució quan el processador quedi lliure. Quan finalment els processos finalitzin l'execució, el sistema operatiu alliberarà els recursos utilitzats per a l'execució i deixarà lliure el processador. El temps des que un procés inicia l'execució fins que la finalitza s'anomena cicle de vida del procés. Per tal que el sistema pugui gestionar les necessitats dels processos durant tot el seu cicle de vida, els identificarà etiquetant-los amb una estat que n'indiqui les necessitats de processament.

1.3.1 Estats d'un procés

Tots els sistemes operatius disposen d'un **planificador de processos** encarregat de repartir l'ús del processador de la forma més eficient possible i assegurant que tots els processos s'executin en algun moment. Per realitzar la planificació, el sistema operatiu es basarà en l'estat dels processos per saber quins necessitaran l'ús del processador. Els processos en disposició de ser executats s'organitzaran en una cua esperant el seu torn.

Depenent del sistema operatiu i de si s'està executant en un procés o un fil, el nombre d'estats pot variar. El diagrama d'estats de la figura 1.10 mostra els estats més habituals.

Figura 1.10. Estats d'un procés



El primer estat que ens trobem és **nou**, és a dir, quan un procés és creat. Un cop creat, passa a l'estat de **preparat**. En aquest moment el procés està preparat per fer

ús del processador, està competint pel recurs del processador. El planificador de processos del sistema operatiu és el que decideix quan entra el procés a executar-se. Quan el procés s'està executat, el seu estat s'anomena **en execució**. Un altre cop, és el planificador l'encarregat de decidir quan abandona el processador.

Assignar un temps d'execució fix a cada procés i un cop acabat aquest temps canviar el procés de l'estat d'execució a l'estat de preparat a tot esperant de ser assignat de nou al processador, pot ser una política del planificador de processos per assignar els diferents torns d'execució.

El planificador de processos segueix una política que indica quan un procés ha de canviar d'estat, quan ha de deixar lliure o entrar a fer ús del processador.

Quan un procés abandoni el processador, perquè el planificador de processos així ho haurà decidit, canviarà a l'estat de preparat i es quedarà esperant que el planificador li assigni el torn per aconseguir de nou el processador i tornar a l'estat d'execució.

Des de l'estat d'execució, un procés pot passar a l'estat de **bloquejat o en espera**. En aquest estat, el procés estarà a l'espera d'un esdeveniment, com pot ser una operació d'entrada/sortida, o l'espera de la finalització d'un altre procés, etc. Quan l'esdeveniment esperat succeeixi, el procés tornarà a l'estat de preparat, guardant el torn amb la resta de processos preparats.

L'últim estat és **acabat**. És un estat al qual s'hi arriba un cop el procés ha finalitzat tota la seva execució i estarà a punt per tal que el sistema n'alliberi quan pugui els recursos associats.

Les transicions de l'estat d'un procés poden ser provocades per algun dels motius següents:

- D'execució a bloquejat: el procés realitza alguna operació d'entrada i sortida o el procés ha d'esperar que un altre procés modifiqui alguna dada o alliberi algun recurs que necessita.
- D'execució a preparat: el sistema operatiu decideix treure un procés del processador perquè ha estat un temps excessiu fent-ne ús i el passa a l'estat de preparat. Assigna a un altre procés l'accés al processador.
- De preparat a execució: és el planificador de processos, depenent de la política que practiqui, l'encarregat de fer aquest canvi.
- De bloquejat a preparat: el recurs o la dada pel la qual havia estat bloquejat està disponible o ha acabat l'operació d'entrada i sortida.
- D'execució a acabat: el procés finalitza les seves operacions o bé un altre procés o el sistema operatiu el fan acabar.

Cada cop que un procés canvia a l'estat d'execució, s'anomena **canvi de context**, perquè el sistema operatiu ha d'emmagatzemar els resultats parcials aconseguits durant l'execució del procés sortint i cal que carregui els resultats parcials de la darrera execució del procés entrant en els registres del processador, assegurant l'accessibilitat a les variables i al codi que toqui seguir executant.

1.3.2 Planificació de processos

La planificació de processos és un conjunt de protocols o polítiques que decideixen en quin ordre s'executaran els processos al processador. Aquestes polítiques segueixen alguns criteris de prioritat segons la importància del procés, el temps d'utilització del processador, el temps de resposta del procés, etc.

La funció de planificació (*scheduler* en anglès) del nucli del sistema operatiu, és l'encarregada de decidir quin procés entra al processador, pot dividir-se en tres nivells:

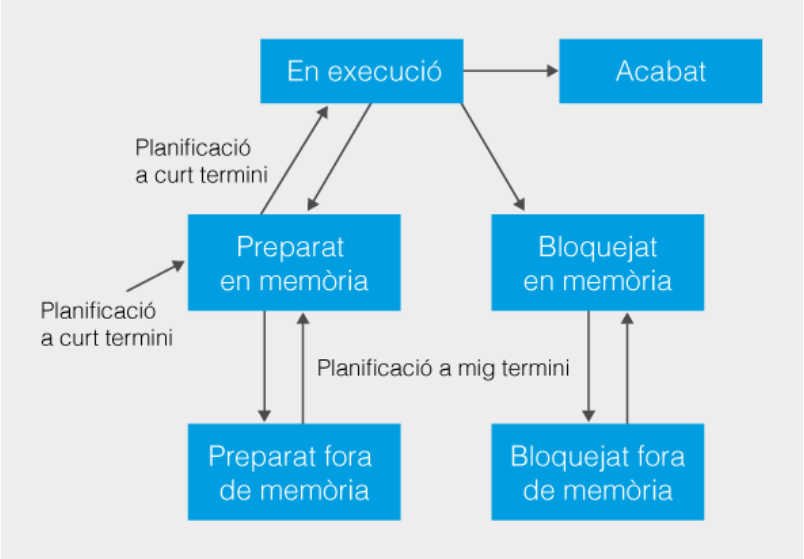
1. **Planificació a llarg termini**: en el moment que es crea un procés es decideixen alguns criteris de planificació, com ara la unitat de temps màxim que un procés podrà romandre en execució (que s'anomena **quantum**) o la prioritat inicial que se li assignarà a cada procés de forma dinàmica per la utilització del processador.

2. **Planificació a curt termini**: cada vegada que un procés abandona el processador cal prendre la decisió de quin serà el nou procés que entrarà a fer-ne ús. En aquest cas les polítiques intenten minimitzar el temps necessari per aconseguir un canvi de context. Els processos recentment executats solen tenir prioritat però cal assegurar que mai s'exclogui permanentment cap procés de l'execució.

3. **Planificació a mig termini**: existeixen altres parts del sistema operatiu que també són importants a la planificació de processos. En el cas del SWAP de memòria, si es treu un procés de la memòria per problemes d'espai fa que no pugui ser planificable de forma immediata. El planificador a mig termini serà l'encarregat de decidir quan cal portar a la memòria principal les dades dels processos emmagatzemats fora i quines caldrà traslladar de la memòria principal a l'espai d'intercanvi.

La figura 1.11 mostra gràficament els nivells de planificació de processos.

FIGURA 1.11. Planificació de processos



Els sistemes multiprocessadors presenten a més dos components de planificació específics.

El component de **planificació temporal** en el qual es troba definida la política de planificació que actua sobre cada processador de forma individual, com si fos un sistema monoprocessador. El planificador temporal decideix quan de temps cal dedicar a processar cada procés, quan cal fer canviar, per a quin motiu, etc.

El component de **planificació espacial**, en el qual es defineix com es reparteixen els processos entre els processadors. És a dir, organitza quin processador executa quin procés.

Aquesta planificació en els multiprocessadors utilitza alguns criteris per aplicar les polítiques d'assignació. El planificador va memoritzant quins són el processos executats sobre cada processador i decideix si serà adequat que torni a executar-se sobre el mateix. Una política d'**afinitat al processador** pot provocar una sobrecàrrega en alguns processadors i fer que l'execució no sigui massa eficient. Per contra una política de **repartiment de càrrega** evita aquesta suposada infrautilització d'alguns processadors, però augmenta les sobrecàrregues a la memòria compartida.

1.3.3 Programació paral·lela transparent

Quan busquem resultats ràpidament o molta potència de càlcul podem pensar en augmentar el número de processadors de la nostra computadora i executar una aplicació de forma paral·lela als diferents processadors. Però l'execució en paral·lel no és sempre possible. En primer lloc perquè hi ha aplicacions que no es poden paral·lelitzar, sinó que la seva execució ha de ser seqüencial. I en segon lloc, per la dificultat per desenvolupar entorns de programació a sistemes paral·lels, ja que es requereix un esforç important pels programadors.

Però, i si en lloc d'aplicar la concurrència explícita, en la qual el programador aporta a l'algoritme les pautes del paral·lelisme, s'aplica una programació de concurrència implícita, en la qual és el sistema operatiu el que decideix quins processos es poden dividir i quins no, aplicant certes regles genèriques i comuns a la majoria de problemes? El programador només hauria d'escollir les regles.

S'han generalitzat els sistemes informàtics multinuclis a servidors, ordinadors de sobretaula i també a sistemes mòbils, *smartphones*, tauletes, etc. Tots aquests dispositius ajuden a potenciar la programació concurrent i paral·lela. Però la programació paral·lela és complexa, cal sincronitzar processos i controlar dades compartides, cosa que afegeix complexitat a la tasca pel programador.

La plataforma Java a Java SE 6 introdueix un conjunt de paquets que proporcionen suport a la programació concurrent i Java SE 7 millora encara més el suport a la programació en paral·lel.

Java proporciona suport a la programació concurrent amb llibreries de baix nivell a través de la classe `java.lang.Thread` i de la interfície `java.lang.Runnable`. Hi ha problemes, però, que poden programar-se de forma paral·lela seguint patrons semblants. L'ús de `Thread` i `Runnable` comporta un esforç addicional del programador, obligant-lo a afegir proves extres per verificar el comportament correcte del codi, evitant lectures i escriptures errònies, sincronitzant les execucions dels processos i evitant els problemes derivats dels bloquejos.

Una aproximació relativament senzilla a alguns d'aquests problemes que poden resoldre's de forma paral·lela amb implantacions típiques que presenten un cert nivell de patronatge són els marcs predefinits de programació paral·lela del llenguatge a Java: **Executor** i **fork-join**. Són llibreries que encapsulen bona part de la funcionalitat i amb ella de la complexitat addicional que la programació de més baix nivell (`Threads`) presenta.

Ús de patrons per resoldre problemes similars

En enginyeria, anomenem patrons a aquelles solucions genèriques que ens permeten resoldre diversos problemes similars canviant petites coses. L'enginyeria informàtica fa servir aquest concepte per referir-se a biblioteques complexes que permeten un alt grau de configuració malgrat que per això sigui necessari escriure aquelles parts específiques que adaptaran el patró al problema que s'intenti resoldre.

Java fa servir aquest concepte en moltes de les seves biblioteques tot definint una estructura base a través d'interfícies i classes genèriques. El programador pot reutilitzar aquesta estructura i el codi ja implementat, usant-la en diferents situacions i adaptant-la amb codi específic per aquelles definicions abstractes i classes que restin indefinides en la implementació de la biblioteca.

La biblioteca `Executor` és una d'aquestes que implementa diversos patrons amb l'objectiu de solucionar de la forma més transparent possible bona part de les situacions en les quals es necessita programació paral·lela. Cada patró s'adapta

a diferents situacions per la qual cosa cal saber distingir les situacions i aplicar correctament el patró que calgui.

Executor

Java ens dóna eines per la simplificació de la programació paral·lela. La idea és dividir un problema en problemes més petits i cadascun d'aquests subproblemes enviar-los a executar per separat. Per tant, d'una tasca complicada en tenim diverses de més simples. Per poder executar en paral·lel totes aquestes subtasques i així augmentar el rendiment, es creen diferents subprocessos o fils, però aquesta creació és totalment transparent pel programador, és la màquina virtual de Java l'encarregada de gestionar la creació de fils per executar en paral·lel les subtasques.

Executor és una millora sobre la creació de fils, ja que s'abstreu de la creació i la gestió dels subprocessos, De fet només cal indicar la tasca o tasques a realitzar de forma paral·lela (instant objectes de tipus Runnable o Callable), escollir el gestor adequat i deixar que aquest s'encarregui de tot.

A grans trets, podem dir que disposem de tres tipus de gestors, un de genèric (`ThreadPoolExecutor`), un capaç de seguir pautes temporals d'execució (`ScheduledThreadPoolExecutor`) i una darrer que per la seva especificitat i complexitat l'estudiarem a banda. És l'anomenat marc Fork-Join.

`ThreadPoolExecutor`, és la classe base que permet implementar el gestor genèric. Els gestors genèrics són adequats per solucionar problemes que es puguin resoldre executant tasques independents entre si. Imaginem que volem saber la suma d'una llista gran de nombres. Podem optar per sumar seqüencialment la llista o bé per dividir-la en dos o tres trossos i sumar cada tros per separat. Al final només caldrà afegir el valor de cada tros al resultat. La suma de cada tros és totalment independent i per tant pot realitzar-se amb independència de la resta. És a dir, només caldria crear tantes tasques com trossos a sumar i passar-les al gestor perquè les executi.

`ThreadPoolExecutor`, no crea pas un fil per cada tasca (instància de `Callable` o `Runnable`). De fet, és habitual assignar-li més tasques que no pas fils. Les tasques pendents romanen en una cua de tasques i un nombre de fils determinat s'encarrega d'anar-les executant. Quants fils cal crear? Segur que aquesta és un decisió difícil i dependrà de cada cas. Per això JAVA ens ofereix 3 formes ràpides de configurar les instàncies de `ThreadPoolExecutor` fent servir la classe `Executors`. Es tracta d'una classe que aglutina un conjunt d'utilitats en forma de mètodes estàtics. Els tres tipus s'obtenen a partir dels mètodes estàtics següents:

- `newCachedThreadPool()`: crea un *pool* que va creant fils a mesura que els necessita i reutilitza els fils inactius.

- `newFixedThreadPool(int numThreads)`: crea un *pool* amb un número de fils fix. L'indicat en el paràmetre. Les tasques s'emmmagatzemen en una cua de tasques i es van associant als fils a mida que aquest quedin lliures.

- `newSingleThreadExecutor()`: crea un *pool* amb un sol fil que serà el que processi totes les tasques.

Qualsevol d'aquestes configuracions obtindrà una instància de `ThreadPoolExecutor` que es comportarà com indiquen els comentaris. Per poder afegir noves tasques, iniciar el processament paral·lel i mantenir-lo actiu fins acabar haver processat totes les tasques, la classe `ThreadPoolExecutor` disposa de mètodes com `execute` per anar introduint una tasca (instància de `Runnable` o `Callable`) cada cop, `invokeAll` per afegir, tot d'una, una llista de diverses tasques, `getPoolSize()` per obtenir el nombre fils en execució o també `getCompleteTaskCount()` per saber el nombre de tasques finalitzades.

`ScheduledThreadPoolExecutor` és un cas específic de *pool*. Es tracta d'un gestor de fils, amb una política d'execució associada a una seqüència temporal. És a dir executa les tasques programades cada cert temps. Pot ser d'una sola vegada o de forma repetitiva. Per exemple si volem consultar el nostre correu cada 5 minuts, mentre realitzem altres feines, podem programar un fil independent amb aquesta tasca usant `ScheduledThreadPoolExecutor`. La tasca podria comprovar el correu i avisar només en cas que hagin arribat nous. D'aquesta manera podem concentrar-nos en la nostra feina deixant que el fil periòdic treballi per nosaltres. Podem obtenir instàncies de `ScheduledThreadPoolExecutor` usant una utilitat de la classe `Executor`, el mètode *static* `newScheduledThreadPool(int numThreads)`.

La classe `ScheduledThreadPoolExecutor` disposa dels mètodes següents per gestionar el retard i cadència:

- `schedule (Callable task, long delay, TimeUnit timeunit)`: crea i executa una tasca (*task*) que és invocada després d'un temps concret (*delay*) expressat en les unitats indicades (*timeunit*).
- `schedule (Runnable task, long delay, TimeUnit timeunit)`: crea i executa una acció (*task*), una sola vegada, després que hagi transcorregut un temps concret (*delay*) expressat en les unitat indicades (*timeunit*).
- `scheduleAtFixedRate (Runnable task, long initialDelay, long period, TimeUnit timeunit)`: crea i executa una tasca (*task*) de forma periòdica. La primera vegada s'invocarà després d'un retard inicial (*initialDelay*) i posteriorment cada període de temps determinat (*period*). El temps es mesura en les unitats indicades (*timeunit*).
- `scheduleWithFixedDelay (Runnable task, long initialDelay, long delay, TimeUnit timeunit)`: crea i executa una tasca (*task*) de forma periòdica. La primera vegada s'invocarà després d'un retard inicial (*initialDelay*) i posteriorment, en acabar la darrera tasca llançada, no es començarà la següent, fins que no hagi passat el temps indicat a *delay*. El temps es mesura en les unitats indicades (*timeunit*).

El mètodes que reben tasques de tipus `Callable` retornaran el resultat obtingut durant el càlcul.

Exemple amb ThreadPoolExecutor

Anem ara a veure un exemple per calcular deu multiplicacions aleatòries d'enters usant un `Executor` amb un màxim de 3 fils. Per a la instanciació del gestor caldrà:

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.
    newFixedThreadPool(3);
```

Això significa que si hi ha més de 3 tasques per executar, únicament n'enviarà 3 a executar i la resta es quedaran bloquejades fins que un fil acabi el seu processament.

La classe interior `Multiplicacio` implementa la interfície `Callable`. Es podria fer servir el mètode *run* i implementar `Runable`. La diferència és que `Callable` retorna valors del resultat en forma de qualsevol objecte. La concurrència es realitza dins del mètode `call()`.

Es crea una llista de deu multiplicacions a l'atzar. I després crida al mètode d'executor `invokeAll()`. Aquest rep una col·lecció de tasques de tipus `Callable`, les executa i en retorna el resultat dins d'un objecte `Future`. La classe `java.util.concurrent.Future` és una interfície dissenyada per donar suport a les classes que mantinguin dades calculades de forma asíncrona. Les instàncies de `Future` obliguen a fer servir els mètodes `get` per accedir als resultats. Això assegura que el valor real només serà accessible quan els càlculs hagin acabat. Mentre durin els càlculs la invocació del mètode `get` implicarà un bloqueig del fil que l'estigui invocant.

Els objectes Future es generen com a conseqüència d'executar el mètode `call` de les instàncies `Callable`. Per cada `Callable` invocada es genera un objecte Future que romandrà bloquejat fins que la instància Callable associada finalitzi.

```
package multiplicallista;

import java.util.*;
import java.util.concurrent.*;

public class MultiplicaLlista {

static class Multiplicacio implements Callable<Integer> {
 private int operador1;
 private int operador2;
 public Multiplicacio(int operador1, int operador2) {
   this.operador1 = operador1;
   this.operador2 = operador2;
 }
 @Override
 public Integer call() throws Exception {
     return operador1 * operador2;
 }
}
  public static void main(String[] args) throws
       InterruptedException, ExecutionException {

     ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors
         .newFixedThreadPool(3);
     List<Multiplicacio> llistaTasques= new ArrayList<
         Multiplicacio>();
     for (int i = 0; i < 10; i++) {
       Multiplicacio calcula = new Multiplicacio((int)(Math.
           random()*10), (int)(Math.random()*10));
       llistaTasques.add(calcula);
     }
     List <Future<Integer>> llistaResultats;
     llistaResultats = executor.invokeAll(llistaTasques);

     executor.shutdown();
```

```
32      for (int i = 0; i < llistaResultats.size(); i++) {
33            Future<Integer> resultat = llistaResultats.get(i);
34            try {
35              System.out.println("Resultat tasca "+i+ " és:" +
                    resultat.get());
36            } catch (InterruptedException | ExecutionException e)
                  {
37            }
38        }
39
40    }
41  }
```

`ThreadPoolExecutor` disposa també del mètode `invokeAny(tasques)` que rep una col·lecció d'instàncies de `Callable` que anirà executant fins que una d'elles finalitzi sense error. En aquest moment retornarà el resultat obtingut per la tasca finalitzada en un objecte `Future`.

Un altre mètode de `ThreadPoolExecutor` que cal conèixer és `shutdown()`, que impedeix executar noves tasques però deixa que els fils que estan en execució puguin acabar.

Exemple amb ScheduledThreadPoolExecutor

En el següent exemple utilitzarem l'executor `ScheduledThreadPoolExecutor` per a programar tasques que s'executin periòdicament, després d'un retard programat.

Primer creem un *pool* de dos fils del `ScheduledThreadPoolExecutor` i utilitzem `scheduleWithFixedDelay`, que ens permet programar i executar l'inici de la tasca (`initialDelay`) i programar les tasques successives després de la primera finalització (*long period*).

Executa el mètode *run* d' `ExecutaFil`. La primera vegada al cap de 2 segons i posteriorment cada 3 segons.

Per últim, el mètode `awaitTermination`es queda a l'espera abans del *shutdown* (aturada), fins que totes les tasques hagin acabat, es produeixi el *timeout* (temps d'espera) o els fils siguin finalitzats abruptament, el que passi primer.

```
1  package tascaprogramada;
2
3  import java.util.Calendar;
4  import java.util.GregorianCalendar;
5  import java.util.concurrent.ExecutionException;
6  import java.util.concurrent.Executors;
7  import java.util.concurrent.ScheduledExecutorService;
8  import java.util.concurrent.TimeUnit;
9
10 public class TascaProgramada {
11
12     public static void main(final String... args) throws
           InterruptedException, ExecutionException {
13        //mostrem hora actual abans d'execució
14         Calendar calendario = new GregorianCalendar();
15        System.out.println("Inici: "+ calendario.get(Calendar.
              HOUR_OF_DAY) + ":" + calendario.get(Calendar.MINUTE) +
               ":" + calendario.get(Calendar.SECOND));
16
17         // Crea un pool de 2 fils
18         final ScheduledExecutorService schExService = Executors.
               newScheduledThreadPool(2);
19         // Crea objecte Runnable.
20         final Runnable ob = new TascaProgramada().new ExecutaFil
               ();
21         // Programa Fil, s'inicia als 2 segons i després es va
               executant cada 3 segons
22         schExService.scheduleWithFixedDelay(ob, 2, 3, TimeUnit.
               SECONDS);
```

```
23          // Espera per acabar 10 segons
24          schExService.awaitTermination(10, TimeUnit.SECONDS);
25          // shutdown .
26          schExService.shutdownNow();
27          System.out.println("Completat");
28      }
29      // Fil Runnable
30      class ExecutaFil implements Runnable {
31          @Override
32          public void run() {
33              Calendar calendario = new GregorianCalendar();
34              System.out.println("Hora execució tasca: "+
                    calendario.get(Calendar.HOUR_OF_DAY) + ":" +
                    calendario.get(Calendar.MINUTE) + ":" +
                    calendario.get(Calendar.SECOND));
35              System.out.println("Tasca en execució");
36
37              System.out.println("Execució acabada");
38          }
39      }
40
41
42  }
```

Quan executem el codi anterior obtindrem per la pantalla uns resultats semblants als següents:

run:

Inici: 11:27:10

Hora execució tasca: 11:27:12

Tasca en execució

Execució acabada

Hora execució tasca: 11:27:15

Tasca en execució

Execució acabada

Hora execució tasca: 11:27:18

Tasca en execució

Execució acabada

Completat

BUILD SUCCESSFUL (total time: 10 seconds)

Fixeu-vos com la primera execució es produeix després de 2 segons, a continuació la tasca s'executa cada 3 segons.

Fork-Join

El marc Fork-Join és una evolució dels patrons Executor. De fet aquí trobem també un gestor de fils (la classe `ForkJoinPool`), però una mica més sofisticat. Processa les tasques amb l'algoritme per robatori (*work-stealing*). Això vol dir que el gestor del *pool* busca fils poc actius i hi intercanvia tasques en espera. A més les tasques poden crear noves tasques que s'incorporaran a la cua de tasques pendents per a

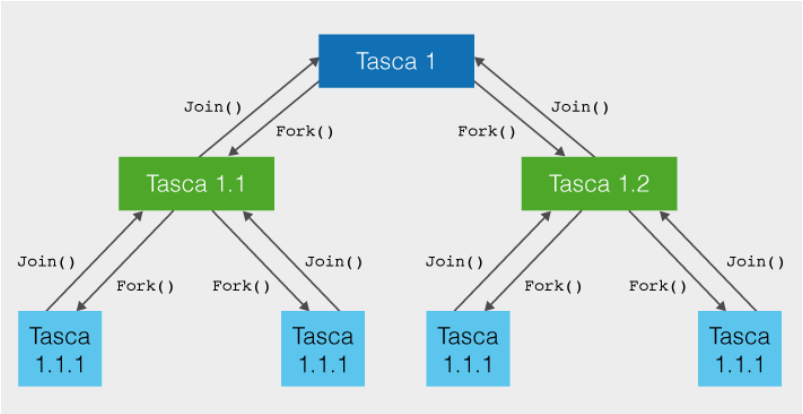
ser executades. L'eficiència aconseguida és força alta, ja que s'aprofita al màxim el processament paral·lel. Per aquest motiu Fork_Join és una alternativa ideal per algoritmes que puguin resoldre's de forma recursiva ja que s'aconseguirà la màxima eficiència intercanviant aquelles tasques que es trobin esperant resultats per d'altres que permetin avançar en els càlculs.

La classe `ForkJoinTask` és una classe abstracta per les tasques que s'executen a `ForkJoinPool` i conté els mètodes `fork` i `join`. El mètode `fork()` situa la tasca invocada a la cua d'execucions en qualsevol moment per tal que sigui planificada. Això permet que a una tasca crear-ne de noves i deixar-les a punt per ser executades quan el gestor ho consideri.

El mètode `join()` aturarà l'execució del fil invocador a l'espera que la tasca invocada finalitzi l'execució i retorni si fos el cas els resultats. El bloqueig del fil posarà en alerta el gestor `ForkJoinPool` que podrà intercanviar la tasca aturada per una altra que resti en espera.

A la figura 1.12 es mostra com cooperen les tasques a través dels mètodes `fork()` i `join()`.

Figura 1.12. Mètodes fork-join



`ForkJoinTask` està implementada per dues classes que l'especialitzen, la classe `RecursiveTask` i la classe `RecursiveAction`. La primera és adequada per a tasques que necessitin calcular i retornar un valor. La segona en canvi representa procediments o accions que no necessiten retornar cap resultat.

Quan fem servir `RecursiveAction` resulta molt pràctic fer servir el mètode `invokeAll`, en comptes de cridar, per cada tasca generada els mètodes `fork` i `join`. Concretament el mètode `invokeAll` fa un invocació de `fork` per cada una de les tasques rebudes com a paràmetre i seguidament s'espera a la finalització de cada fil amb una crida al mètode `join` de les tasques engegades. `invokeAll` no retorna cap resultat, per tant no resulta gaire útil d'utilitzar amb objectes de tipus `RecursiveTask`.

Algoritmes recursius i algoritmes iteratius

Els algoritmes recursius tenen la peculiaritat de poder-se resoldre realitzant diverses vegades una mateixa operació però usant valors diferents. Cada execució s'encarrega de realitzar un càlcul parcial fent servir les dades d'una o més

execucions anteriors de la mateixa operació. El darrer resultat parcial calculat coincidirà amb la solució final. Per tal que el càlcul recursiu sigui factible és necessari que existeixi un solució trivial per algun dels valors a operar. Per exemple l'agoritme que calcula el valor factorial d'un número és recursiu perquè podem dir que el valor factorial d'un enter positiu és: $n! = n*(n-1)!$ excepte quan n val 1 (el cas trivial) ja que sabem que $1! = 1$.

Usant un llenguatge de programació podem expressar aquest algoritme recursiu fent:

```
long factoria(long n){
    if(n==1){
       return n;
    }else{
       return n*factorial(n-1);
    }
}
```

Cal recordar que els algoritmes recursius consumeixen molta memòria i que en un processament seqüencial, poden resultar poc eficients si el seu càlcul requereix moltes crides recursives. Però qualsevol algoritme recursiu pot transformar-se sempre en iteratiu. En un processament seqüencial, els algoritmes iteratius són més eficients que els recursius. Ara bé, és possible aprofitar el processament paral·lel i les característiques recursives d'alguns algoritmes per aconseguir executar en paral·lel les crides recursives de manera que incrementem l'eficiència final. Malauradament, l'increment d'eficiència només serà efectiu si el càlcul a realitzar té prou entitat.

Per donar l'entitat idònia als càlculs i agilitzar l'eficiència, es combina el càlcul iteratiu amb el recursiu. És a dir, es defineix un llindar o condició a partir de la qual seria preferible resoldre el problema iterativament. Mentre no s'arribi al llindar el càlcul es derivarà a la versió recursiva executada en paral·lel, fent servir el marc Fork-Join. Quan arribem al llindar, en canvi, realitzarem el càlcul seqüencialment. L'elecció del llindar no és trivial perquè d'ella dependrà en bona part disposar d'un algoritme eficient o no.

Així doncs, l'esquema d'execució de les tasques a implementar a les instàncies de ForkJoinTask seria similar al codi següent:

```
if (el que queda per resoldre surt més a compte resoldre-ho directament ){ //
     lindar
  execució seqüencial
}else{
  divisió recursiva de la tasca.
  crida/planificació de l'execució paral·lela de les tasques creades
  recollida dels resultats i combinació adequada per obtenir el calcul desitjat
       .
}
```

Veiem el que acabem d'explicar amb un exemple. Buscarem dins d'un *array* d'enters el número més gran. La recursió es basarà en dividir la col·lecció d'enters en dos i demanar per cada tros l'obtenció del valor mínim i el càlcul seqüencial en un algoritme de cerca iteratiu.

Usarem RecursiveTask perquè ens cal retornar el valor mínim. Els valors de partida els passarem sempre a través del constructor de la tasca i els emmagat-

zemarem com atributs per tenir-los disponibles en el mètode `compute` que és l'equivalent específic de les tasques `ForkJoinTask` al `run` o `call` ja vistos en parlar de l'`Executor`.

```
package ioc.dam.m09.u1.forkjoin.maxim;

import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class MaximTask extends RecursiveTask<Short> {
    //private static final int LLINDAR=10000000;
    private static final int LLINDAR=10000000;
    private short[] arr ;
    private int inici, fi;

    public MaximTask(short[] arr, int inici, int fi) {
            this.arr = arr;
            this.inici = inici;
            this.fi = fi;
    }

    private short getMaxSeq(){
        short max = arr[inici];
        for (int i = inici+1; i < fi; i++) {
            if (arr[i] > max) {
                    max = arr[i];
            }
        }
        return max;
    }

    private short getMaxReq(){
        MaximTask task1;
        MaximTask task2;
        int mig = (inici+fi)/2+1;
        task1 = new MaximTask(arr, inici, mig);
        task1.fork();
        task2 = new MaximTask(arr, mig, fi);
        task2.fork();
        return (short) Math.max(task1.join(), task2.join());
    }

    @Override
    protected Short compute() {
        if(fi - inici <= LLINDAR){
            return getMaxSeq();
        }else{
            return getMaxReq();
        }
    }

    public static void main(String[] args) {

        short[] data = createArray(100000000);

        // Mira el número de processadors
        System.out.println("Inici càlcul");
        ForkJoinPool pool = new ForkJoinPool();

        int inici=0;
        int  fi= data.length;
        MaximTask tasca = new MaximTask(data, inici, fi);

        long time = System.currentTimeMillis();
        // crida la tasca i espera que es completin
        int result1 = pool.invoke(tasca);
```

```
63          // màxim
64          int result=  tasca.join();
65          System.out.println("Temps utilitzat:" +(System.
                currentTimeMillis()-time));
66
67          System.out.println ("Màxim es " + result);
68      }
69
70
71      private static short [] createArray(int size){
72          short[] ret = new short[size];
73          for(int i=0; i<size; i++){
74              ret[i] = (short) (1000 * Math.random());
75              if(i==((short)(size*0.9))){
76                  ret[i]=1005;
77              }
78          }
79          return ret;
80      }
81  }
```

Proveu diferents llindars per veure que l'actual és el més eficient.

1.4 Sincronització i comunicació entre processos

Comunicar-se i sincronitzar-se són les dues accions més importants en l'execució de processos concurrents. Quan diversos processos estan en execució a la vegada no podem controlar l'ordre en què acabaran executant-se, ja que això dependrà de la política de planificació escollida i dels processos que es trobin en execució a cada moment. Però l'ordre de processament, en algunes ocasions, pot arribar a ser fonamental per aconseguir un resultat correcte. Les tècniques que ens ajuden a controlar l'ordre d'execució dels diferents processos s'anomenen tècniques de comunicació i sincronització.

1.4.1 Competència de recursos i sincronització

Els processos concurrents es poden classificar en independents o cooperants segons el seu grau de col·laboració. Un procés **independent** no necessita ajuda ni cooperació d'altres processos. En canvi, els processos **cooperants** estan dissenyats per treballar conjuntament amb altres processos i, per tant, s'han de poder comunicar i interactuar entre ells.

Aquestes col·laboracions impliquen una coordinació de tots el processos implicats. A vegades la coordinació requereix de cert nivell de comunicació per poder-se sincronitzar. Tot i així, una comunicació detallada és força costosa i sovint n'hi ha prou a assegurar només que els processos no accedeixin a la vegada a un mateix recurs o executin una mateixa operació. En aquest darrer cas direm que els processos competeixen entre si perquè no hi ha manera de saber l'ordre exacte amb què finalment s'executaran els processos.

En la programació concurrent el procés de **sincronització** permet que els processos que s'executen de forma simultània es coordinin, aturant l'execució d'aquells que vagin més avançats fins que es compleixin les condicions òptimes per estar segurs que els resultats finals seran correctes.

Per sincronitzar processos podem fer servir diferents mètodes:

- **Sincronisme condicional**: o condició de sincronització. Un procés o fil es troba en estat d'execució i passa a estat de bloqueig tot esperant que una certa condició es compleixi per continuar la seva execució. Un altre procés fa que aquesta condició es compleixi i així el primer procés passa de nou a l'estat d'execució.

- **Exclusió mútua**: es produeix quan dos o més processos o fils volen accedir a un recurs compartit. S'han d'establir protocols d'execució perquè no accedeixin de forma concurrent al recurs.

- **Comunicació per missatges**: en la qual els processos s'intercanvien missatges per comunicar-se i sincronitzar-se. És la comunicació típica en sistemes distribuïts i és utiltzada també en sistemes no distribuïts.

1.4.2 Sincronització i comunicació

La col·laboració entre processos dóna lloc a un seguit de problemes clàssics de comunicació i sincronització. Sincronitzar i comunicar processos és, doncs, bàsic per tal de solucionar-los.

Seccions crítiques

Les seccions crítiques són un dels problemes que amb més freqüència es donen en programació concurrent. Tenim diversos processos que s'executen de forma concurrent i cadascun d'ells té una part de codi que s'ha d'executar de forma exclusiva ja que accedeix a recursos compartits com ara fitxers, variables comuns, registres de bases de dades, etc.

La solució passarà per obligar a accedir als recursos a través de l'execució d'un codi que anomenarem secció crítica i que ens permetrà protegir aquells recursos amb mecanismes que impedeixin l'execució simultània de dos o més processos dins els límits de la secció crítica.

Aquests algoritmes de sincronització que eviten l'accés a una regió crítica per més d'un fil o procés i que garanteixen que únicament un procés estarà fent ús d'aquest recurs i la resta que volen utilitzar-lo estaran a l'espera que sigui alliberat, s'anomena **algoritme d'exclusió mútua**.

Exclusió mútua (MUTEX, *mutual exclusion* en anglès) és el tipus de sincronització que impedeix que dos processos executin simultàniament una mateixa secció crítica.

Un mecanisme de sincronització en forma de codi que protegeixi la secció crítica haurà de tenir una forma com el següent:

```
Entrada_Secció_Critica /* Sol·licitud per executar Secció Crítica
     */

/* codi Secció Crítica */

Sortida_Secció_Critica /* altre procés pot executar la secció Crí
    tica */
```

L'`Entrada_Secció_Crítica` representa la part del codi en la qual els processos demanen permís per entrar a la secció crítica. La `Sortida_Secció_Critica` en canvi, representa la part que executen els processos quan surten de la secció crítica alliberant la secció i permet a d'altres processos entrar-hi.

Per validar qualsevol mecanisme de sincronització d'una secció crítica s'han de complir els criteris següents:

- Exclusió mútua: no pot haver-hi més d'un procés simultàniament en la secció crítica.
- No inanició: un procés no pot esperar un temps indefinit per entrar a executar la secció crítica.
- No interbloqueig: cap procés de fora de la secció crítica pot impedir que un altre procés entri a la secció crítica.
- Independència del maquinari: inicialment no s'han de fer suposicions respecte al número de processadors o la velocitat dels processos.

Un error de consistència típic quan no hi ha control sobre una secció crítica es pot il·lustrar amb l'exemple de dos processos que volen modificar una variable comuna x. El procés A vol incrementar-la: x++. El procés B disminuir-la: x–;. Si els dos processos accedeixen a llegir el contingut de la variable al mateix temps, els dos obtindran el mateix valor, si fan la seva operació i guarden el resultat, aquest serà inesperat. Dependrà de qui salvi el valor d'x en darrer lloc.

La taula 1.1 mostra un exemple similar. Un codi és accessible per dos fils o processos, veiem que si no hi ha cap tipus de control sobre l'accés, el primer fil accedeix a les instruccions i abans d'arribar a la instrucció d'increment de la variable a++, el segon procés entra a executar el mateix codi. El resultat és que el segon procés pren el valor 4, per tant erroni, ja que el primer procés no hauria augmentat la variable.

TAULA 1.1. Secció crítica

Procés 1	Temps	Procés 2
pout.print(a);	1	
	2	pout.print(a);
a=a+4;	3	
	4	a=a+4;
pout.print("+4=");	5	
pout.println(a);	6	
	7	pout.print("+4=");
	8	pout.println(a);

En aquest exemple suposarem què pot representar la sortida a un fitxer, què arriba per paràmetre a la zona crítica i que cada procés treballa amb un fitxer diferent. Suposarem també que la variable *a* té un valor inicial de *4* i que es tracta d'una variable compartida per ambdós processos. Podeu imaginar que les sortides seran força sorprenents, ja que en ambdós fitxers indicaria: *4+4=12*.

Per evitar aquest problema, únicament un fil hauria d'executar aquesta part de codi de forma simultània. Aquesta part de codi, que és susceptible a aquest tipus d'error, s'ha de declarar com a secció crítica per evitar aquest tipus d'error.

Les instruccions que formen part d'una secció crítica s'han d'executar con si fossin una única instrucció. S'han de sincronitzar els processos per tal que un únic procés o fil pugui excloure de forma temporal a la resta de processos d'un recurs compartit (memòria, dispositius, etc.) de tal manera que la integritat del sistema quedi garantida.

Productor-consumidor

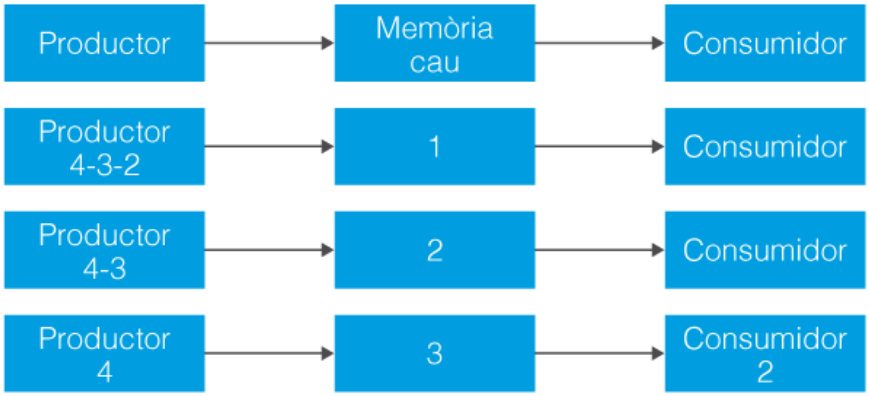
El problema productor-consumidor és un exemple clàssic on cal donar un tractament independent a un conjunt de dades que es van generant de forma més o menys aleatòria o si més no d'una forma en què no és possible predir en quin moment es generarà una dada. Per evitar un ús excessiu dels recursos de l'ordinador esperant l'arribada de dades, els sistema preveu dos tipus de processos: els productors, encarregats d'obtenir les dades a tractar i els consumidors, especialitzats en fer el tractament de les dades obtingudes pels productors.

En aquesta situació el productor genera un seguit de dades que el consumidor recull. Imaginem que és el valor d'una variable que el productor modifica i el consumidor l'agafa per utilitzar-la. El problema ve quan el productor produeix dades a un ritme diferent del que el consumidor les agafa. El productor crea una dada i canvia la variable al seu valor. Si el consumidor va més lent, el productor té temps a generar una nova dada i torna a canviar la variable. Per tant el procés del consumidor ha perdut el valor de la primera dada. En el cas que sigui el consumidor el que va més ràpid, pot passar que agafi dues vegades una mateixa dada, ja que el productor no ha tingut temps de substituir-la, o que no trobi res

per consumir. La situació pot complicar-se encara més si disposem de diversos processos productors i consumidors.

A la figura 1.13 podem veure com dos processos comparteixen un recurs comú (la memòria) i la il·lustració del problema quan no està sincronitzat el procés productor i el consumidor, i el productor és més ràpid que el consumidor.

Figura 1.13. Productor-consumidor



Imaginem que en la figura 1.13 la memòria comú és com una caixa que és capaç de guardar una única dada, un enter. Existeix un procés productor que genera els números enters i els deixa a la caixa. Mentrestant hi ha un procés consumidor que agafa el número enter de la caixa.

Com és el cas, el productor deixa a la memòria el número 1 i abans que el procés consumidor agafi la dada, genera un altre número, el 2, que substitueix l'anterior. El consumidor ara sí que agafa el número 2 però, tal com podem veure, el número 1 s'ha perdut, produint segurament uns resultats erronis.

Una manera de solucionar el problema consisteix a ampliar la ubicació on el productor escriu les dades de manera que sigui possible mantenir diverses dades a l'espera que siguin consumides mentre els processos consumidors estiguin ocupats. És a dir, els productors emmagatzemaran les dades en un llista (*array*), que tradicionalment es coneix com a *buffer* i els consumidors les aniran extraient.

Un ***buffer*** és un espai de memòria per emmagatzemar dades. Es poden implementar en forma de cues.

La figura 1.14 és l'esquema d'un procés productor que deixa informació a un *buffer* i que és agafada per un procés consumidor.

Figura 1.14. Buffer

Malauradament aquest mecanisme no soluciona tots els problemes, ja que pot ser que un consumidor intenti accedir a les dades malgrat el productor no n'hagi encara escrit cap, pot passar que l'espai destinat a emmagatzemar la col·lecció de dades s'ompli a causa que la producció de dades sigui sempre molt més ràpida que els processos consumidors, o bé podria donar-se el cas que dos processos productors coincidissin a l'hora de deixar una dada o que diversos processos consumidors intentessin accedir a l'hora.

Per tant, ha d'existir un mecanisme que aturi l'accés a la dada dels productors i dels consumidors en cas necessari. Una secció crítica. Malauradament no n'hi ha prou a restringir l'accés a les dades, perquè podria donar-se el cas que un procés consumidor esperant l'arribada d'una dada impedeixi l'accés dels processos productors de forma que la dada no arribés mai. Una situació com la descrita es coneix amb el nom de problemes d'interbloqueig (*deadlock* en anglès).

Anomenem ***interbloqueig*** a la situació extrema que ens trobem quan dos o més processos estan esperant l'execució de l'altre per poder continuar de manera que mai aconseguiran desbloquejar-se.

També anomenem ***inanició*** a la situació que es produeix quan un procés no pot continuar la seva execució per falta de recursos. Per exemple si féssim créixer el *buffer* il·limitadament.

Per solucionar el problema cal sincronitzar l'accés al *buffer*. S'ha d'accedir en exclusió mútua per tal que els productors no alimenti el *buffer* si aquest ja està ple i els consumidors no hi puguin accedir si està buit. Però a més caldrà independitzar les seccions crítiques d'accés dels productors de les seccions crítiques d'accés dels consumidors evitant així que es pugui produir l'interbloqueig.

Això però obligarà a crear un mecanisme de comunicació entre seccions crítiques de manera que cada cop que un productor deixi una dada disponible, avisi els consumidors que puguin restar esperant que almenys un d'ells pugui iniciar el processament de la dada.

Lectors-escriptors

Un altre tipus de problema que apareix en la programació concurrent, és el produït quan tenim un recurs compartit entre diversos processos concurrents com poden ser un fitxer, una base de dades, etc. que va actualitzant-se periòdicament. En aquest cas els processos lectors no consumeixen la dada sinó que només la utilitzen i per tant es permet la seva consulta de forma simultània, malgrat que no la seva modificació.

Així, els processos que accedeixin al recurs compartit per llegir el seu contingut s'anomenaran **lectors**. En canvi, els que hi accedeixin per modificar-lo rebran el nom de **escriptors**.

Si la tasca de lectors i escriptors no es realitza de forma coordinada podria passar que un lector llegís diverses vegades la mateixa dada o que l'escriptor modifiques el contingut abans que haguessin llegit tots els lectors, o que la dada actualitzada

per un escriptor malbaratés l'actualització d'un altre, etc. A més, la manca de coordinació obliga als lectors a comprovar periòdicament si els escriptors han fet modificacions, cosa que farà augmentar l'ús del processador i per tant podria minvar l'eficiència.

Aquest problema de sincronització de processos s'anomena **problema de lectors-escriptors**. Per evitar-lo cal assegurar que els processos escriptors accedeixen de forma exclusiva al recurs compartit i que a cada modificació s'avisa als processos lectors interessats en el canvi.

Així, els processos lectors poden restar en espera fins que són avisats que hi ha noves dades i poden començar a llegir; d'aquesta manera s'evita que els lectors estiguin constantment accedint al recurs sense que l'escriptor hagi posat cap nova dada, optimitzant, d'aquesta manera, els recursos.

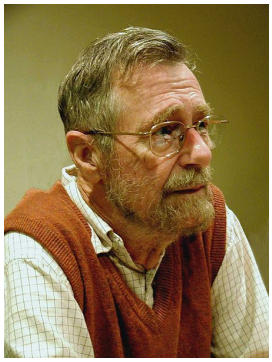
1.4.3 Solucions a problemes de sincronització i comunicació

Al llarg de la història s'han proposat solucions específiques pels problemes anteriors que val la pena tenir en compte, malgrat que resulta difícil generalitzar-ne una solució ja que depenen de la complexitat, la quantitat de seccions crítiques, del nombre de processos que requereixin exclusió mútua i de la interdependència existent entre processos. Aquí veurem la sincronització per mitjà de semàfors, de monitors i de pas de missatges.

Semàfors

Imaginem una carretera d'un sol carril que ha de passar per un túnel. Hi ha un semàfor a cada extrem del túnel que ens indica quan podem passar i quan no. Si el semàfor està en verd el cotxe passarà de forma immediata i el semàfor passarà a vermell fins que surti. Aquest símil ens introdueix a la definició real d'un semàfor.

Els semàfors són una tècnica de sincronització de memòria compartida que impedeix l'entrada del procés a la secció crítica tot bloquejant-lo. El concepte va ser introduït per l'informàtic holandès Dijkstra per resoldre el problema l'exclusió mútua i permetre resoldre gran part dels problemes de sincronització entre processos.



Edsger Wybe Dijkstra, informàtic holandès. La seva idea d'exclusió mútua ideada durant la dècada dels 60 és utilitzada per molts processadors i programadors moderns

Els semàfors, no només controlen l'accés a la secció crítica sinó que a més disposen d'informació complementària per poder decidir si cal o no bloquejar l'accés d'aquells processos que ho sol·licitin. Així per exemple, serviria per solucionar problemes senzills (amb poca interdependència) del tipus escriptors-lectors o productors-consumidors.

La solució per exemple en el cas dels escriptors-lectors passaria per fer que els lectors abans de consultar la secció crítica demanessin permís d'accés al semàfor, el qual en funció de si es troba bloquejat (vermell) o alliberat (verd) aturarà l'execució del procés sol·licitant o bé el deixarà continuar.

Els escriptors per altra banda, abans d'entrar a la secció critica, manipularan el semàfor posant-lo en vermell i no el tornaran a posar en verd fins que hagin acabat d'escriure i abandonin la secció crítica.

De forma genèrica distingirem 3 tipus d'operacions en un semàfor:

El semàfor admet 3 operacions:

- Inicialitza (*initial*): es tracta de l'operació que permet posar en marxa el semàfor. La operació pot rebre un valor per paràmetre que indicarà si aquest començarà boquejat (vermell) o alliberat (verd).
- Allibera(s) (*sendSignal*): canvia el valor intern del semàfor posant-lo en verd (l'allibera). Si existeixen processos en espera, els activa per tal que finalitzin la seva execució.
- Bloqueja(s) (*sendWait*): serveix per indicar que el procés actual vol executar la secció crítica. En cas que el semàfor es trobi bloquejat, s'aturà l'execució del procés. També permet indicar que cal posar el semàfor en vermell.

En cas que es necessiti exclusió mútua, el semàfor disposarà també d'un sistema d'espera (per mitjà de cues de processos) que garanteixi l'accés a la secció crítica d'un únic procés a la vegada.

En realitat la implementació d'un semàfor dependrà molt del problema a resoldre, malgrat que la dinàmica del funcionament sigui sempre molt similar. Així per exemple els semàfor que donin suport al problema de tipus productor-consumidor, necessiten implementar-se fent servir exclusió mútua tant per alliberar com per bloquejar. A més, l'execució d'alliberament incrementarà en una unitat un comptador intern, mentre que l'execució de bloqueig a banda d'aturar el procés quan el semàfor es trobi bloquejat, disminuirà en una unitat el comptador intern.

Tenint en compte que el processos productors executaran sempre l'operació d'alliberament (incrementant el comptador intern) i els processos consumidors demanaran accés executant la operació de bloqueig (que disminuirà el comptador intern), és fàcil veure que el valor del comptador serà sempre igual a la quantitat de dades que els productors ha generat sense que els consumidors hagin encara de consumit. Així podem deduir que si en algun moment el valor arriba al valor zero, significarà que no hi han dades a consumir i per tant farem que el semàfor es trobi sempre bloquejat en aquest cas, però es desbloquegi així que el comptador incrementi el seu valor.

A més de resoldre problemes de tipus productor-consumidor o lector-escritor, podem fer servir semàfors per gestionar problemes de sincronització on un procés hagi d'activar l'execució d'un altre o d'exclusió mútua assegurant que només un procés aconseguirà accedir a la secció crítica perquè el semàfor restarà bloquejat fina a la sortida.

Així si volem sincronitzar dos processos fent que un d'ells (p1) executi una acció sempre abans que l'altra (p2), usant un semàfor, l'inicialitzarem a 0 per assegurar que es troba bloquejat. La codificació del procés p2 assegura que abans de

qualsevol crida a l'acció que volem controlar, sol·licitarà accés al semàfor amb un `sendWait`. Per contra, a la codificació del procés p1 caldrà situar sempre una crida a `sendSignal` just després d'executar l'acció a controlar (veure taula 1.2).

D'aquesta forma ens assegurarem que el procés p1 executarà l'acció sempre abans que p2. Com a exemple, imaginem dos processos. Un ha d'escriure *Hola*, (procés p1) i el procés p2 ha d'escriure *món*. L'ordre correcte d'execució és primer p1 i després p2, per aconseguir escriure *Hola món*. En cas que el procés P1 s'executi abans que p2, cap problema: escriu *Hola* i fa un `signal` al semàfor (semàfor=1). Quan el procés p2 s'executa trobarà semàfor=1, per tant no quedarà bloquejat, podrà fer un `sendWait` al semàfor (semàfor=0) i escriurà *món*.

Taula 1.2.

Procés 1 (p1)	Procés 2 (p2)
initial(0)	
...	
System.out.print("Hola")	sendWait()
sendSignal()	System.out.print("món")

Però què passa si s'executa primer el procés p2? Com que trobarà semàfor a 0, es quedarà bloquejat en fer la petició cridant *sendWait* fins que el procés p1 escrigui *Hola* i faci el *sendSignal*, desbloquejant el semàfor. Llavors p2, que estava bloquejat, s'activarà, posarà el semàfor de nou a vermell (semàfor=0) i escriurà *món* a la pantalla.

Un altre exemple que pot il·lustrar l'ús de semàfors, seria el d'una oficina bancària que gestiona el nostre compte corrent al qual es pot accedir des de diferents oficines per ingressar diners o treure diners. Aquestes dues operacions modifiquen el nostre saldo. Serien dues funcions com les següents:

```
public void ingressar(float diners) {
        float aux;
        aux=llegirSaldo();
        aux=aux+diners;
        saldo=aux;
        guardarSaldo(saldo);
    }

   public void treure(float diners) {
        float aux;
        aux=llegirSaldo();
        aux=aux-diners;
        saldo=aux;
        guardarSaldo(saldo);
    }
```

El problema ve quan de forma simultània es vol fer un ingrés i es vol treure diners. Si per una banda estem traient diners del compte corrent i per una altra banda algú està fent un ingrés, es podria crear una situació anòmala. Hi haurà dos processos concurrents, un traurà diners i l'altre n'ingressarà. Si accedeixen al mateix temps a `llegirSaldo()` els dos agafen el mateix valor, imaginem 100€. El procés que vol ingressar diners, ho vol fer amb la quantitat de 300€. I el que vol treure'n, en vol 30€.

Si continuem l'execució dels dos processos, depenent de quin sigui l'ordre d'execució de les instruccions, ens podem trobar amb un saldo diferent. Si després de llegir el saldo, el procés d'ingrés acaba l'execució i guarda el saldo, guardaria 400 € (100 € + 300 €). Però posteriorment acabaria el procés de treure diners i guardaria a saldo el valor de 70 € (100 € - 30 €). Hem perdut l'ingrés. Hi ha dos processos que s'estan executant a una secció crítica que hauríem de protegir en exclusió mútua. Únicament un procés ha de poder accedir a aquesta secció crítica i poder modificar la variable compartida *saldo*.

Per evitar el problema podem utilitzar un semàfor. L'iniciarem a 1, indicant el número de processos que podran entrar a la secció crítica. I tant en el procés de treure dines com en el d'ingressar-ne afegirem un `sendWait()` a l'inici de les seccions crítiques i un `sendSignal()` al final.

```
public void ingressar(float diners) {
          sendWait();
          float aux;
          aux=llegirSaldo();
          aux=aux+diners;
          saldo=aux;
          guardarSaldo(saldo);
          sendSignal();
  }

    public void treure(float diners) {
          sendWait();
          float aux;
          aux=llegirSaldo();
          aux=aux-diners;
          saldo=aux;
          guardarSaldo(saldo);
          sendSignal();
  }
```

D'aquesta forma quan un procés entra a la secció crítica d'un mètode, agafa el semàfor. Si és 1, podrà fer el `sendWait`, per tant el semàfor es posarà a 0, tancat. I cap altre procés no podrà entrar a cap dels dos mètodes. Si un procés intenta entrar, trobarà el semàfor a 0 i quedarà bloquejat fins que el procés que té el semàfor faci un `sendSignal`, posi el semàfor a 1 i alliberi el semàfor.

Utilitzar semàfors és una forma eficient de sincronitzar processos concurrents. Resol de forma simple l'exclusió mútua. Però des del punt de vista de la programació, els algoritmes són complicats de dissenyar i d'entendre, ja que les operacions de sincronització poden estar disperses pel codi. Per tant es poden cometre errors amb facilitat.

Monitors

Una altra forma de resoldre la sincronització de processos és l'ús de monitors. Els monitors són un conjunt de procediments encapsulats que ens proporcionen l'accés a recursos compartits a través de diversos processos en exclusió mútua.

Les operacions del monitor estan encapsulades dins d'un mòdul per protegir-les del programador. Únicament un procés pot estar en execució dins d'aquest mòdul.

El grau de seguretat és alt ja que els processos no saben com estan implementats aquests mòduls. El programador no sap com i en quin moment es criden les operacions del mòdul, així és més robust. Un monitor, un cop implementat, si funciona correctament sempre funcionarà bé.

Un monitor es pot veure com una habitació, tancada amb una porta, que té a dins els recursos. Els processos que vulguin utilitzar aquests recursos han d'entrar a l'habitació, però amb les condicions que marca el monitor i únicament un procés a la vegada. La resta que vulgui fer ús dels recursos haurà d'espera que surti el que és dins.

Per la seva encapsulació, l'única acció que ha de prendre el programador del procés que vulgui accedir al recurs protegit és informar al monitor. L'exclusió mútua hi és implícita. Els semàfors, en canvi, s'han d'implementar amb una seqüència correcta de senyal i esperar per tal de no bloquejar el sistema.

Un **monitor** és un algoritme que fa una abstracció de dades que ens permet representar de forma abstracta un recurs compartit mitjançant un conjunt de variables que defineixen el seu estat. L'accés a aquestes variables únicament és possible des d'uns mètodes del monitor.

Els monitor ha de poder incorporar un mecanisme de sincronització. Per tant, s'han d'implementar. Es pot fer ús de senyals. Aquests senyals s'utilitzen per impedir els bloquejos. Si el procés que és en el monitor ha d'esperar un senyal, es posa en estat d'espera o bloquejat fora del monitor, permetent que un altre procés faci ús del monitor. Els processos que són fora del monitor, estan a l'espera d'una condició o senyal per tornar a entrar.

Aquestes variables que s'utilitzen pels senyals i són utilitzades pel monitor per la sincronització s'anomenen **variables de condició**. Aquestes poden ser manipulades amb operacions de `sendSignal` i `sendWait` (com els semàfors).

- *sendWait*: un procés que està esperant a un esdeveniment indicat per una variable de condició abandona de forma temporal el monitor i es posa a la cua que correspon a la seva variable de condició.
- *sendSignal*: desbloqueja un procés de la cua de processos bloquejats amb la variable de condició indicada i es posa en estat preparat per entrar al monitor. El procés que entra no ha de ser el que més temps porta esperant, però s'ha de garantir que el temps d'espera d'un procés sigui limitat. Si no existeix cap procés a la cua, l'operació *sendSignal* no té efecte, i el primer procés que demani l'ús del monitor entrarà.

Un monitor consta de 4 elements:

- Variables o mètodes permanents o privats: són les variables i mètodes interns al monitor que només són accessibles des de dins del monitor. No es modifiquen entre dues crides consecutives al monitor.

- Codi d'inicialització: inicialitza les variables permanents, s'executa quan el monitor és creat.
- Mètodes externs o exportats: són mètodes que són accessibles des de fora del monitor pels processos que volen entrar a fer-ne ús.
- Cua de processos: és la cua de processos bloquejats a l'espera del senyal que els alliberi per tornar a entrar al monitor.

wait, *notify* i *notifyAll*, són operacions que permeten sincronitzar el monitor.

En el camp de la programació un monitor és un objecte en el qual tots els seus mètodes estan implementats sota exclusió mútua. En el llenguatge Java són objectes d'una classe en els quals tots els seus mètodes públics són *synchronized*.

Un semàfor és un objecte que permet sincronitzar l'accés a un recurs compartit i un monitor és una interfície d'accés al recurs compartit. Són l'encapsulament d'un objecte, per allò que fa un objecte més segur, robust i escalable.

Sincronitzar a Java

A Java l'execució d'objectes no està protegida. Si volem aplicar exclusió mútua a Java hem d'utilitzar la paraula reservada *synchronized* a la declaració d'atributs, de mètodes o en un bloc de codi dins d'un mètode.

Tots els mètodes amb el modificador *synchronized* s'executaran en exclusió mútua. El modificador assegura que si s'està executant un mètode sincronitzat cap altre mètode sincronitzat es pugui executar. Però els mètodes no sincronitzats poden estar executant-se i amb més d'un procés a la vegada. A més, únicament el mètode sincronitzat pot estar executat per un procés, la resta de processos que volen executar el mètode s'hauran d'esperar que acabi el procés que l'està executant.

Sincronitzar mètodes a Java

```
1   public synchronized  void metodeSincronitzat {
2       //part de codi sincronitzat
3   }
```

Si no ens interessa sincronitzar tot el mètode, sinó únicament una part del codi, utilitzarem la paraula reservada *synchronized* sobre el mateix objecte que ha cridat el mètode en el qual es troba la part del codi que s'ha de sincronitzar.

Sincronitzar blocs de codi a Java

```
1 public  void metodeNoSincronitzat {
2     //part de codi sense sincronitzar
3     synchronized(this){
4       //part de codi sincronitzat
5     }
6
7     //part de codi sense sincronitzar
8   }
```

`this` és l'objecte que ha cridat al mètode, però també podem utilitzar un altre objecte `synchroniced(objecte)`, si el que volem és realitzar més d'una operació atòmica sobre un objecte.

Java, per tant, ens proporciona amb *synchronized* els recursos que ens donaria la implementació d'un semàfor o d'un monitor. Les seves llibreries ens proporcionen un accés en exclusió mútua i controlen els errors de bloqueig o interbloqueig que es pugui ocasionar entre fils en execució.

Atomicitat d'una operació i classes atòmiques de Java

L'únic àrbitre que tenim en la gestió de processos és el planificador. S'encarrega de decidir quin procés fa ús del processador. Després les instruccions o operacions que formen el procés es van executant al processador seguint un ordre concret.

Una operació o un grup d'operacions s'han de poder executar com si fossin una única instrucció. A més, no es poden executar de forma concurrent amb qualsevol altra operació en la qual hi hagi involucrada una dada o recurs que utilitza la primera operació.

L'atomicitat d'una operació és poder garantir que no es poden executar dues operacions de forma concurrent si fan ús d'un recurs compartit fins que una de les dues deixa lliure aquest recurs. Una operació atòmica únicament ha d'observar dos estats: l'inicial i el resultat. Una operació atòmica, o s'executa completament o no ho fa en absolut.

L'**atomicitat real** són les instruccions que executen a codi màquina. En canvi, l'**atomicitat model** és un grup d'instrucció, a codi màquina, que s'executa de forma atòmica.

Classes d'operacions atòmiques

La instrucció `x++;` o `x= x + 1;` té la següent seqüència d'instruccions atòmiques reals:

1. Llegir valor x
2. Afegir a x1
3. Guardar valor x

L'atomicitat de model entén la instrucció completa, `x = x + 1` com a atòmica.

Dos processos executen en paral·lel el següent exemple:

```
package operacionsatomiques;
public class OperacionsAtomiques {
  public static void main(String[] args) {
    int x=0;
    x=x+1;
    System.out.println(x);
  }
}
```

Si tenim en compte que els dos processos executen en paral·lel la instrucció `x=x+1;` en l'atomicitat real una ordenació d'execució seria la següent (taula 1.3):

Taula 1.3.

1	int x=0;	
	Procés 1 (x=x+1)	**Procés 2(x=x+1)**

2	Llegir valor x		
3			Llegir valor x
4	Afegir a x 1		
5			Afegir a x 1
6	Guardar valor x		
7			Guardar valor x
8		System.out.printl(x);	

Depenent de l'ordre d'execució de les operacions atòmiques d'un procés i l'altre el resultat o el valor d'x pot variar.

Classes atòmiques de Java

Les classes atòmiques de Java són una primera solució als problemes de processos concurrents. Permeten executar algunes operacions com si fossin atòmiques reals.

Una operació atòmica té únicament dos estats: l'inicial i el resultat, és a dir, es completa sense interrupcions des de l'inici fins al final.

Volatile és un modificador, utilitzat en la programació concurrent, que es pot aplicar a alguns tipus primitius de variables. Indica que el valor de la variable quedarà guardat a la memòria i no a un registre del processador. Així, és accessible per un altre procés o fil per modificar-la. Per exemple: `volatile int c;`

A Java, l'accés a les variables de tipus primitius (excepte *double* i *long*) és atòmic. Com que també és atòmic l'accés a totes les variables de tipus primitiu en les quals apliquem el modificador *volatile*.

Java assegura l'atomicitat de l'accés a les variables **volàtils** o primitives, però no a la seves operacions. Això vol dir que l'operació `x++;` en la qual `x` és un `int`, no és una operació atòmica i per tant no és un fil segur.

Java incorpora, des de la versió 1.5, al paquet java.util.concurrent.atomic, les classes que ens permeten convertir en atòmiques (fils segurs) algunes operacions, com ara augmentar, reduir, actualitzar i afegir un valor. Totes les classes tenen mètodes *get* i *set* que també són atòmiques.

Les classes que incorpora són: `AtomicBoolean`, `AtomicInteger`, `AtomicIntegerArray`, `AtomicLong`, `AtomicLongArray` i actualitzadors que són bàsicament derivats d'una variable tipus *volatile* i són `AtomicIntegerFieldUpdater`, `AtomicLongFieldUpdater` i `AtomicReferenceFieldUpdater`.

Per tant, podem convertir l'operació `x++` o `x=x+1` en una operació atòmica .

Classe Comptador sense operacions atòmiques

```
1 public class Comptador {
2     private int x = 0;
3     public void augmenta() {
4         x++;
5     }
6     public void disminueix() {
7         x--;
8     }
9     public int getValorComptador() {
```

```
10          return x;
11      }
12    }
```

Per convertir a operacions atòmiques l'exemple anterior, utilitzarem els mètodes de la classe `AtomicInteger` del paquet java.util.concurrent.atomic.AtomicInteger.

Classe contador utilitzant classes atòmiques

```
1
2  import java.util.concurrent.atomic.AtomicInteger;
3      public class ContadorAtomic {
4        private AtomicInteger x =new AtomicInteger(0);
5        public void augmenta() {
6          x.incrementAndGet();
7        }
8        public void disminueix() {
9          x.decrementAndGet();
10       }
11       public int getValorContadorAtomic() {
12         return x.get();
13       }
14   }
```

A la documentació oficial podeu trobar més informació sobre aquest paquet. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html

Col·leccions concurrents

Les col·leccions són objectes que contenen múltiples dades d'un mateix tipus. Les col·leccions poden estar involucrades també en l'execució de processos concurrents i ser utilitzades per diferents processos de manera que s'acabin compartint també les dades contingudes a la col·lecció.

A Java per treballar amb col·leccions hem d'utilitzar el Framework Collections en el qual es troben els paquets java.util i java.util.concurrent, que contenen les classes i interfícies que ens permeten crear col·leccions.

A Java les interfície List<E>, Set<E> i Queue<E> serien les col·leccions més importants. Representen llistes, conjunts i cues. Map<E> és també una col·lecció de *key*/valor. Cadascuna amb les seves característiques d'ordenació, els mètodes d'inserció, d'extracció i consulta, si permeten elements repetits o no, etc.

El problema amb aquestes col·leccions és que en la programació concurrent en la qual siguin compartides per diferents processos o fils de forma simultània, podrien donar resultats inesperats.

Alguns llenguatges de programació van solucionar aquest problema sincronitzant les classes. A Java s'utilitza, per exemple, Collections.synchronizedList(List<T> list) sobre la interfície List<E> per poder sincronitzar la classe que la implementa, la classe `ArrayList`. Altres classes que implementen la interfície List és `Vector`. Java ja sincronitza aquesta classe.

La classe `CopyOnWriteArrayList` cada cop que es modifica l'*array* es crea en la memòria un *array* amb el contingut sense processos de sincronització. Això fa que les consultes no siguin tan costoses.

La cua FIFO (en anglès, *First In Firts Out*) és una estructura de dades implementada de forma que els primers elements en entrar a l'estructura són els primers en sortir-ne.

En la versió 1.5 va aparèixer java.util.concurrent que incorpora classes que ens permeten solucionar problemes de concurrència d'una forma simple.

`ArrayList` és una col·lecció de dades i si volem que sigui utilitzada per diferents fils d'execució el que hem de fer és sincronitzar totes les operacions de lectura i escriptura per preservar la integritat de les dades. Si aquest `ArrayList` és molt consultat i poc actualitzat, la sincronització penalitza molt l'accés. Java crea la classe `CopyOnWriteArrayList`, que és l'execució de fil segur d'una `ArrayList`.

La interfície **BlockingQueue** implementa una cua FIFO, que bloqueja el fil d'execució si intenta treure de la cua un element i la cua està buida o si intenta inserir un element i està plena. També conté mètodes que fan que el bloqueig duri un temps determinat abans de donar un error d'inserció o lectura i així evitar una execució infinita.

La classe `SynchronousQueue` és una cua de bloqueig, implementa en BlockingQueue. El que fa és que per cada operació d'inserció ha d'esperar una d'eliminació i per cada inserció ha d'espera una eliminació.

`ConcurrentMap` és una altra classe que defineix una taula *hash* amb operacions atòmiques.

Podeu trobar més informació sobre el paquet a la plana oficial de Java. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html

Comunicació amb missatges

Una solució als problemes de concurrència de processos, molt adequada sobretot pels sistemes distribuïts, són els missatges. Recordem que un sistema distribuït no comparteix memòria, per tant no és possible la comunicació entre processos amb variables compartides.

Aquest sistema de sincronització i comunicació està inclòs en la majoria de sistemes operatius i es fa servir tant per la comunicació entre diferents ordinadors, com per comunicar entre processos d'una mateix ordinador. La comunicació a través de missatges sempre necessita un procés emissor i un altre de receptor per poder intercanviar informació. Per tant, les operacions bàsiques seran enviar(missatge) i rebre(missatge).

Donat que en aquesta comunicació poden estar implicats diversos processos, cal permetre a les operacions bàsiques identificar qui és el destinatari i fins i tot convindria en ocasions identificar l'emissor. Caldrà estendre les operacions bàsiques incorporant a qui va dirigit el missatge o com el receptor podria indicar de qui espera el missatge.

Segons la forma de referenciar el destinatari i l'emissor, parlarem d'enviament de missatges en **comunicació directa** o **comunicació indirecta**.

En la **comunicació directa**, l'emissor identifica el receptor explícitament quan envia un missatge i viceversa, el receptor identifica explícitament l'emissor que li envia el missatge. Es crea un enllaç de comunicació entre el procés emissor i

procés receptor. Les ordres d'enviament i recepció serien:

enviar(A,missatge)->Enviar un missatge al procés A

rebre(B,misstage)->Rebre un missatge del procés B

Aquest sistema de comunicació ofereix una gran seguretat ja que identifica els processos que actuen en la comunicació i no introdueix cap retard a l'hora d'identificar els processos. Per altra banda això també és un desavantatge, ja que qualsevol canvi en la identificació de processos implica un canvi en la codificació de les ordres de comunicació.

La comunicació directa només és possible d'implementar si podem identificar els processos. És a dir caldrà que el sistema operatiu i el llenguatge de programació suportin la gestió de processos. També cal que els processos s'executin en un mateix ordinador, ja que sinó és impossible accedir-hi de forma directa.

Quan els processos s'executin en diferents ordinadors, serà necessari interposar entre ells un sistema de missatgeria que enviï i reculli els missatges amb independència de quin sigui el procés emissor i receptor. Només caldrà que ambdós processos tinguin accés al mateix sistema d'intercanvi de missatges i que aquest permeti identificar d'alguna manera l'emissor i el receptor. Cal adonar-se que es tracta identificació interna del sistema de missatgeria. En cap cas són referencies directes als processos.

En la **comunicació indirecta** es desconeixen les referències als processos emissor i receptor del missatge. És necessària l'existència d'un sistema de missatgeria capaç de distribuir missatges identificats amb algun valor específic i únic, i permetre als processos la consulta dels missatges a partir d'aquest identificador per destriar els vagin dirigits a ells. En certa forma és com vehicular la comunicació a través d'una bústia comuna (l'identificador). L'emissor envia el missatge a una bústia específica i el receptor recull el missatge de la mateixa bústia.

Les ordres serien de la següent forma:

enviar(BústiaA,missatge)→ El procés emissor deixa el missatge a la bústia A

rebre(BústiaA,missatge)→ El procés receptor recull el missatge de la bústia A

Aquest sistema és més flexible que la comunicació directa ja que ens permet tenir més d'un enllaç de comunicacions entre els dos processos i podem crear enllaços de tipus *un a molts*, *molts a un* i *molts a molts*. En els enllaços de comunicació *un a molts* com ara els sistemes client/servidor les bústies s'anomenen **ports**. Aquests múltiples enllaços no es podrien fer amb el sistema de comunicació directa. El sistema operatiu és l'encarregat de realitzar l'associació de les bústies als processos i ho pot fer de forma dinàmica o estàtica.

D'altra banda, si tenim en compte la immediatesa en la recepció dels missatges entre processos parlarem de: comunicació **síncrona** o **asíncrona**.

En la **comunicació síncrona** el procés que envia un missatge a un altre procés es queda a l'espera de rebre resposta. Per tant, l'emissor queda bloquejat fins que rep aquesta resposta. I viceversa. En canvi en la **comunicació asíncrona** el procés que

envia el missatge continua la seva execució sense preocupar-se de si el missatge ha estat rebut o no. Per tant aquí haurà d'existir una memòria intermèdia o que es vagin acumulant els missatges. El problema vindrà quan la memòria intermèdia es trobi plena. El procés que envia el missatge queda bloquejat fins que troba l'espai dins la memòria intermèdia.

Un símil a aquests dos tipus de sincronització podria ser l'enviament de missatges a través de correu electrònic o de xat. El primer, que va deixant els missatges a una bústia sense preocupar-se si els receptor el rep o no, seria una comunicació asíncrona. En canvi en un xat la comunicació és totalment síncrona ja que l'emissor i el receptor han de coincidir en el temps.

Implementació usant Java

La comunicació i sincronització entre processos d'ordinadors diferents connectats en xarxa (sistemes distribuïts) requereixen d'una mecanisme de transferència que assumeixi la comunicació indirecta, els sòcols o *sockets*. Es tracta d'unes biblioteques que disposen de tots els mecanismes necessaris per enviar i rebre missatges a través de la xarxa.

Dins del mateix ordinador, Java disposa de mètodes com `wait()`, `notify()` i `notifyAll()`, que modifiquen l'estat d'un fil aturant o activant l'execució dels processos referenciats. Aquests mètodes han de ser invocats sempre dins d'un bloc sincronitzat (amb el modificador `synchronize`).

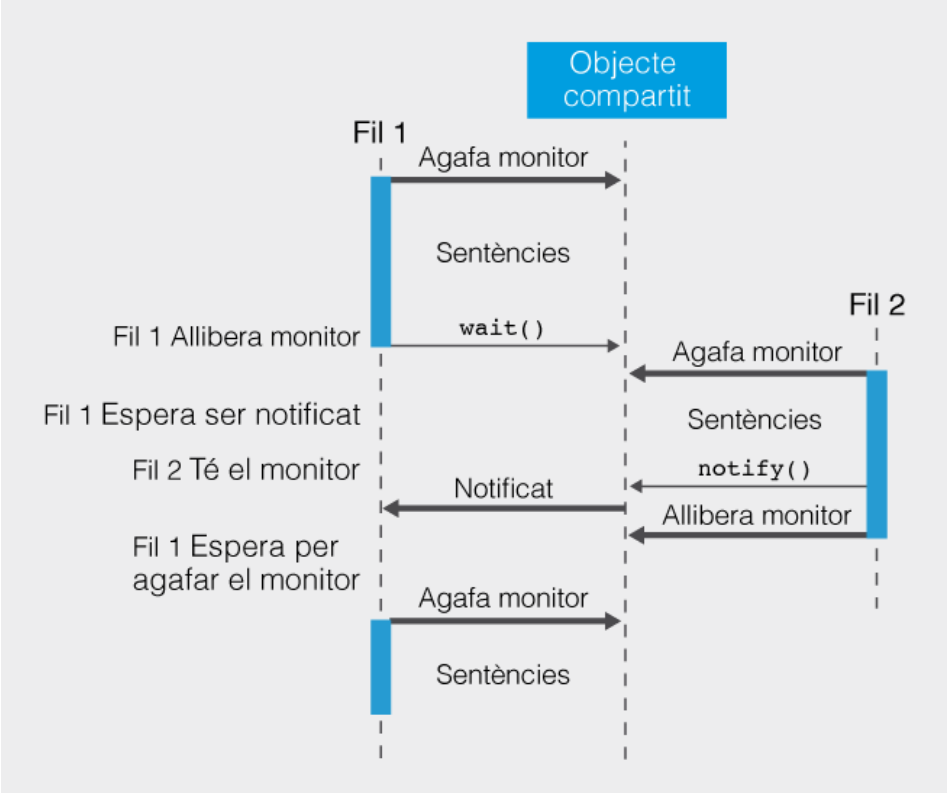
`wait()`: posa el fil que s'està executant en estat d'espera. Aquest fil abandona la zona d'exclusió mútua i espera, en una cua d'espera, a ser tornat a a activar per un mètode `notify()` o `notifyAll()`.

`notify()`: un fil de la cua d'espera passa a l'estat de preparat.

`notifyAll()`: tots els fils de la cua d'espera passen a l'estat de preparat.

Quan un procés entra en la secció crítica, és a dir que controla el monitor, cap altre fil pot entrar a la secció crítica del mateix objecte que està controlat pel mateix monitor. Per poder coordinar-se i comunicar-se es faran ús del mètodes anteriors. Quan no es donen les condicions per continuar amb l'execució d'un fil que es troba a la secció crítica, aquest pot abandonar la secció i permetre que un altre fil que es troba en espera pugui agafar el monitor i entrar a la secció crítica. En la següent figura (figura 1.15) es mostra com dos fils volen accedir a un recurs comú i es comuniquen per sincronitzar-se. Imaginem que el recurs compartit és una targeta SIM d'un telèfon mòbil. El fil 1 vol accedir a la targeta per fer una trucada, agafa el monitor del recurs compartit per tenir accés exclusiu a la targeta, però la SIM no està activada encara. La condició per poder executar-se no es compleix. Per tant, executa el mètode `wait()` que el manté en espera fins que la condició es compleixi. Qualsevol altre procés que vulgui fer la trucada i agafi el monitor farà el mateix, executarà el mètode `wait()` i es quedarà a la cua d'espera.

Figura 1.15. wait() i notify()



El fil 2 és un procés que s'encarrega d'activar la targeta SIM. Quan agafa el monitor té accés amb exclusió mútua a la targeta. Fa les sentències per activar la SIM i executa `notify()`. Aquesta sentència notifica al primer fil, que està a la cua de processos en espera, que ja es compleixen les condicions per poder continuar l'execució dels processos de trucades i allibera el monitor. La diferència entre `notify()` i `notifyAll()` és que la notificació es fa a un fil en espera o a tots. Per últim, el fil 1 agafa el monitor i executa la seves sentències de trucada.

A Java, fent ús de ***synchronized*** es creen semàfors. Si hi afegim els mètodes `wait()`, `notify()` o `notifyAll()` crearem monitors fàcilment i de forma eficient.

2. Programació multifil

La programació multifil permet dur a terme diferents fils d'execució a la vegada, és a dir, permet realitzar diferents tasques en una aplicació de forma concurrent. La majoria de llenguatges de programació permeten treballar amb fils i realitzar programació multifil. També són multifil la majoria de les aplicacions que fem servir als nostres ordinadors (editors de text, navegadors, editors gràfics...), fet que ens dóna la sensació de més agilitat d'execució.

En un editor de text, per exemple, un fil pot estar controlant l'ortografia del text, un altre pot estar atent a les pulsacions sobre les icones de la interfície gràfica i l'altre guardant el document.

2.1 Els fils

La programació tradicional està dissenyada per treballar de forma seqüencial, de manera que quan acaba un procés se'n posa en marxa un altre. Els fils són una forma d'executar paral·lelament diferents parts d'un mateix programa. A Java els fils són coneguts com a *threads*.

2.1.1 Definició de fil

> Un fil o *thread* és la unitat de processament més petita que pot ser planificada per un sistema operatiu.

El fils ens permeten executar concurrentment diferents tasques amb una despesa mínima de recursos ja que tots els fils d'un mateix procés comparteixen l'espai de memòria i la seva creació consumeix poc temps de processador. Són **entitats lleugeres**.

De la mateixa manera que un sistema operatiu permet executar diferents processos a la vegada per concurrència o paral·lelisme, dins d'un procés hi haurà un o diversos fils en execució.

Els fils representen exclusivament les execucions de les instruccions d'un programa que es duen a terme simultàniament en el context d'un mateix procés. És a dir, compartint l'accés a la mateixa zona de memòria assignada al procés al que pertanyen. Cada procés conté com a mínim un fil, malgrat que en pot arribar a contenir molts.

2.1.2 Relació procés i fil

Recordeu que els processos no comparteixen memòria entre ells, són independents, porten informació sobre el seu estat i interactuen amb altres processos a través de mecanismes de comunicació donats pel sistema. Aquesta comunicació o el canvi d'estat d'un procés són mecanismes costosos pel microprocessador. És per això que s'anomenen entitats pesades. Els fils, en canvi, comparteixen recursos. El canvi d'estat i d'execució es realitza molt més ràpidament i poden comunicar-se entre ells fent servir les dades de la memòria que comparteixen. El temps d'ús del microprocessador dedicat a la gestió de fils és força menyspreable.

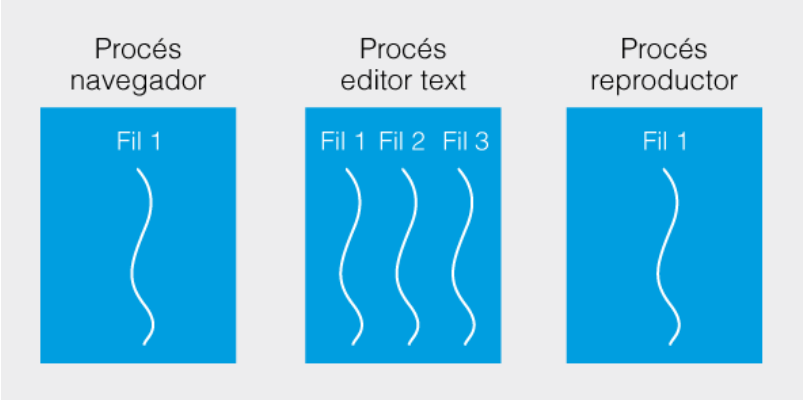
El fet que els fils d'execució d'un mateix procés comparteixin els recursos fa que qualsevol fil pugui modificar-los. Si un fil modifica una dada en la memòria, la resta dels fils tenen accés a la nova dada de forma immediata. Quan hi ha un procés executant-se, com a mínim sempre hi ha un fil en execució. Parlem de processos multifil quan s'executen diversos fils concurrentment, realitzant diferents tasques i col·laborant entre ells.

Quan un procés finalitza, tots els seus fils també ho fan. De forma equivalent, quan finalitzen tots els fils d'un procés, el procés també acaba i tots els seus recursos són alliberats.

Els sistemes operatius actuals admeten la concurrència, és a dir, poden executar diferents processos a la vegada. Podem estar navegant per Internet amb un navegador, escoltant música amb un reproductor i, amb un editor de text, redactant un document. Cada aplicació que s'executa és un procés i cada un d'aquests processos té com a mínim un fil d'execució, però en podrien tenir més d'un. L'editor de text podria tenir un fil que s'encarregués de veure quina tecla estem picant, un altre que vagi comprovant l'ortografia i un altre que de tant en tant s'activi per guardar el document. Cal veure que un fil pertany a un procés i que els diferents fils poden fer tasques diferents dins del seu procés.

A la figura 2.1 veiem un símil dels tres processos anteriors amb els seus fils d'execució.

Figura 2.1. Diferents processos amb fils

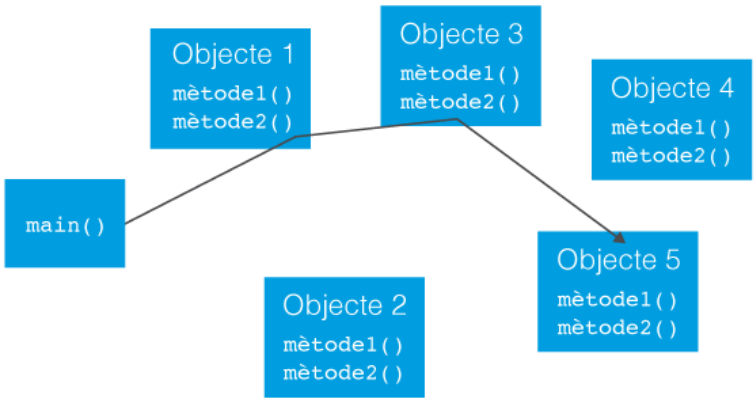


2.1.3 El fils a Java

Quan comencem un programa en Java hi ha un fil en execució, el **fil principal**, que és creat pel mètode `main()`. Aquest fil és important ja que és l'encarregat, si cal, de crear la resta de fils del programa. S'ha de programar l'aplicació perquè el fil principal sigui l'últim en acabar la seva execució. Això s'aconsegueix fent esperar els fils creats pel fil principal que aquest últim finalitzi.

Si no es creen més fils que el principal, només n'existeix un i serà l'encarregat de fer les crides als mètodes i crear el objectes que indiqui el programa. A la figura 2.2 podem veure com el fil principal va executant seqüencialment els mètodes dels diferents objectes que el programa li va indicant.

Figura 2.2. Un únic fil d'execució

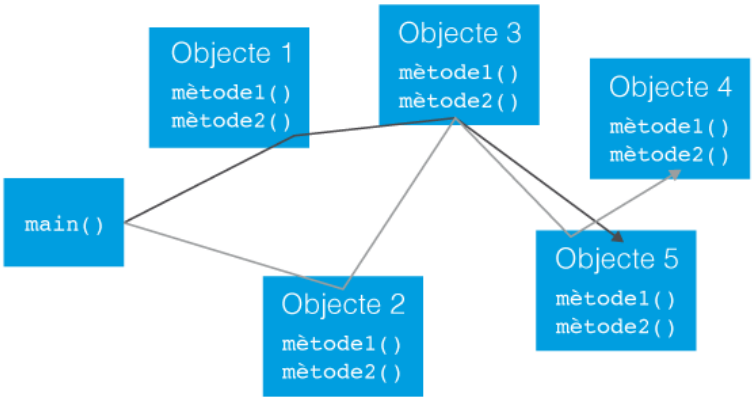



```
package filexecuciounic;
public class FilExecucioUnic {

    public static void main(String[] args) {
       for (int x=0;x<10;x++)
            System.out.println("x=" + x);
    }
}
```

En l'exemple anterior, quan s'executa el `main()`, s'imprimeixen els diferents valors d' `x`. Això passa en un únic fil d'execució.

Però des del fil principal es poden crear altres fils que aniran executant altres mètodes dels objectes instanciats o mètodes nous que calgui crear, de forma simultània. A la figura 2.3 veiem com, des del fil principal, es crea un altre fil que va seguint un ordre d'execució diferent de l'execució del principal i com, en un moment donat, executen de forma simultània el mateix mètode del mateix objecte.

Figura 2.3. Dos fils d'execució



Un fil creat pel `main()` pot crear altres fils, que a la vegada poden crear nous fils. Els fils creats depenen directament dels seus creadors, d'aquesta manera es creen jerarquies de pares i fills de fils d'execució.

Java defineix la funcionalitat principal de la gestió dels fils a la classe `Thread`. Gràcies a la classe `Thread` podem crear i executar nous fils, aturar-los, reprendre'n l'execució, esperar la finalització dels fils creats, etc. La funcionalitat de `Thread` es complementa amb la funcionalitat específica de la classe `object` destinada a resoldre la sincronització entre fils. La sincronització s'encapsula dins la classe `Object` per tal que qualsevol execució disposi sempre dels mètodes i mecanismes necessaris per dur-la a terme.

L'herència ens permetrà, de forma fàcil, incorporar la funcionalitat de la classe `Thread` a les nostres implementacions. Tot i així, a causa que Java no suporta l'herència múltiple, el llenguatge Java ens ofereix una forma alternativa d'incorporar la funcionalitat de `Thread` a les nostres implementacions fent servir la interfície `Runnable`. Veiem l'especificació dels principals constructors i mètodes d'aquesta classe.

La classe Thread

Constructors:

- `Thread()`: reserva memòria per crear un objecte `Thread`.
- `Thread(Runnable target)`: reserva memòria per la creació d'un nou objecte de la classe `Thread` el qual gestionarà l'execució de l'objecte Runnable passat per paràmetre.
- `Thread(Runnable target, String name)`: reserva memòria per la cre-

ació d'un nou objecte de la classe `Thread` el qual gestionarà l'execució de l'objecte Runnable passat per paràmetre. L'objecte creat es podrà identificar amb el nom passat en el segon paràmetre.

- `Thread(String name)`: reserva memòria per crear un objecte `Thread` identificat amb el nom passat per paràmetre.
- `Thread(ThreadGroup group, Runnable target)`: reserva memòria per un nou objecte `Thread` que gestionarà l'execució de l'objecte Runnable (*target*) i que pertany al grup de fils (*group*).
- `Thread(ThreadGroup group, Runnable target, String name)`: reserva memòria per un nou objecte `Thread` identificat amb el nom *name*, que pertany al grup de fils (*group*) i que gestionarà l'execució de l'objecte `Runnable` *target*.
- `Thread(ThreadGroup group, String name)`: crea un nou objecte `Thread` identificat per *name* i que pertany al grup de fils concret (*group*) .

Mètodes més importants:

- `static Thread currentThread()`: ens retorna el fil que s'està executant.
- `string getName()`: retorna el nom del fil.
- `void setName(String nom)`: assigna un nom al fil.
- `int getPriority()`: ens diu la prioritat d'execució del fil.
- `void setPriority(int Prioritat)`: assigna la prioritat d'execució del fil.
- `ThreadGroup getThreadGroup()`: retorna el grup de fils al qual pertany el fil.
- `boolean isAlive()`: mira si el fil està actiu. És a dir, si hi ha hagut una crida inicial al seu mètode `start()` i encara no ha finalitzat l'execució.
- `boolean isDaemon()`: ens indica si el fil és un dimoni o no.
- `void setDaemon(boolean on)`: indica si el fil serà un dimoni o un fil d'usuari. Els dimonis estan lligats al seu creador, si el seu creador acaba la seva execució, els *fills* dimonis també finalitzen.
- `void join()`: Atura l'execució del fil que fa la crida i espera la finalització del fil invocat.
- `void join(long Millis)`: espera la finalització del fil invocat, però un cop passat el temps del paràmetre (expressat en mil·lisegons), es continuarà l'execució.
- `void run()`: Representa l'inici d'execució del fil. Aquesta execució comença immediatament després d'haver cridat el mètode `star()`.

- `static void sleep(long Millis)`: atura el fil invocat, durant el nombre de mil·lisegons indicats al paràmetre.
- `void start()`: posa un fil en estat de preparat per executar-se.
- `static void yield()`: fa que el fil que s'està executant passi a estat de preparat. Surt del processador.
- `static void sleep(long Millis)`: fa que un fil que s'està executant s'adormi durant un temps, que s'estableix en mil·lisegons. El fil no perd cap propietat de bloqueig.
- `String toString()`: retorna una cadena que representa el fil, nom prioritat i el seu grup.

Podeu trobar més informació sobre mètodes, propietats i constructors de la classe `Thread` a:

http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.html

La interfície Runnable

En el llenguatge Java, haver d'heretar de la classe Thread per crear classes amb aquesta funcionalitat comporta un problema no pas menor, ja que Java no suporta l'herència múltiple i per tant ens resultarà impossible fer que la nova classe n'estengui cap altre que no sigui Thread. És a dir, ens quedarà molt limitada la implementació de classes fil, usant herència. Per evitar aquesta limitació podem fer servir la Interfície Runnable. La interfície Runnable força l'existència del mètode *run* però no limita l'herència. De fet els objectes `Runnable` no tenen la capacitat d'executar-se com un fil independent. Per això cal fer servir un objecte `Thread` per gestionar la seva execució.

La classe Object

La classe `Object` és molt extensa. Aquí únicament ens centrarem en els mètodes que ens proporciona, relacionats amb els fils. Són mètodes que ens permeten la comunicació i manipulació de fils. Aquesta classe, jeràrquicament, es troba a la part superior de totes les classes de Java, per tant cada classe de Java hereta la seva funcionalitat. Cada nova classe, doncs, inclou els mètodes següents:

- `wait()`: aquest mètode fa que un fil d'execució passi a un estat d'espera fins que se li notifiqui que pot continuar l'execució.
- `notify()`: és el mètode encarregat de notificar a un fil que es troba en espera que pot continuar l'execució.
- `notifyAll()`: funciona de forma similar a `notify()`, però notifica que poden continuar l'execució tots els fils en espera.

Aquests mètodes són utilitzats quan executem la programació concurrent de diferents fils (programació multifil), i serveixen per fer esperar un fil que un altre acabi de fer alguna tasca. També que, un cop acabada aquesta tasca, notifiqui al fil en espera que pot continuar la seva execució. Aquests mètodes únicament poden ser cridats dins de mètodes sincronitzats:

```
synchronized public void metodeSincronitzat(){
   //...
}
```

O dins de blocs de codi sincronitzats:

```
public void metode(){
   synchronized (this) {
      //....
     }
//...
}
```

2.2 Gestió de fils

Quan un fil és llançat hauríem de poder controlar el seu comportament per comprovar que la nostra aplicació fa el que hauria de fer. És cert que no tenim el control total sobre la forma en què s'executen el fils, però sí que podem controlar moltes coses. Hem de conèixer com es comporta un fil, comprendre el seu cicle de vida, per, a continuació, veure quines són les possibilitats de control que tenim sobre ells.

2.2.1 Creació i execució d'un fil

Existeixen dues formes de creació de fils a Java:

- Heretant la classe `Thread`.
- Implementant la interfície `Runnable`.

Si volem crear una classe en la qual les seves instàncies siguin fils, haurem d'heretar la classe `Thread`. Aquesta té un mètode anomenat `run()` (*public void run()*) que haurem de sobreescriure per codificar les instruccions a processar durant l'execució del fil. Podríem dir que el mètode `run()` és per un fil, el que és el mètode `main()` per un programa.

L'execució del mètode `run` no garantir una execució en un fil independent. Si volem invocar el processament d'un fil caldrà cridar el mètode `start()`. La seva invocació iniciarà l'execució d'un nou fil just a la primera instrucció del mètode `run()`. És a dir la crida d'*start* implica la invocació posterior del mètode `run`.

Al següent exemple crearem una classe que hereta de la classe `Thread`, per tant, les seves instàncies seran fils. El constructor de la classe rep un `String` que serà la paraula que volem imprimir. Al mètode `run()` implementem el codi que permetrà imprimir 5 vegades la paraula.

```
package heretafil;

public class HeretaFil extends Thread {

    String strImprimir;
    public HeretaFil(String strP) {
        strImprimir=strP;
    }

    public void run(){
        for(int x=0;x<5;x++){
            System.out.println(strImprimir+ " " + x);
        }

    }

    public static void main(String[] args) {

        Thread primer = new HeretaFil("Fil 1");
        Thread segon = new HeretaFil("Fil 2");
        // Hem creat dos fils primer i segon, però no s'han executat.
        // Per poder-lo executar s'ha de cridar al mètode start()
        primer.start();
        segon.start();

        System.out.println("Final Fil Principal");

    }
}
```

Les línies següents creen dos objectes tipus `HeretaFil` que són fils:

```
Thread primer = new HeretaFil("Fil 1");
        Thread segon = new HeretaFil("Fil 2");
```

Un cop creats els fils s'han de posar en execució, per això es crida al mètode de la classe `thread`, `start()`.

El mètode `start()` fa una inicialització del fil i el deixa a l'organitzador de fils. Quan l'organitzador de fils ho decideix, es crida al mètode `run()`, que és qui executa el codi del fil.

```
primer.start();
        segon.start();
```

Aquest exemple ha creat dos fils des del fil principal que han executat el mateix codi. Una possible sortida del programa podria ser:

```
run:
Fil 2 0
Fil 1 0
Fil 2 1
Fil 1 1
Final Fil Principal
Fil 2 2
Fil 1 2
Fil 2 3
```

```
Fil 1 3
Fil 2 4
Fil 1 4
BUILD SUCCESSFUL (total time: 0 seconds)
```

Tal com es pot observar, l'ordre es va intercalant. Tenim tres fils d'execució: el principal i els dos creats. Aquesta pot ser una possible sortida, ja que l'execució dels tres fils és independent i pot ser normal que acabi el fil principal abans que els dos creats per ell.

Ja que Java no suporta l'herència múltiple, si volem crear fils en els quals la nostre classe hereti d'alguna altra classe, s'haurà d'utilitzar la interfície `Runnable`. La classe `Thread` conté un constructor que ens permet passar-li com a paràmetre qualsevol objecte que implementi la interfície `Runnable`: *Thread (Runnable unFil)*.

En l'exemple següent es reprodueix el codi anterior però implementant la interfície `Runnable`.

En aquest cas els objectes de la classe `RunnableFil` no són fils ja que no hereten de la classe `Thread`. Si volem que els objectes de la classe `RunnableFil` s'executin com a fils independents haurem de crear un objecte de la classe `Thread` i passar-los com a paràmetres.

```
package runnablefil;

public class RunnableFil implements Runnable {

String strImprimir;
    public RunnableFil(String strP) {
        strImprimir=strP;
    }

    public void run(){
        for(int x=0;x<5;x++){
            System.out.println(strImprimir+ " " + x);
        }

    }

    public static void main(String[] args) {

        //Creem dos objecte de la classe RunnableFil
        RunnableFil objRunnable1 = new RunnableFil("Fil 1");
        RunnableFil objRunnable2 = new RunnableFil("Fil 2");
        //Creem dos Fils i li passem per paràmetres els objecte de la classe
            RunnableFil
        Thread primer = new Thread(objRunnable1);
        Thread segon = new Thread(objRunnable2);
        // Hem creat dos fils primer i segon, però no s'han executat.
        // Per poder-lo executar s'ha de cridar al mètode start()
        primer.start();
        segon.start();

        System.out.println("Final Fil Principal");

    }
}
```

Podem utilitzar l'extensió de la classe `Thread` o bé una implementació de la interfície `Runnable` indistintament per crear fils. Heretar de la classe `Thread`

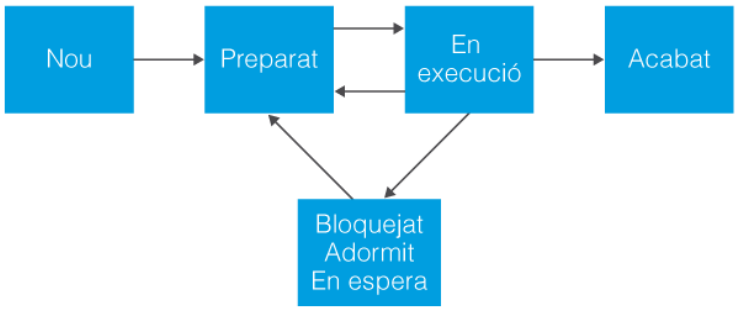
limita les possibilitats de la nostra classe a no poder heretar d'una altra classe en cas que ho necessitem. En canvi, implementar la interfície `Runnable` pot ser una mica més complex, però ens permet heretar d'una altra classe i copiar el seu comportament. És l'opció més utilitzada pels programadors per crear fils.

És possible, però, que el programa principal (el fil principal) finalitzi l'execució abans que els fils creats per ell. Per evitar això, podem utilitzar un mètode de la classe `thread`, `sleep()` que deixa el fil en estat de suspens durant un cert temps. O també utilitzar el mètode `join()` que permet fer que el fil pare esperi que el fils fills acabin.

2.2.2 Estats i canvis d'estat d'un fil

El cicle de vida d'un fil contempla diferents estats pels que pot anar passant durant la seva execució (figura 2.4).

Figura 2.4. Estats d'un fil



Nou és l'estat en què es troba un fil quan ha estat creat amb el mètode `new()`. El mètode `start()` posa el fil en estat de **preparat**. Encara no està en execució, si no que està preparat per utilitzar el processador si l'organitzador de fils ho troba oportú. Si l'organitzador li indica que passa a fer ús del processador, entra en l'estat d'**en execució**. De l'estat d'execució pot passar a un estat de **bloquejat**, **adormit** o **en espera** per diferents motius. En aquests estats un fil no s'executa ni es troba a la cua dels fils en espera d'execució. Quan algun esdeveniment el desbloquegi o arribi la hora de despertar-se, el fil transitarà a l'estat de preparat per tornar a ser executat. Direm que un fil està viu (*alive*), mentre transiti entre els estats *preparat*, *en execució*, *bloquejat*, *adormit* o *en espera*.

Quan un fil finalitzi la seva execució arribant al final dl mètode `run`, passarà a l'estat d'acabat i direm que està mort (*dead*), perquè des d'aquest estat no podrà tornar-se a executar més. Així doncs, la invocació del mètode `start` d'un fil acabat, provocarà una excepció en temps d'execució. La finalització d'un fil no implica pas que l'objecte `Thread` desaparegui i alliberi la memòria. Les instàncies de Thread segueixen les mateixes regles que qualsevol altre objecte, mentre es trobin assignades a alguna variable, es mantindran en memòria. Podem forçar

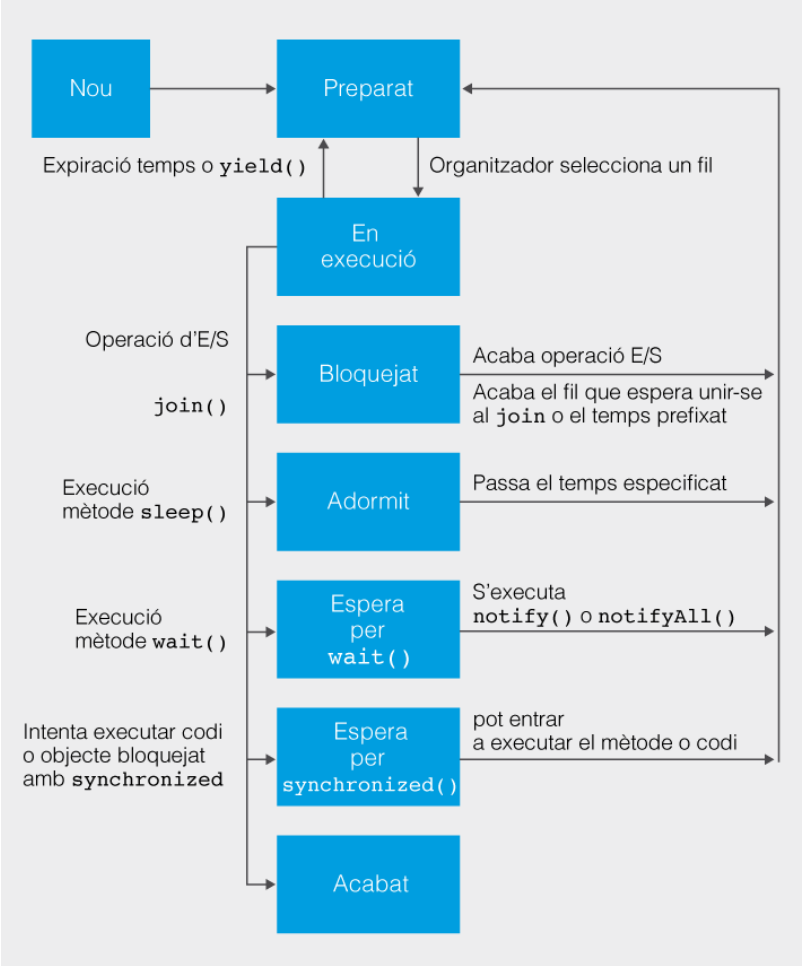
l'alliberament de memòria assignant un valor `null` a la variable que mantenia l'objecte `Thread`, un cop aquest hagi finalitzat.

Mentre el fil està en execució, podrà sortir-ne per diversos motius:

- Ha expirat el temps d'ús del processar per part del fil o s'executa el mètode `yield()`.
- Inicia una operació d'entrada/sortida i passa a un estat de **bloquejat** fins que la operació acabi.
- Crida al mètode `join()` d'un altre fil per esperar la finalització del mateix, El fil invocador passa a l'estat de **bloquejat** fins que el fil invocat passi a l'estat *acabat*.
- Intenta executar un codi que està sincronitzat (*synchronized*) i no pot perquè hi ha un altre fil executant-lo, passa a un estat de **bloquejat** fins que el bloc sincronitzat quedi alliberat.
- Es crida el mètode `wait()` passant-lo a l'estat *d'espera*. Quan un fil entre en estat d'espera, només en podrà sortir si s'invoca algun dels seus mètodes `notify()` o `notifyAll()`.
- S'invoca el mètode `sleep`. El fil passa a estat adormit fins que hagi transcorregut el temps indicat com a paràmetre. L'estat adormit, és molt útil per mantenir un fil aturat durant un temps determinat minimitzant al màxim els recursos utilitzats. Suposem que un programa de tractament de textos, disposa d'un fil que guarda els documents automàticament cada 10 minuts.

La figura 2.5 resumeix tots els canvis i transaccions que són possibles en els estats d'un fil.

Figura 2.5. Canvi d'estat d'un fil



La classe `Thread` conté el mètode `isAlive()` que ens indica si un fil ha estat inicialitzat (amb el mètode `start()`). Si `isAlive()` retorna *false* , sabem que el fil està a l'estat de **nou** o **finalitzat**, no sent possible diferenciar aquests dos estats. En canvi si retorna *true* ens diu que el fil ha estat inicialitzat amb `start()` i es trava en qualsevol dels altres estats.

2.2.3 Mètode thread.join()

La classe `thread` conté el mètode no estàtic `join()` que permet que un fil d'execució esperi a un altre. Dit d'una altra manera, la classe `thread` permet, a través del mètode `join()`, que un fil s'uneixi al final de la seva execució a un altre fil.

Si tenim un fil A que no pot completar la seva feina fins que un altre fil B hagi acabat, el que s'ha de fer és que el fil A s'uneixi a al fil B. El fil A estarà en estat de **bloquejat** fins que el fil B hagi acabat la seva execució i estigui en estat d'**acabat** (*dead*). El que farà `fil.join()`, doncs, és esperar en estat de bloquejat, fins que el fil que l'ha invocat acabi. En cas que el fil que l'ha invocat ja hagi acabat ignora l'ordre.

També és utilitzat per unir el fils d'execució d'una aplicació per tal d'evitar que el fil principal acabi abans que els fils creats en el codi de l'aplicació.

En la figura 2.6 es veu com des d'un fil (el fil principal) es creen 3 fils que executen les seves funcions (tirar un dau) i s'esperen que el fil que l'ha invocat acabi (el fil principal).

La figura correspon al següent codi, que crea tres fils i cada un llença un dau. El programa principal ha de mostrar el resultat de la suma de les tres tirades:

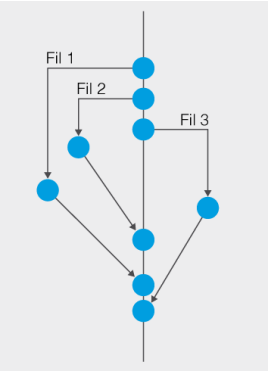
```
package joinfils;

class TiradaDaus {
    private int tiradaDau;

    public TiradaDaus (int e) {
        tiradaDau=e;
    }

    public synchronized int getSumaTirada() {
        return tiradaDau;
    }

    public synchronized void setSumaTirada(int e) {
        tiradaDau += e;
    }
   }


public class JoinFils implements Runnable {

    private TiradaDaus xobj;

    public JoinFils(TiradaDaus m) {
        xobj=m;
    }
    public void run(){
        try{
            Thread.sleep(1000);
            int resultatDau=(int) (Math.random()*6) + 1;
            xobj.setSumaTirada(resultatDau);
            System.out.println("Tirada fil "+Thread.currentThread().getName() +
                ": " +resultatDau);
        }catch (InterruptedException e){
        }
    }

    public static void main(String[] args) throws InterruptedException {
        TiradaDaus ans=new TiradaDaus(0);

        JoinFils obj1 = new JoinFils(ans);
        JoinFils obj2 = new JoinFils(ans);
        JoinFils obj3 = new JoinFils(ans);
        Thread fil_1 = new Thread(obj1);
        fil_1.setName("Dau 1");
        Thread fil_2 = new Thread(obj2);
        fil_2.setName("Dau 2");
        Thread fil_3 = new Thread(obj3);
        fil_3.setName("Dau 3");
        fil_1.start();
        fil_2.start();
        fil_3.start();
        fil_1.join(); //Espera el fil_1 que el fil principal, el que l'ha
            invocat acabi
        fil_2.join();
        fil_3.join();

```

```
        System.out.println("Total tirada: "+ ans.getSumaTirada());
        System.out.println("Final Fil Principal");

    }

}
```

Resultat:

```
run:
Tirada fil Dau 1: 2
Tirada fil Dau 3: 1
Tirada fil Dau 2: 5
Total tirada: 8
Final Fil Principal
BUILD SUCCESSFUL (total time: 1 second)
```

Figura 2.6. Esquema de com actua el mètode "join" aturant l'execució del fil principal a l'espera que acabin els tres fils que ha creat



Evidentment per treure la suma de les tres tirades, en la línia:

```
System.out.println(ans.getSumaTirada());
```

hem d'esperar que els tres fils acabin l'execució. I per fer-ho hem de fer que el fil principal (en aquest cas ja que els fils s'han executat des del *main*) esperi per acabar la seva execució que els tres fils acabin. Això ho fem amb el mètode `join`.

```
fil_1.join(); //Espera el fil_1 que el fil principal, el que l'ha invocat,
    acabi
fil_2.join();
fil_3.join();
```

Si no esperem la finalització dels fils, el fil principal dóna de resultat la suma de les tres tirades i pot ser un resultat erroni, ja que és possible que algun dels tres fils no hagi acabat. Sense el `join` aquest seria un possible resultat:

```
run:
Tirada fil Dau 1: 4
Tirada fil Dau 2: 1
Total tirada: 5
Final Fil Principal
Tirada fil Dau 3: 6
BUILD SUCCESSFUL (total time: 1 seconds)
```

Tal com podem veure, el programa principal mostra el resultat únicament de dos tirades, els fils que s'han executat abans de la finalització del fil principal. L'última tirada no l'ha tinguda en compte.

2.3 Sincronització de fils

En entorns multifils que s'executen concurrenment és habitual que entre els diferents fils d'execució hi hagi la necessitat de compartir dades. Aquesta compartició de dades ens pot servir per comunicar fils. Utilitzarem variables compartides per indicar que un fil ha acabat una certa funció o no i, per tant, un altre fil podrà executar la part de codi que estava bloquejat per aquesta variable. També podem compartir dades d'un objecte comú. Diversos fils han de modificar una propietat d'un objecte. En ambdós casos, convé sincronitzar l'accés als recursos compartits per evitar incosistència de dades o interferències. Qualsevol variable accessible pot ser manipulada per un fil o el fil principal. Si aquest accés no es controla pot provocar situacions no esperades o errònies.

Treballant amb fils aquesta compartició de variables és molt fàcil i gens costosa, ja que els fils d'un mateix procés comparteixen el mateix espai de memòria.

Java proporciona mecanismes per sincronitzar els fils que tracten aquests problemes.

Quan més d'un fil ha d'accedir al mateix recurs és quan apareix la possibilitat d'operacions que s'executen concurrentment utilitzant el mateix objecte o recurs i, per tant, es poden donar casos de dades corruptes o resultats no esperats. Aquestes zones de codi que accedeixen a un recurs compartit i que no poden ser accessibles al mateix temps per més d'un fil, s'anomenen seccions crítiques. Un efecte equivalent és quan diversos fils volen executar el codi d'una secció crítica. Els fils cooperen utilitzant el protocol que, per realitzar algunes operacions sobre un objecte, han d'agafar el bloqueig de l'objecte. Agafar el bloqueig d'un objecte significa que no deixa que altres fils agafin aquest bloqueig fins que el fil que el té l'alliberi. Això s'anomena exclusió mútua. Si aquesta acció es realitza correctament, el fils d'execució no realitzaran operacions que interfereixin entre si.

Allò que s'intenta és que les operacions sobre les seccions crítiques siguin atòmiques. Que s'executin totalment o no s'executin.

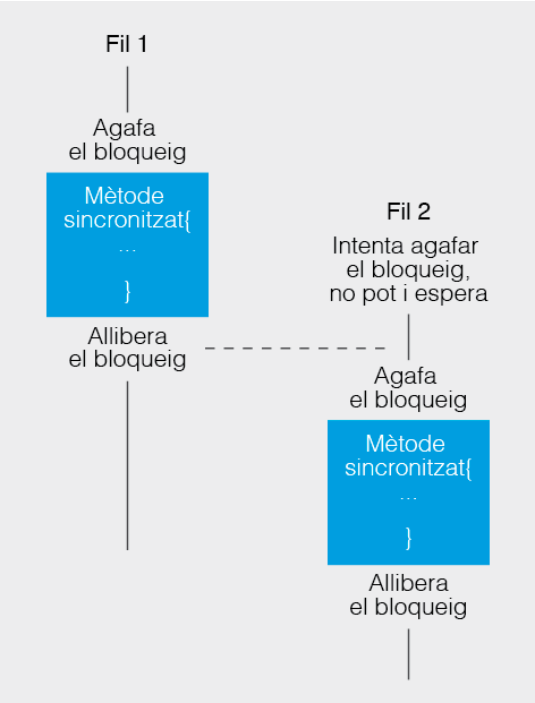
Per defecte, a Java un objecte no està protegit. Això vol dir que qualsevol nombre de fils pot executar codi dins de l'objecte. L'exclusió mútua s'aconsegueix a Java amb la paraula reservada `synchronized`. Aquesta paraula pot ser aplicada a blocs de codi dins d'un mètode i també a mètodes sencers.

2.3.1 Mètodes synchronized

Si volem protegir els objectes d'una classe de possibles interferències a un entorn multifil, és a dir, que els mètodes d'aquesta classe s'executin en exclusió mútua, haurem de posar el modificador ***synchronized*** als mètodes que formin part de la secció crítica. El modificador *synchronized* ens garanteix que cap altre mètode sincronitzat de l'objecte podrà executar-se. Quan un fil executa un mètode sincronitzat agafa el bloqueig de l'objecte fins que acaba la seva execució i s'allibera del bloqueig. Si altre fil crida el mètode sincronitzat del mateix objecte, queda bloquejat fins que quedi el fil que l'està executant l'alliberi del bloqueig.

En la figura 2.7 es mostra com dos fils intenten executar un mètode sincronitzat. Un agafa el bloqueig i l'altre es queda a l'espera que l'alliberi per poder executar el mètode.

Figura 2.7. Bloqueig-Alliberar mètode sincronitzat



Els mètodes sincronitzats proporcionen exclusió mútua per accedir al recurs compartit. Els mètode no sincronitzats del mateix objecte poden ser executats per diversos fils a la vegada. La possessió del bloqueig d'un mètode és única per fil. És a dir, un fil no pot bloquejar dos cops el mateix mètode, però sí que pot fer crides recursives a un mètode bloquejat per ell. La forma natural d'alliberar un bloqueig és que el fil arribi al final del codi del mètode o trobi una sentència `return`. També pot alliberar-lo per altres motius, com ara que es llanci una excepció.

Aquest sistema que ens proporciona Java d'exclusió mútua fa que el programador no hagi de recordar de posar *bloqueig* i *alliberar bloqueig* a cada mètode. Java ja ho fa per nosaltres.

En el següent codi es pot veure com sincronitzem alguns mètodes d'una classe i d'altres no. Aquells que volem que s'executin en exclusió mútua els sincronitzem amb la paraula reservada `synchronized`.

```
public class sincronitzaMetodes {

 public void metodeNoSincronitzat_1(){
 //.........
 }

  public synchronized void metodeSincronitzat_1(){
 //.........
 }
  public synchronized void metodeSincronitzat_2(){
 //.........
 }
    public void metodeNoSincronitzat_2(){
 //.........
 }
}
```

En una situació en la qual dos fils estan accedint a la mateixa instància d'una classe i volen executar el mateix mètode accedint i modificant simultàniament el mateix objecte i mateixes variables, si aquest accés no està controlat i les dades que modifiquen els fils serveixen per indicar com continua el programa funcionant, el resultat serà amb tota seguretat erroni i inesperat.

Imaginem una agència de viatges que venen passatges d'avió. Hi ha dues persones interessades en viatjar amb la seva família a Roma al vol SP233 a les 20.00h. El primer que hauria de fer el sistema és mirar si hi ha places suficients al vol i si n'hi ha, descomptar les places de l'avió i assignar-les al client.

Però què passaria si les dues persones volen fer la reserva al mateix temps?

Al `main` de la classe `AgenciaViatges` es creen dos fils sobre la mateixa instància de l' `object`. Quan s'executen al mètode `run()`, el dos clients intenten fer la reserva de 3 seients per la seva família de forma simultània. Inicialment només hi ha 5 seients lliures.

En el mètode `run` es crida al mètode `gestioSeientsLliures(3)`. Aquest mètode el que farà és: si hi ha tres seients lliures els assigna al client (és a dir, els resta dels seients lliures).

Si hi ha seients lliures s'executa un `sleep()` d'1 segon, que farà adormir el fil per simular un sistema més real amb més volum de dades.

> El mètode `sleep(milisegons)` deixa el fil en estat de **bloquejat** durant un temps marcat en milisegons i un cop passat aquest temps torna a l'estat de **preparat**. Durant aquest temps si el fil tenia el bloqueig d'un objecte, el continuarà tenint. No perd el bloqueig durant aquest temps.

Després de despertar el fil, crida el mètode `reservaSeients()` de la classe `SeientsAvio` que restarà els seients reservats dels seients lliures.

```java
package agenciaviatges;


//Implementem runnable per poder crear fils sobre les seves instàncies
public class AgenciaViatges implements Runnable {

    private SeientsAvio sa = new SeientsAvio();

    public void run(){
                gestioSeientsAvio(3);
    if(sa.getSeientsLliures()<=0)
                    System.out.println("No hi ha seients lliures");
         }

    public void gestioSeientsAvio(int numSeientsReserva){

//Comprovem si hi ha seients suficients
        if(sa.getSeientsLliures()>=numSeientsReserva)
        {
            System.out.println(Thread.currentThread().getName()+" farà una reserva de "+ numSeientsReserva +" places");
            try {
                    Thread.sleep(1000); //adormim el fil 1 segon
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
             }
  //Fem la reserva dels seients
            sa.reservaSeients(numSeientsReserva);

            System.out.println(Thread.currentThread().getName() + " Reserva realitzada amb èxit."+" Les places lliures són "+sa.getSeientsLliures());
         }else{
            System.out.println("No hi ha places suficients pel client." + Thread.currentThread().getName()+" Les places lliures són "+sa.getSeientsLliures());
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }


    }


    public static void main(String[] args) {

        AgenciaViatges objAg = new AgenciaViatges();

        //creem els dos fils sobre la mateixa instància
        Thread Fil_1 = new Thread(objAg);
        Thread Fil_2 = new Thread(objAg);
        Fil_1.setName("Client 1");
        Fil_2.setName("Client 2");

        Fil_1.start();
        Fil_2.start();

    }
}


class SeientsAvio {
   //comencem amb 5 seients lliures a l'avió
    private int seientsLliures = 5;
```

```
65
66      public int getSeientsLliures(){
67          return seientsLliures;
68      }
69
70      public boolean getSeientsLliures(int numPlaces){
71          if (numPlaces<=seientsLliures)
72              return true;
73          else
74              return false;
75      }
76
77      public void reservaSeients(int numSeientsReserva){
78          seientsLliures = seientsLliures - numSeientsReserva;
79      }
80
81  }
82  /
```

Un possible resultat és:

```
1  run:
2  Client 1 farà una reserva de 3 places
3  Client 2 farà una reserva de 3 places
4  Client 1 Reserva realitzada amb èxit. Les places lliures són 2.
5  Client 2 Reserva realitzada amb èxit. Les places lliures són -1
6  BUILD SUCCESSFUL (total time: 2 seconds)
```

S'ha produït un error inesperat, hem reservat places que no existeixen. Hem creat un *overbooking* a l'avió.

Aquest error s'ha produït perquè el Client 1 ha consultat les places lliures de l'avió i ha vist que n'hi havia 5. S'ha adormit durant 1 segon i mentre estava adormit i abans de fer la reserva (és a dir, restar els 3 seients), el Client 2 ha consultat el número de seients i ha vist que també en quedaven 5 i que, per tant també podia fer la seva reserva, ignorant que el Client 1 està pendent de reservar les seves. Finalment tots dos completen el procés i creen un número negatiu de seients lliures.

La solució a aquest problema és utilitzar semàfors sobre les seccions crítiques. Les seccions crítiques en aquest exemple són la consulta de seients lliures i la resta de seients reservats. És a dir, les operacions de `gestioAvio()`.

Un semàfor és com una cadenat amb una única clau. Aquesta clau únicament la tindrà el client que l'agafa primer i cap altre client pot agafar-la fins que ell la deixi.

El que farem serà protegir els mètodes de la secció crítica amb aquest cadenat, de manera que si el Client 1 consulta els seients lliures agafa la clau i tanca el semàfor, quan el Client 2 vol consultar el seients lliures, necessita la clau i com que no la pot agafar es queda en espera fins que el Client 1 acaba la seva transacció (restar els seients reservat dels lliures) i allibera la clau.

Java pot crear semàfors amb la paraula clau *synchronized*. Posarem el cadenat al mètode encarregat de fer tota la gestió de consulta i reserva de seients, és a dir, al mètode `gestioAvio()`. D'aquesta manera podrem estar segurs que quan un client agafa la clau del cadenat cap altre client pot fer aquesta gestió.

Per fer-ho, hem d'afegir dues línies més a l'inici i al final del mètode que mostren qui i quan agafa la clau i després l'allibera.

```
public synchronized void gestioSeientsAvio(int numSeientsReserva){

    System.out.println(Thread.currentThread().getName() + " té la clau del
        cadenat");
//Comprovem si hi ha seients suficients
        if(sa.getSeientsLliures()>=numSeientsReserva)
        {
            System.out.println(Thread.currentThread().getName()+" farà una
                reserva de "+ numSeientsReserva +" places");
            try {
                    Thread.sleep(1000); //adormim el fil 1 segon
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
             }
  //Fem la reserva dels seients
            sa.reservaSeients(numSeientsReserva);

            System.out.println(Thread.currentThread().getName() + " Reserva
                realitzada amb èxit"+" Les places lliures són "+sa.
                getSeientsLliures());
         }else{
            System.out.println("No hi ha places suficients pel client: " +
                Thread.currentThread().getName()+" .Les places lliures són "+
                sa.getSeientsLliures());
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
     System.out.println(Thread.currentThread().getName() + " deixa la clau del
         cadenat.");

    }
```

El resultat serà el següent

```
run:
Client 1 té la clau del cadenat.
Client 1 fa una reserva de 3 places.
Client 1 reserva realitzada amb èxit. Les places lliures són 2.
Client 1 deixa la clau del cadenat.
Client 2 té la clau del cadenat.
No hi ha places suficients pel Client 2. Les places lliures són 2.
Client 2 deixa la clau del cadenat.
BUILD SUCCESSFUL (total time: 2 seconds)
```

Tal com es pot veure, el Client 1 agafa la clau del cadenat i fa tota la gestió: consulta el número de seient lliures i fa la reserva dels seients. Quan acaba tot el procés deixa la clau.

Posteriorment agafa la clau el Client 2, que quan consulta el número de seients lliures s'adona que no n'hi ha suficients per fer la seva reserva. Acaba dient que no hi ha places suficients i allibera la clau.

2.3.2 Objectes synchronized

Quan sincronitzem mètodes estem bloquejant aquests mètodes per tal que únicament un fil pugui executar-los a la vegada. Però si sincronitzem molt, tots els avantatges que ens proporciona la concurrència es perden. Per tant, hem de sincronitzar únicament aquell codi que ens interessa. Amb Java podem sincronitzar només una part de codi dins d'un mètode, aquella part que volem que s'executi en exclusió mútua. Per sincronitzar únicament una part del codi necessitem fer referència a un objecte. El següent codi mostra com es sincronitza una part del codi sobre l'objecte que ha cridat al mètode `metodeNoSincronitzat()`.

La part del codi sincronitzada farà que s'executi en exclusió mútua amb qualsevol altre bloc sincronitzat del mateix objecte. També s'executaran en exclusió mútua els mètodes amb el modificador *synchronized*.

```
class Sincronitzat {

    public void metodeNoSincronitzat(){
        //Part de codi no sincronitzat

        synchronized (this){
            //Codi sincronitzat
        }

        //Part de codi no sincronitzat

    }
}
```

En l'exemple anterior podríem modificar el codi per sincronitzar un bloc de codi de la següent manera:

```
public class AgenciaViatges implements Runnable {

    private SeientsAvio sa = new SeientsAvio();
    public void run(){
        //Codi no sincronitzat
       System.out.println("Seients Lliures: " + sa.getSeientsLliures());
       synchronized(sa){
                gestioSeientsAvio(3);
                if(sa.getSeientsLliures()<=0)
                   System.out.println("No hi ha seients lliures");
         }
//Codi no sincronitzat

}

    public  void gestioSeientsAvio(int numSeientsReserva){...
//.
}
```

En l'exemple no es guanya gaire ja que no hi ha codi per sobre del bloc sincronitzat. També hem de veure que no cal sincronitzar el mètode `gestioSeientsAvio(int)` , ja que al cridar-lo des d'un un bloc sincronitzat, l'objecte el manté bloquejat. Per sincronitzar el bloc podríem haver posar `synchronized(this)`, ja que l'objecte que bloquegem és el mateix que invoca el mètode.

També podem sincronitzar part del codi sobre un objecte diferent a aquell que ha cridat el mètode que s'està executant. Abans d'executar el codi que està sincronitzat, hem d'agafar el bloqueig de l'objecte `altreObjecte`. Cap altre fil haurà de tenir el bloqueig d'aquest objecte.

```
class SincronitzatAltreObjecte {

    public void metodeNoSincronitzat(){
        //Part de codi no sincronitzat

        synchronized (altreObjecte){
            //Codi sincronitzat per l'objecte altreObjecte
        }

        //Part de codi no sincronitzat

    }
}
```

És un mètode que podem fer servir si volem realitzar més d'una operació atòmica sobre un objecte, com per exemple consultar les places lliures d'un avió i després realitzar una reserva.

```
synchronized (seientsAvio){
      if (seientsAvio.getSeientsLliures(numPlaces))
              seientsAvio.reservaSeients.(numPlaces);
      }
```

2.3.3 Sincronitzar mètodes estàtics

Els mètodes estàtics també es poden sincronitzar. Ja que únicament es protegeix una còpia de les dades estàtiques que es manipulen, únicament es necessita un bloqueig per classe.

Totes les instàncies tenen un objecte associat a java.lang.Class. Aquesta instància és la que s'utilitza per protegir (sincronitzar) els mètodes estàtics.

Si volem que un mètode protegeixi una classe sencera, tots el objectes creats d'una classe, haurem de sincronitzar un mètode estàtic.

La sincronització de mètodes estàtics és similar als mètodes no estàtics.

```
public static synchronized void metodeSincronitzatEstatic() {...
 //.
}
```

També podem sincronitzar blocs de codi de mètodes estàtics. Per fer-ho hem d'indicar quina classe volem sincronitzar. En l'exemple, `LaMevaClasse.class` és una classe literal. El que fa la màquina virtual de Java és anar a buscar la instància de `Class` que està representant la classe que crida `LaMevaClasse`.

```
public static void metodeNoSincronitzat() {
//Codi no sincronitzat
synchronized(LaMevaClasse.class) {
```

```
//Codi sincronitzat
}
//Codi no sincronitzat
}
```

El següent codi és equivalent a l'anterior:

```
public static void metodeNoSincronitzat() {
//Codi no sincronitzat
Class lmc = Class.forName("LaMevaClasse");
synchronized(lmc) {
//Codi sincronitzat
}
//Codi no sincronitzat
}
```

2.3.4 Variables volatile

Les variables primitives a Java són atòmiques en lectura i actualització però no en operacions. Per tant, podem ometre'n la sincronització si l'únic que volem fer és modificar un valor d'una variable primitiva. D'aquesta manera augmentem el rendiment de l'aplicació. Hem de tenir en compte, però, un aspecte important. La màquina virtual de Java optimitza codi quan els compila si es troba un codi com el següent:

```
class mètodeExemple {
private bolean valor = false;
public void cambiarValor () {
valor = true;
}
public class classeExemple {

    private boolean flag = false;

    public void cambiarFlag(){
        flag=true;
    }

    public synchronized void metodeExemple () {
        flag = false;
        // un altre fil prodria cridar al mètode cambiarFlag()
        if (flag) {
            // ...
        }
     }
 }
```

La màquina virtual de Java pot optimitzar el codi pensant que el que es troba dins d' `if` (*flag*) no s'executarà mai, ja que *flag* no pot ser mai *true* perquè la línia anterior el posa a *false*. Si creu que és codi mort no s'executarà mai. Però un altre fil pot modificar el valor de *flag* accedint al mètode `cambiarFlag()` quan un primer fil està a punt d'entrar a `if` i, per tant, sí que es podria entrar al bloc d' `if`.

La forma d'evitar-ho és declarar la variable *flag* com a ***volatile***. D'aquesta manera li podem dir a la màquina virtual de Java que un altre fil pot modificar aquest

valor. La paraula reservada ***volatile*** aplicada sobre variables indica que se'n farà l'actualització de forma atòmica.

2.3.5 Interbloqueig

Els avantatges de sincronitzar mètodes a Java són evidents. No podríem realitzar concurrència sense aquest tipus de seguretat i bloquejos. Però hem d'anar amb cura quan apliquem aquests mètodes. En primer lloc, només hem de sincronitzar aquells mètodes o blocs de codi necessaris, ja que abusar de la sincronització penalitza el rendiment de l'aplicació. En segon lloc perquè si no sincronitzem correctament es podem produir errors. Evidentment podem realitzar bloquejos sense fer servir les primitives que ens proporciona Java, però la font d'errors és major, ja que hem de recordar d'agafar el bloqueig i alliberar-lo, cosa que amb *synchronized*, tal com hem vist, no és necessària.

L'error més greu que es pot produir quan es treballa en programació multifil relativament complexa és l'anomenat **interbloqueig** (en anglès *deadlock*). L'interbloqueig es produeix quan dos fils s'estan executant concurrentment i cada un té un bloqueig exclusiu que l'altre necessita per continuar. El fenomen pot passar amb més de dos fils i per norma general passa quan els fils agafen els mateixos bloquejos en diferent ordre. La realitat és que és difícil que passi un interbloqueig en aplicacions simples amb un bloqueig o dos, però quan la programació és més complexa, amb molts bloquejos i alliberaments a tot el codi, el problema és molt real.

Imaginem una aplicació que gestiona els recursos d'una entitat bancària (comptes bancaris, documents, etc.) que permetin que les dades puguin ser transferides d'un recurs a un altre. Cada recurs té un bloqueig associat per a ell. Si es vol fer una transferència, el fil ha d'agafar el bloqueig de tots els recursos que intervenen en la transferència. Si dos fils inicien la transferència que afecta a dos recursos i els fils agafen el bloqueig dels recursos en diferent ordre ja tenim un interbloqueig.

```
1  public class FerTransferencia {
2      private static class CompteCorrent {
3          //Codi de la classe
4      }
5      private CompteCorrent transferenciaA = new CompteCorrent();
6      private CompteCorrent transferenciaB = new CompteCorrent();
7
8      public int metodeTransferenciaArxius() {
9          synchronized(transferenciaA) { // Possible interbloqueig
10             synchronized(transferenciaB) {
11                 //Codi del mètode
12             }
13         }
14     }
15     public void metodeTransferenciaCalers() {
16         synchronized(transferenciaB) { // Possible interbloqueig
17             synchronized(transferenciaA) {
18                 //Codi del mètode
19             }
20         }
21     }
22 }
```

La classe és estàtica, cosa molt usual quan volem protegir amb sincronització els objectes d'una aplicació. Si mirem el codi anterior i imaginem un fil que comença a executar el mètode `metodeTransferenciaArxius` i agafa el bloqueig sobre l'objecte `transferenciaA`. Un segon fil entra i executa el mètode `metodeTransferenciaCalers` i agafa el bloqueig sobre el mètode `transferenciaB`.

En aquest moment s'ha produït un interbloqueig. El primer fil espera que s'alliberi la `transferenciaB` per poder agafar el bloqueig i continuar la seva execució. En canvi, el segon fil està esperant que s'alliberi `transferenciaA` per poder agafar el bloqueig i continuar la seva execució. Evidentment cap de les dues coses passarà. Hi haurà una espera infinita, un interbloqueig. És possible que aquest mateix codi s'executi sempre bé, sense que es produeixi un interbloqueig mai, però la probabilitat hi és.

En aquest cas, la solució és realment senzilla. Només cal assegurar que la sincronització d'ambdós objectes (transferenciaA i transferenciaB) es produeixi sempre en el mateix ordre. En el codi que ens ocupa, simplement caldrà canviar les ordres de sincronització en un dels mètodes, per exemple en `metodeTransferenciaCalers` perquè per tal que l'ordre de sincronització coincideixi amb el del mètode `metodeTransferenciaArxius`.

No sempre és, però, tan trivial de detectar un possible codi amb interbloqueig. En el següent exemple, hi ha un mètode que s'encarrega de fer una transferència entre dos comptes, de `compteOrigen` a `compteDesti`. El codi bloqueja els dos comptes perquè la transferència sigui segura i atòmica.

```
public void transferirDiners(CompteCorrent compteOrigen, CompteCorrent
    compteDesti,double quantitat) {
    synchronized (compteOrigen) {
      synchronized (compteDesti) {
        if (compteOrigen.hihaSaldoSuficient(quantitat) {
          compteOrigen.reintregre(quantitat);
          comptedesti.ingres(quantitat);
        }
      }
    }
  }
```

Ara bé, que un fil invoca la següent sentència:

```
transferirDiners(compte2100, compte8888, quantitatIngressar1);
```

El fil intentarà bloquejar el primer el compte `compte2100` i posteriorment el compte `compte8888` abans d'executar el codi sincronitzat.

Però que passarà, si mentre es realitza el bloqueig de `compte2100`, un altre fil invoca la següent instrucció?

```
transferirDiners(compte8888, compte2100, quantitatIngressar2);
```

El que succeirà és que el segon fil agafarà el bloqueig del compte `compte8888` i en intentar realitzar el bloqueig de l'altre compte, es queda a l'espera perquè el primer fil el mantindrà bloquejat. El primer fil restarà també boquejat esperant

que el compte `compte8888` quedi lliure, però això no es produirà mai, doncs ens trobem davant d'una situació d'interbloqueig.

Per solucionar-ho, haurem d'aconseguir de nou, que els bloquejos es facin sempre en el mateix ordre, independentment de quin sigui el compte d'origen i quin el compte de destí.

En el següent codi veurem una possible solució que passa per crear dos objectes de tipus `compteCorrent` i assignar-li el compte d'origen o destí en funció del seu ordre numèric. Hem creat també un mètode que ens retorna el número de compte. Ara, sempre agafarà el bloqueig en el mateix ordre independentment de la crida que es faci a `transferirDiners()`.

```
1  public void transferirDiners(CompteCorrent compteOrigen, CompteCorrent
       compteDesti,double quantitat) {
2     CompteCorrent primerBloqueig, segonBloqueig;
3      if (compteOrigen.numeroCompte() == compteDesti.numeroCompte())
4        throw new Exception("No es pot fer una transferència sobre el mateix
             compte");
5      else if (compteOrigen.numeroCompte() < compteDesti.numeroCompte()) {
6        primerBloqueig = compteOrigen;
7        segonBloqueig = compteDesti;
8      }
9      else {
10       primerBloqueig = compteDesti;
11       segonBloqueig = compteOrigen;
12     }
13     synchronized (primerBloqueig) {
14       synchronized (segonBloqueig) {
15
16     synchronized (compteOrigen) {
17       synchronized (compteDesti) {
18         if (compteOrigen.hihaSaldoSuficient(quantitat) {
19           compteOrigen.reintregre(quantitat);
20           comptedesti.ingres(quantitat);
21         }
22       }
23     }
24   }
```

La part més important per evitar interbloquejos és la documentació. Si creem una bona documentació explicant on s'agafen els bloquejos, quins objectes hi intervenen i com han de ser les crides als mètodes, ens estalviarem molts problemes quan algun altre programador hagi de fer un manteniment de l'aplicatiu. Els programadors sabran que cridar un mètode en concret pot implicar un interbloqueig.

2.4 Compartir informació entre fils

La paraula clau *synchronized* ens dóna seguretat i integritat en mètodes o parts de codi. Però a més de sincronització i protecció, els fils també han de poder comunicar-se entre ells. Java ens proporciona mètodes que fan que els fils esperin i continuïn la seva execució. Són els mètodes: `wait()`, `notify()` i `notifyAll()`.

2.4.1 Mètodes wait(), notify() i notifyAll()

Sincronitzar mètodes ens proporciona una exclusió mútua per accedir a una secció crítica. Però si existeixen fils interdependents o un fil ha d'esperar una condició per poder continuar la seva execució i aquesta condició l'ha de donar un altre fil, el sincronisme no ho soluciona.

Imaginem dos fils que controlen l'enviament i el processament de correus electrònics. El primer fil està buscant correus electrònics per posar a la cua d'enviats i el que processa els correus els construeix per deixar-los a la cua. Per què el primer fil ha d'estar malgastant temps i recursos cada dos segons buscant a la cua de correus per enviar si encara no n'hi ha?. És millor si el posem en suspensió i quan arribi un correu a la cua ja l'avisarem i verificarem el correu per enviar-lo.

Els mètodes que proporciona Java per esperes i notificacions són:

- `wait()`: treu el fil en curs i manté el bloqueig de la zona d'exclusió mútua. L'envia a una **cua d'espera** o **conjunt d'espera** (*wait set* en anglès) i el passa a l'estat d'**en espera**.

Haurà d'esperar a la cua fins que un altre fil el tregui d'allà i el canviï a l'estat de **preparat** mitjançant els mètodes `notify()` o `notifyAll()`.

- `notify()`: un fil de la cua d'espera és seleccionat per passar a l'estat de **preparat**.
- `notifyAll()`: tots els fils del la cua d'espera passen a l'estat de **preparat**.

Tant `notify()` com `notifyAll()` posen a *preparat* els fils de la cua d'espera, si n'hi ha. Si no hi ha cap fil esperant, la instrucció és ignorada.

Aquests mètodes són de la classe `object`, per tant són heretats per totes les classes de forma implícita.

> Els tres mètodes, `wait()`, `notify()` i `notifyAll()`, han d'executar-se sobre el fil que té el bloqueig. Per tant, totes les crides a aquests mètodes s'han de fer des de mètodes o blocs de codi sincronitzats.

La utilització de sincronisme i comunicació de fils a Java és la forma de crear monitors. Recordem que un monitor és una estructura d'alt nivell que gestiona la concurrència de forma que només permet a un únic fil agafar el monitor (un únic fil hi té accés a la vegada) i conté una llista de fils en espera per entrar a gestionar el seu accés.

Existeix una forma més o menys estàndard d'implementar els fils que han d'esperar que succeeixi un esdeveniment per poder continuar l'execució.

El codi següent mostra de quina manera, si una condició no és certa, el fil passarà a l'estat *en espera* i deixarà lliure el bloqueig sobre l'objecte. Quan un altre fil li notifiqui que pot continuar la seva execució i s'executi, tornarà a avaluar la condició i, si és certa, executarà el codi. L'encarregat de notificar que pot continuar i que ha de sortir de la cua d'espera és un mètode que fa que la condició es faci certa.

```
synchronized void metodeWait () {
while (!condicio)
try {
wait();
}
catch (InterruptedException e) {
// codi d'excepció
}
// codi que s'executa quan la condició és certa.
}

synchronized void metodeCanviCondicio(boolean valor) {
condicio = valor;
notify(); // o notifyAll()
}
```

Tot allò que hem de tenir en compte quan programem mètodes d'espera i notificació és:

- El mètode `metodeWait()` ha d'executar-se sincronitzat per tal d'assegurar que en canviar la condició del `while` i produir-se la notificació, només un fil aconseguirà executar el codi situat a la sortida del bucle.
- El mètode `metodeCanviCondicio()` modifica el valor de la condició usada en el `while` del mètode anterior i notifica el canvi per tal que si hi ha algun fil esperant dins del bucle, s'alliberi. El fil desbloquejat tornarà a bloquejar-se en cas que la condició avaluada prengués el valor cert, però aconseguiria sortir del bucle i continuar l'execució en cas que la condició fos falsa. Aquestes dues operacions s'han d'executar sempre de forma atòmica, doncs en cas contrari perdria la seva efectivitat, perquè podria passar que es cridés el mètode `notify()` entre l'alliberació del bloqueig i el canvi d'estat a **en espera**. En aquest cas el mètode `notify()` no afectaria al fil.
- Un fil que és alliberat a través d'un `notify()` passarà a l'estat de **preparat**. És a dir, no vol dir que s'executi immediatament sinó que tornarà a lluitar per agafar el bloqueig amb altres fils en el mateix estat.
- Quan un fil passa pel `wait()` es queda en espera immediatament, i no fa res per veure si allò que estava esperant ja ha passat. Això només passa si el fil que executa el mètode `metodeCanviCondicio()`, invoca `notify()`.

En el següent codi d'exemple veurem com aplicar l'espera i la notificació en un programa petit.

La idea és crear dos fils: un ha d'escriure *hola* i l'altre *adéu*. Però s'ha de fer en ordre, primer imprimim *hola* i després *adéu*. Si no implementem una espera i una notificació i llancem els dos fils, l'ordre de impressió podria ser erroni.

```java
package holaadeum;

public class HolaAdeuM {
    public static void main(String args[]) {
        escriuHolaAdeu eha = new escriuHolaAdeu();
  new E_Adeu(eha);
        new E_Hola(eha);
    }
}


class E_Hola implements Runnable {
      escriuHolaAdeu eh;
      E_Hola (escriuHolaAdeu eh) {
    this.eh =  eh;
    new Thread(this, "Hola").start();
      }
      public void run() {
            try{
                for(int x=0;x<5;x++) {
                    Thread.sleep(1000);
                    eh.eHola();
                }
            }catch (InterruptedException e){
            }
      }
}

class E_Adeu implements Runnable {
      escriuHolaAdeu eh;
      E_Adeu (escriuHolaAdeu eh) {
    this.eh =  eh;
    new Thread(this, "Adéu").start();
      }
      public void run() {
    for(int x=0;x<5;x++) {
          eh.eAdeu();
    }
      }
}

class escriuHolaAdeu {

    boolean escritHola = false;

    public synchronized void eAdeu() {
        while (escritHola == false) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        escritHola = false;
        System.out.println(" Adéu ");
    }

    public synchronized void eHola() {
        System.out.println(" Hola ");
        escritHola = true;
        notify();
    }

}
```

La classe `escriuHolaAdeu()` és la que conté els mètodes d'escriptura i la lògica de l'espera i notificació.

El mètode `eAdeu()` és l'encarregat d'escriure *Adéu*. Com que aquest mètode ha d'esperar que *hola* estigui escrit, va comprovant la variable booleana `escritHola` i, mentre sigui falsa, envia el fil a la cua d'espera. Únicament quan el mètode `eHola()` és executat per un fil que imprimeix el missatge, col·loca la variable a `true` i posteriorment llança un `notify()` que allibera el fill que es troba en espera.

Quan el fil és seleccionat per executar-se troba la variable a `true`, salta el `while` i escriu el missatge. Després torna a posar la variable booleana a `false` per les execucions següents.

El resultat seria semblant a:

```
run:
 Hola  Adéu  Hola  Adéu  Hola  Adéu  Hola  Adéu  Hola  Adéu
BUILD SUCCESSFUL (total time: 5 seconds)
```

Una altra manera de solucionar el mateix problema és utilitzant un **monitor**. Un monitor no és més que una classe, una col·lecció de mètodes i dades, que s'utilitza com a eina de sincronització amb la característica que si un fil agafa el control del monitor, cap altre fil pot agafar-lo fins que el primer no el deixi.

Els fils poden cridar els mètodes visibles del monitor, però no poden accedir directament a les estructures de dades internes del monitor.

Bàsicament, un monitor ha de tenir: un mètode que enviï els fils a la cua d'espera (`wait()`) depenent d'una condició; un altre mètode que rescati els fils de la cua d'espera (`notify()`) i canviï el valor de la condició; i un constructor que inicialitzi la variable de la condició.

La classe `escriuHolaAdeu()` de l'exemple anterior podria ser el monitor de l'aplicació.

2.4.2 Productor-consumidor

Un exemple clàssic de programació concurrent i multifils és el model de productor-consumidor. Es tracta d'un fil (productor)que genera una sortida que és agafada per un altre fil (consumidor).

Imaginem un sistema en el qual un fil (productor) va generant números aleatoris i un altre fil (consumidor) els va agafant per mostrar-los.

Ens hi trobem diferents classes. La classe `monitor` serà el nostre recipient de números. Hi hauran dos mètodes: `agafar()` que ens retornarà el número generat pel productor i `deixar(int)` que deixarà el número a disposició del consumidor. Per tant, la classe monitor realitzarà el control de transacció i comunicació entre els dos fils: productor i consumidor.

```
class Monitor {

    private int numero = 0;
    void Monitor(){
```

```
        numero=0;
    }

    public synchronized int afagar() {
        return( numero );
    }

    public synchronized void deixar( int num ) {
        numero = num;
    }
 }
```

La classe `productor` s'encarregarà d'anar produint els números i de deixar-los a disposició del consumidor del monitor.

```
class Productor implements Runnable {
    private Monitor safata;

    public Productor( Monitor s ) {
        safata = s;
     }
    private int numero =0;

    public void run() {

        // Posa 10 lletres a la canonada
        for( int i=0; i < 5; i++ ) {
                numero= (int)(10*Math.random());
                safata.deixar(numero);
                System.out.println( "Produït el número "+numero );
                try {
                    Thread.sleep( (int)(Math.random() * 1000 ) );
                } catch( InterruptedException e ) {;}
            }
        }
    }
```

La classe `consumidor` s'encarregarà d'anar consumint els números del monitor.

```
class Consumidor implements Runnable {

    private Monitor safata;

    public Consumidor( Monitor s ) {
        safata = s;
     }


    public void run() {
        int num;

        for( int i=0; i < 5; i++ ){
                num = safata.afagar();
                System.out.println( "Número agafat: "+num );
                try {
                    Thread.sleep( (int)(Math.random() * 2000 ) );
                } catch( InterruptedException e ) {;}
             }
        }
    }
```

La classe principal crea dos fils: un productor i un consumidor i els executa.

```
package productor.consumidor;

public class ProductorConsumidor {
```

```
4
5   public static void main(String[] args) throws InterruptedException {
6        Monitor saf = new Monitor();
7
8       Productor p = new Productor(saf);
9       Consumidor c = new Consumidor(saf);
10      Thread productor = new Thread(p);
11      Thread consumidor = new Thread(c);
12
13      productor.start();
14      consumidor.start();
15
16
17  }
18
19 }
```

El resultat de l'execució del codi anterior és:

```
1  run:
2  Produït el número 4
3  Número agafat: 4
4  Produït el número 3
5  Número agafat: 3
6  Número agafat: 3  // El consumidor ha anat més ràpid que el productor i ha
       tornat a agafar el número 3
7  Produït el número 2
8  Produït el número 5
9  Número agafat: 5
10 Produït el número 2
11 Número agafat: 2
12 BUILD SUCCESSFUL (total time: 5 seconds)
```

Hem obviat part de codi per poder veure els possibles errors. Tal com es veu en el resultat, és un petit desastre. El model planteja alguns problemes. Ara no hi ha comunicació entre fils, per tant la producció i el consum de números és arbitrari, depèn de la velocitat del productor i del consumidors (que es troba simulada per `sleep()`). Si el consumidor és més ràpid que el productor, pot passar que agafi el mateix número diversos cops i si el productor és més ràpid pot sobreescriure el número i el consumidor mai el podrà agafar. Una solució al problema d'un productor més ràpid que un consumidor és col·locar un *buffer*, una cua que permeti guardar més d'un número.

Una cua LIFO és una estructura de dades en forma de cua en la qual l'element que entra a l'estructura en últim lloc és el primer en sortir-ne (*Last In Firts Out* en anglès). Aquest tipus de cua també s'anomena *pila*.

Crearem una cua tipus LIFO per anar guardant els números produïts pel `productor` i solucionarem la comunicació entre fils. Per fer això hem de modificar la classe `monitor`.

```
1  class Monitor {
2
3      private int cuaNumeros[] = new int[6];
4      private int segNum = 0;
5
6      private boolean cuaPlena = false;
7      private boolean cuaBuida = true;
8
9
10
11     public synchronized int afagar() {
12
13         // Si la cua es troba buida envia el fil a la cua d'espera
14          while( cuaBuida  ) //surt quan cuaBuida és false
15             {
16             try {
```

```
                wait();
            } catch( InterruptedException e ) {

                }
            }
        // Disminueix segNum perquè agafa un número i hi haurà un número menys
            a la cua
        segNum--;
        // Si no queden números
        if( segNum == 0 )
            cuaBuida = true; // La cua es troba buida
        cuaPlena = false;// La cua no està plena perquè hem consumit un número

        notify();

        // Retorna el número al fil consumidor
        return( cuaNumeros[segNum] );

    }

    public synchronized void deixar( int num ) {
       // Si la cua està plena, no hi entren més números, enviem el fil a la
           cua d'espera
        while( cuaPlena ) //Si cuaPlena és false, podrà continuar produint
            {
            try {
                wait();
            } catch( InterruptedException e ) {

                }
            }
        // afegim el nou número a la cua al primer lloc disponible
        cuaNumeros[segNum] = num;
        // augmentem els números a la cua
        segNum++;
        // Mirem si la cua està plena
        if( segNum == 6 )
            cuaPlena = true;
        cuaBuida = false; //és false perquè acaben de produir un número
        notify();
        }
    }
```

Ara, al monitor, controlem quan el consumidor vol agafar un número si hi ha números disponibles i si no n'hi ha enviem el fil a l'estat d'**en espera**. En cas contrari consumim el número, l'eliminem de la cua i despertem el fils productors que puguin estar en espera.

És possible que hi hagi productors en espera perquè la cua de números està plena.

Quan produïm també controlem si la cua de números és plena. Si es troba plena, no entren més números i, per tant, enviem a l'estat d'espera al fil productor fins que un fil consumidor el desperti i el posi en estat de preparat perquè ja ha consumit un número i queda espai a la cua. Finalment desperta els fils consumidors que puguin estar en espera ja que la cua es troba buida.

Un resultat possible és el següent:

```
run:
Produït el número 3
Número agafat: 3
Produït el número 1
Produït el número 5 //produeix dos números seguits
Número agafat: 5 // que són consumits en ordre LIFO aquí
```

```
Número agafat: 1
Produït el número 6
Número agafat: 6
Produït el número 3
Produït el número 0
Produït el número 9
Número agafat: 9
Número agafat: 0
Produït el número 7
Produït el número 7
Produït el número 3
Número agafat: 3
Número agafat: 7
Número agafat: 7
Número agafat: 3
BUILD SUCCESSFUL (total time: 9 seconds)
```

Si ens fixem en l'ordre, s'han produït diversos números seguits sense que s'hagin consumit. Quan el consumidor actua, agafa el número que més tard ha entrat a la cua de números i els va extraient en aquest ordre.

2.5 Agrupació de fils

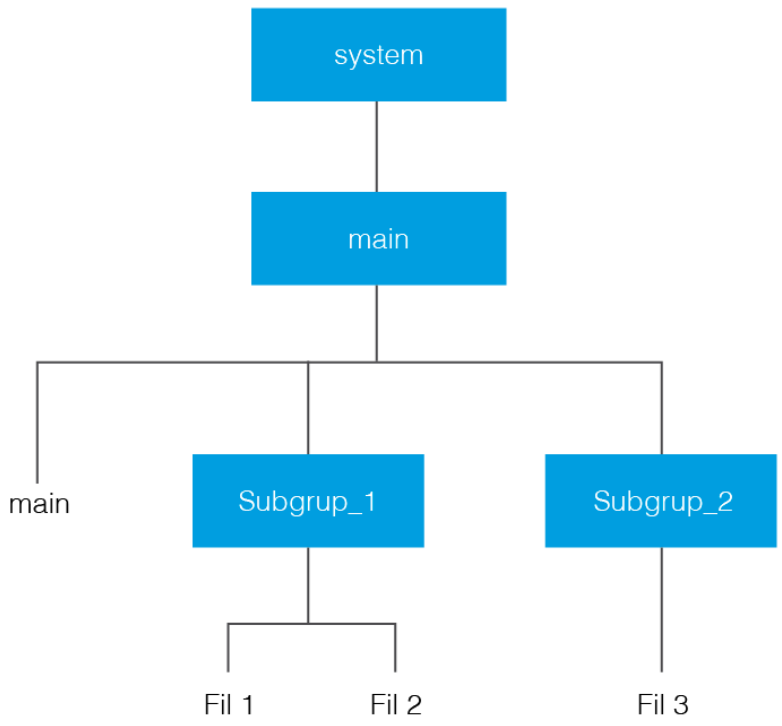
En un entorn de xarxa, un servidor rep moltes peticions a la vegada: de programes en execució, de clients, càlculs complexes, transaccions de base de dades, etc. Un fil acostuma a crear fils nous per atendre aquestes peticions. Si el número de peticions és gran, el número de fils també ho és i l'administració dels fils és complicada. Per simplificar aquesta administració es creen grups de fils. La classe `ThreadGroup` és la que defineix i implementa tot allò relacionat amb grups de fils.

2.5.1 Agrupació per defecte i creació de grups

Tots els fils d'execució a Java pertanyen a un grup. Els grups de fils són un mecanisme que ens permet agrupar fils a un únic objecte i ens permet la manipulació de tots el fils que pertanyen al grup com si fos un i no de forma individual. Podríem arrancar o adormir tots els fils d'un grup, aplicant el mètode corresponen a l'objecte grup.

En la figura 2.8 podem veure com existeix com a mínim un fil d'execució principal creat pel mètode `main` i després els diferents subgrups creats per l'aplicació. Cada grup té un número de fils. El `subgrup_1` té dos fils (`Fil1` i `Fil2`) i el `subgrup_2` un fil (`Fil3`).

Figura 2.8. Grups de fils



La classe `ThreadGroup` ens proporciona dos constructors:

- `ThreadGroup(String name)`
- `ThreadGroup(ThreadGroup parent, String name)`

Tots els fils tenen un grup i tots els grups creats a l'aplicació tenen un grup pare. Els dos constructors creen grups i els donen un nom. La diferència és l'elecció de a quin subgrup pertanyen.

El primer constructor pertany al mateix grup que el fil que fa li fa la crida. El segon constructor indica a quin pare pertany. Per exemple, si el fil principal fa una crida a `ThreadGroup(String name)` , el nou grup de fils té com a grup pare el `main`.

El exemple següent mostra com s'utilitzen els dos constructors. És crea un subgrup com a grup pare el `main` amb el nom `subGrup_1` i posteriorment es crea un altre grup que té com a pare el grup que acaben de crear:

```
public static void main (String [] args) {
ThreadGroup sGrupFils1 = new ThreadGroup ("subGrup_1");
ThreadGroup sGrupFils1_2 = new ThreadGroup (sGrupFils1, "subGrup_1_2");
}
```

La creació de grups de fils no té gaire utilitat si no li assignem fils al grup.

En el següent codi es mostra com es crea un grup anomenat subgrup 2 i posteriorment com es crea un fil (Fil 1) i l'assignem al grup.

```
ThreadGroup sgr = new ThreadGroup ("subgrup 2");
Thread fil = new Thread (sgr, "Fil 1");
```

2.5.2 Classe ThreadGroup

`ThreadGroup` conté tota la funcionalitat per la manipulació de grups de fils. Un grup pot tenir un número de fils il·limitat. Habitualment tots el fils d'un grup tenen alguna cosa en comú: qui els va crear, quina funció tenen o quan han d'executar-se o finalitzar. En el següent codi creem un grup (`subGrup 1`) que té com a pare `main`. Després creem dos fils i li diem que pertanyen al grup creat. Creem un altre grup (`subgrup 2`) i creem un fil (`Fil 3`) que pertany al nou grup. És l'esquema de la figura 2.8.

El mètode `activeGroupCount()` ens diu quants grups hi ha actius i `currentThread ().getThreadGroup ()`, com es diu el grup del fil actiu. `list()` és un mètode que dóna els detalls del grup actiu i els seus subgrups (nom, prioritat...)

```
public class GrupsFils {

    public static void main(String[] args) {
 ThreadGroup sg = new ThreadGroup ("subgrup 1");
      Thread fil1 = new Thread (sg, "Fil 1");
      Thread fil2 = new Thread (sg, "Fil 2");
      sg = new ThreadGroup ("subgrup 2");
      Thread fil3 = new Thread (sg, "Fil 1_1");
      sg = Thread.currentThread ().getThreadGroup ();
      int agc = sg.activeGroupCount ();
      System.out.println ("Grup de fil actiu " + sg.getName () +
                          ". Grups actius: " + agc);
      sg.list ();

    }
}
```

El codi anterior té el següent resultat:

```
run:
Grup de fil actiu main. Grups actius: 2
java.lang.ThreadGroup[name=main,maxpri=10]
    Thread[main,5,main]
    java.lang.ThreadGroup[name=subgrup 1,maxpri=10]
    java.lang.ThreadGroup[name=subgrup 2,maxpri=10]
BUILD SUCCESSFUL (total time: 1 second)
```

La prioritat d'un grup és la prioritat més alta dels fils que pertanyen al grup.

Els mètodes següents de la classe `ThreadGroup` únicament tenen efecte sobre el grup, però no sobre els fils del grup:

- `getMaxPriority`: retorna la prioritat del grup.
- `setMaxPriority`: modifica la prioritat del grup, no la dels fils. Per tant, si disminuïm la prioritat del grup, pot ser que hi hagi un fil amb una prioritat més alta que el grup.
- `getDaemon` i `SetDaemon`: retorna si és un grup dimoni i assigna a un grup de fils si es comportarà com a dimoni o no.

- `getName`: retorna el nom del grup.
- `getParent`: retorna el pare d'un grup.
- `parentOf`: ens diu si el grup de fil pertany al grup que li passem per paràmetres.
- `toString`: converteix en *string* els detalls del grup (nom, prioritat, etc).

2.6 Gestió de fils per part del sistema operatiu

La màquina virtual de Java (JVM, *Java Virtual Machine* en anglès) és un sistema multifil, i és l'encarregada de gestionar els fils, d'assignar el temps d'execució, les prioritats, etc. És l'encarregada de decidir quin fil entra a executar-se. La gestió que fa dels fils és similar a la que té el sistema operatiu quan gestiona més d'un procés. La màquina virtual de Java és un procés dins del sistema operatiu, és a dir, ja que el fils s'executen a la màquina virtual, comparteixen recursos.

2.6.1 Planificació de fils

La manera que té el sistema operatiu de comportar-se amb l'execució de fils és un tema important en la programació multifil. La planificació fa referència a quina política s'ha de seguir per decidir quin fil agafa el control del processador i en quin moment. També s'ha de planificar quan ha de deixar d'executar-se. Java està dissenyat per tal que sigui un sistema portable. El sistema de fils ha de ser suportat per qualsevol plataforma i, ja que cada plataforma funciona de forma diferent, el tractament dels fils ha de ser molt general. Com a característiques més importants de la planificació de fils tenim:

- Tots els fils tenen assignada una prioritat i l'encarregat de decidir quin fil s'executa ha de garantir que els fils amb prioritat més alta tinguin preferència. Però això no implica que en un moment donat un fil amb prioritat més baixa estigui executant-se.
- S'ha de garantir que tots els fils s'executin en algun moment.
- El temps assignat a cada fil per fer ús del processador pot aplicar-se o no depenent del sistema operatiu en el qual s'executa la màquina virtual.

Amb aquestes premisses el programador ha de tenir en compte a l'hora de programar que els fils poden intercalar la seva execució en qualsevol moment. Si volem una ordenació en l'execució de fils, hem d'afegir codi a la nostra programació per tal que passi.

2.6.2 Prioritat de fils

Les prioritats dels fils a Java són inicialment la mateixa que els fils creadors. Per defecte tenen una prioritat de 5. El rang de prioritats va des d'1, definida amb la propietat de `Thread` `MIN_PRIORITY` a 10 definida per `MAX_PRIORITY`. La prioritat per defecte amb valor 5 és `NORM_PRIORITY`.

Els fils amb una prioritat més alta seran posats a execució per part del planificador en primer lloc i els de prioritat més baixa quan els de prioritat superior estiguin bloquejats o hagin acabat l'execució. La classe `Thread` té dos mètodes que permeten veure i modificar les prioritats dels fils:

- `setPriority()` permet modificar la prioritat d'un fil:

```
fil.setPriority(8);
```

- `getPriority()` torna la prioritat d'un fil:

```
System.out.println"(Prioritat fil " +  fil.getPriority());
```

Swing és una API de Java que proporciona la interfície gràfica d'usuari a Java. És l'API que substitueix a l'altre API gràfica (AWT) que únicament permet una aparença gràfica igual al sistema nadiu de la plataforma en la qual s'executa. En canvi Swing també permet una aparença que no té res a veure amb la plataforma nativa.

Els fils de les classes que generen interfícies d'usuari (Swing i AWT) tenen per defecte una prioritat de 6. D'aquesta manera la interacció amb l'usuari és més fluïda.

Si no hi ha cap esdeveniment d'entrada els fils estan a l'espera. Així els altres fils no són perjudicats per ells. Quan hi ha un esdeveniment d'entrada per ratolí o teclat els fils responen immendiatament.

El mètode `yield()` fa que un fil que s'està executant passi de nou a l'estat i a la cua de preparats. D'aquesta manera permet que altres fils de prioritat superior o de la mateixa prioritat es puguin executar. Però pot passar que un cop passat a preparat torni a ser escollit per executar-se de nou.

En el següent codi d'exemple es mostra com creem dos fils que competiran per qui acaba primer la seva execució. En condicions normals, amb la mateixa prioritat, el guanyador s'aniria alternant, depenent de quin escollís l'organitzador de la cua de fils. Però al modificar les prioritats dels fils, al primer li assignem la prioritat més baixa i al segon la més alta. D'aquesta manera ens assegurem que sempre guanyi el segon. Això farà que el segon acapari el processador fins que acabi la seva execució.

Els resultats també dependran del sistema operatiu sobre el qual s'executa la màquina virtual, ja que les prioritats poden variar d'un sistema a un altre.

Utilitzem el mètode `setName()` de la classe `Thread` per assignar-li un nom a cada fil.

```java
package prioritatsfils;

public class PrioritatsFils implements Runnable {

String strImprimir;
    public PrioritatsFils(String strP) {
        strImprimir=strP;
    }

    public void run(){
        for(int x=0;x<100;x++){
            System.out.println(strImprimir);
        }

    }

    public static void main(String[] args) {
        PrioritatsFils objRunnable1 = new PrioritatsFils("Corredor 1");
        PrioritatsFils objRunnable2 = new PrioritatsFils("Corredor 1");

        //Creem el fil amb l'objecte runnable
        Thread primer = new Thread(objRunnable1);
        //Assignem un nom al fil primer
        primer.setName("Corredor 1");

        Thread segon = new Thread(objRunnable2);
        //Assignem un nom al fil primer
        segon.setName("Corredor 1");
        //Canviem la prioritat de fil primer i li posem les més baixa
        primer.setPriority(Thread.MIN_PRIORITY);

        //Mostrem el nom del fil i la seva prioritat
        System.out.println("Prioritat " + primer.getName()+ ": " + primer.
            getPriority());

        //Canviem la prioritat de fil segon i li posem les més altes
        segon.setPriority(Thread.MAX_PRIORITY);
        System.out.println("Prioritat " + segon.getName()+ ": " + segon.
            getPriority());

        // Cridem al mètode start() per tal que comenci per posar-los a
            preparats
        primer.start();
        segon.start();

        System.out.println("Final Fil Principal");
            }
}
```

2.6.3 Fils dimonis

Quan creem un fil i abans de cridar el mètode `start()` podem indicar que el fil serà executat com un dimoni (*daemon* en anglès). Això permetrà al fil executar-se a un segon pla. Els fils dimonis tenen la prioritat més baixa. Si un fil dimoni crea un altre fil, aquest també serà un dimoni. Aquesta qualitat s'hereta. I no es pot canviar un cop el fil ha estat inicialitzat. Es troba a l'estat de preparat.

Garbage collector és un dimoni creat per la màquina virtual de Java, que s'encarrega d'alliberar memòria ocupada per objectes que no estan referenciats.

Per indicar que un fil és un dimoni s'utilitza el mètode `setDaemon(true)`. En el cas de `setDaemon(false)`, el fil és d'usuari, l'opció per defecte. Els fils dimonis finalitzen quan finalitza l'últim fil d'usuari i l'aplicació acaba.

```
fil.setDaemon(true);
```

Per veure si un fil és un dimoni, s'utilitza el mètode `isDaemon()` que retornarà un booleà indicant si el fil és un dimoni o no.

```
System.out.println(fil.isDaemon());
```

El següent exemple mostra com es crea un dimoni. I després com, des d'aquest dimoni, es crea un *array* de 10 dimonis. Hem de veure que els fils dimonis creats pel dimoni inicial no els indiquem amb `setDaemon(true)`, però hereten aquesta qualitat del fil que l'ha creat.

Si volguéssim crear fils que no fossin dimoni, hauríem d'indicar-ho expressament amb `setDaemon(false)`, abans del mètode `start()` del fil.

```
package filsdimonis;
import java.io.IOException;

public class FilsDimonis extends Thread{
  private static  final int MIDA = 10;
  private Thread[] fils = new Thread[MIDA];
  public FilsDimonis(){
    setDaemon(true);
    start();
  }
  public void run(){
    for(int i = 0; i<MIDA; i++){
      fils[i]= new CrearDimoni(i);
    }
    for(int i=0; i<MIDA; i++){
    System.out.println("fils[" + i + "].isDaemon() = " + fils[i].isDaemon() );
    }
    while(true) yield();
  }

class CrearDimoni extends Thread{
  public CrearDimoni(int i){
    //si un fil no ens interessa que sigui dimoni, ho hem d'indicar aquí
    //setDaemon(false); abans del mètode start()
    //en cas contrari, hereta la qualitat de dimoni del seu pare.

    System.out.println("Dimoni creat " + i );
    start();
  }
  public void run(){
    while (true) yield();
  }
}

public static void main(String[] args) throws IOException{
    Thread fil = new FilsDimonis();
    System.out.println("isDaemon() = " +  fil.isDaemon() );
    System.out.println("Pica una tecla per finalitzar");
    System.in.read();
  }
    }
```

2.7 Documentació d'aplicacions i depuració d'aplicacions multifils

Una documentació eficient soluciona molts problemes a l'hora de mantenir l'aplicació, ja que les persones encarregades del manteniment, millores o ampliacions poden no ser les mateixes que la van construir. O fins i tot poden anar canviant en cada una de les fases de la creació de l'aplicació.

Un programador ha d'intentar comentar el seu codi per tal que sigui el màxim d'entenedor per als altres programadors.

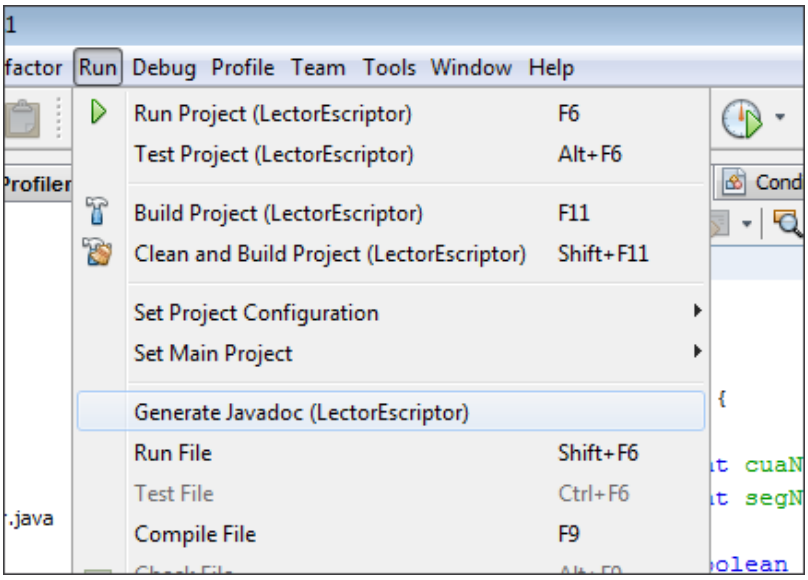
Java ens proporciona una eina per crear documentació tècnica a nivell d'implementació, s'anomena **JavaDoc**. JavaDoc analitza el codi i els comentaris sobre el codi i genera una documentació que pot ser en HTML amb un llistat de les classes i mètodes. Fins i tot indica si hem utilitzat algun mètode obsolet.

És molt important, doncs, que els codi estigui comentat amb /* */.

A NetBeans per generar la documentació amb JavaDoc hem d'anar al menú *Executar* i clicar sobre l'opció *Generar Javadoc* (figura 2.9).

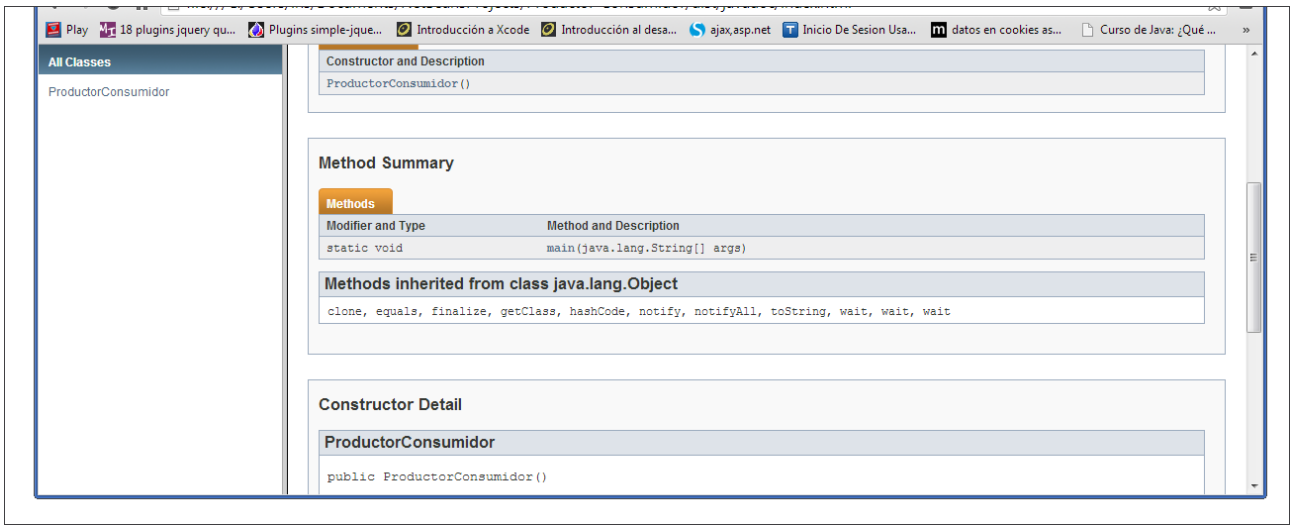
NetBeans és entorn de desenvolupament integrat (IDE, *Integrated Development Environment*, en anglès). Pensat per desenvolupar aplicacions, va ser creat en Java per Sun MicroSystems i és lliure i gratuït.

Figura 2.9. Crear javadoc



El resultat és un document HTML (figura 2.10).

FIGURA 2.10. Document creat per javadoc



2.7.1 Depuració d'aplicacions multifils

Depurar aplicacions és una de les tasques que més sovint fem els programadors. Ens permet verificar l'estat de la nostra aplicació en un punt en concret. Posar marques al codi o utilitzar les eines de depuració (*debugging*, en anglès) amb totes les utilitats que ens proporcionen alguns IDE, és una tasca imprescindible per revisar la nostra aplicació. Depurar una aplicació seqüencial en la qual únicament hi ha una línia d'execució és simple amb una eina de depuració. Anem avançant pas a pas al fil d'execució i anem mirant valors de variable, condicions... fins que acabem. Però en la programació multifil hi ha més d'un fil d'execució. Haurem d'anar consultant cada fil per separat per veure el seu funcionament i quins valors va prenent.

Veurem com podem depurar una aplicació multifil i utilitzarem l'eina per detectar possibles errors com l'interbloqueig a una aplicació.

Per fer les proves utilitzarem l'IDE de Netbeans. Amb altres IDEs el funcionament és similar. Per provar com consultar els fils obrirem un projecte i seleccionarem el projecte a la finestra de projectes i triarem l'opció *Debug*. El projecte que analitzarem és el següent:

```
1  package nauespaial;
2
3  import java.awt.Color;
4  import java.awt.Graphics;
5  import java.awt.Graphics2D;
6  import java.awt.Image;
7  import java.util.Random;
8  import javax.swing.ImageIcon;
9  import javax.swing.JFrame;
10 import javax.swing.JPanel;
11
12 /**
```

```
13  *
14  * @author jaleo
15  */
16 public class NauEspaial  extends javax.swing.JFrame {
17
18     public NauEspaial() {
19         initComponents();
20     }
21
22
23     @SuppressWarnings("unchecked")
24
25     private void initComponents() {
26
27         setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
28         setBackground(new java.awt.Color(255, 255, 255));
29
30         javax.swing.GroupLayout layout = new javax.swing.GroupLayout(
               getContentPane());
31         getContentPane().setLayout(layout);
32         layout.setHorizontalGroup(
33             layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
                   LEADING)
34             .addGap(0, 400, Short.MAX_VALUE)
35         );
36         layout.setVerticalGroup(
37             layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
                   LEADING)
38             .addGap(0, 300, Short.MAX_VALUE)
39         );
40
41         pack();
42     }
43
44
45     public static void main(String args[]) {
46
47         try {
48             for (javax.swing.UIManager.LookAndFeelInfo info : javax.swing.
                   UIManager.getInstalledLookAndFeels()) {
49                 if ("Nimbus".equals(info.getName())) {
50                     javax.swing.UIManager.setLookAndFeel(info.getClassName());
51
52                     break;
53                 }
54             }
55
56         } catch (ClassNotFoundException ex) {
57             java.util.logging.Logger.getLogger(NauEspaial.class.getName()).log(
                   java.util.logging.Level.SEVERE, null, ex);
58         } catch (InstantiationException ex) {
59             java.util.logging.Logger.getLogger(NauEspaial.class.getName()).log(
                   java.util.logging.Level.SEVERE, null, ex);
60         } catch (IllegalAccessException ex) {
61             java.util.logging.Logger.getLogger(NauEspaial.class.getName()).log(
                   java.util.logging.Level.SEVERE, null, ex);
62         } catch (javax.swing.UnsupportedLookAndFeelException ex) {
63             java.util.logging.Logger.getLogger(NauEspaial.class.getName()).log(
                   java.util.logging.Level.SEVERE, null, ex);
64         }
65
66            NauEspaial f = new NauEspaial();
67             f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
68             f.setTitle("Naus Espaials");
69             f.setContentPane(new PanelNau());
70             f.setSize(480, 560);
71             f.setVisible(true);
72
73         }
74     }
```

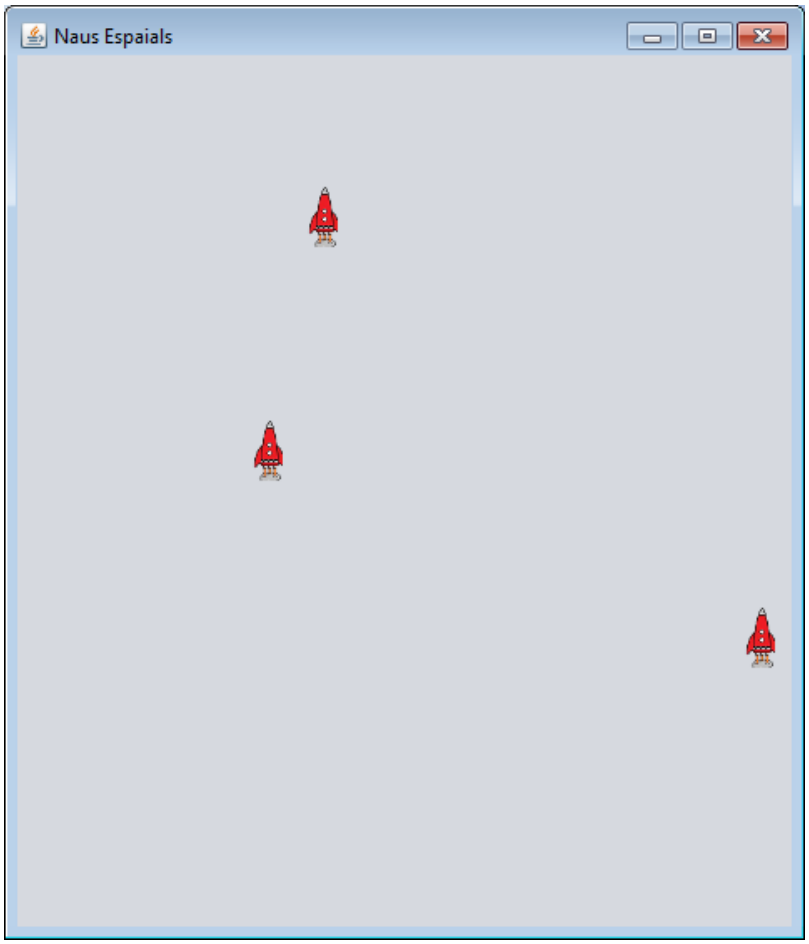
```
75
76
77
78  class PanelNau extends JPanel implements Runnable{
79
80      private int numNaus=3;
81      Thread[] filsNau;
82      Nau[] nau;
83
84
85
86      public PanelNau(){
87
88         filsNau = new Thread[numNaus]; //Creo dos hilos
89         nau = new Nau[numNaus];
90         for (int i=0;i<filsNau.length;i++){
91            filsNau[i]= new Thread(this);
92            filsNau[i].setName("Fil Nau-"+i);
93
94            Random rand = new Random();
95            int velocitat=rand.nextInt(50);
96            int posX=rand.nextInt(100)+30;
97            int posY=rand.nextInt(100)+30;
98            int dX=rand.nextInt(3)+1;
99            int dY=rand.nextInt(3)+1;
100           nau[i]= new Nau(posX,posY,dX,dY,velocitat);
101
102          }
103
104           for (int i=0; i<filsNau.length; ++i)
105              filsNau[i].start();
106 }
107
108
109 public void run(){
110      for (int i=0; i<filsNau.length; ++i){
111       while (filsNau[i].currentThread()== filsNau[i]) {
112             try {
113                 filsNau[i].sleep(nau[i].velocitat()); //
114                 nau[i].moure();
115              }catch (InterruptedException e) { }
116             repaint();
117          }
118
119       }
120 }
121
122
123   public void paintComponent(Graphics g) {
124       super.paintComponent(g);
125       for(int i=0; i<nau.length;++i)
126           nau[i].pinta(g);
127     }
128   }
129
130
131
132
133 class Nau {
134     private int x, y;
135     private int dsx, dsy, v; //desplaçaments, v sleep
136     private int tx = 10; //marge x
137     private int ty = 10; //marge i image
138
139     private String img = "/images/nau.jpg";
140     private Image image;
141
142
143
144       public Nau(int x, int y, int dsx, int dsy, int v ){
```

```
145         this.x=x;
146         this.y=y;
147         this.dsx= dsx;
148         this.dsy=dsy;
149         this.v=v;
150         image = new ImageIcon(Nau.class.getResource("/images/nau.png")).
                getImage();
151
152     }
153
154     public int velocitat (){ //sleep en milisegons
155         return v;
156     }
157
158     public void moure (){
159       x=x + dsx;
160       y=y + dsy;
161     // arriva als marges
162       if ( x>= 450-tx || x<= tx)
163           dsx = -dsx;
164       if ( y >= 500- ty || y<=ty )
165          dsy = -dsy;
166
167     }
168     public void pinta (Graphics g){
169           Graphics2D g2d = (Graphics2D)g;
170           g2d.drawImage(this.image, x, y, null);
171     }
172
173
174 }
```

Si no tenim posats punts de ruptura l'aplicació s'executarà fins a acabar. L'aplicació és un panell en el qual es mostren unes imatges que es mouen. Cada imatge (nau espacial) és controlada per un fil. L'aplicació crea 3 fils, per tant hi haurà tres imatges al panell en moviment (figura 2.11).

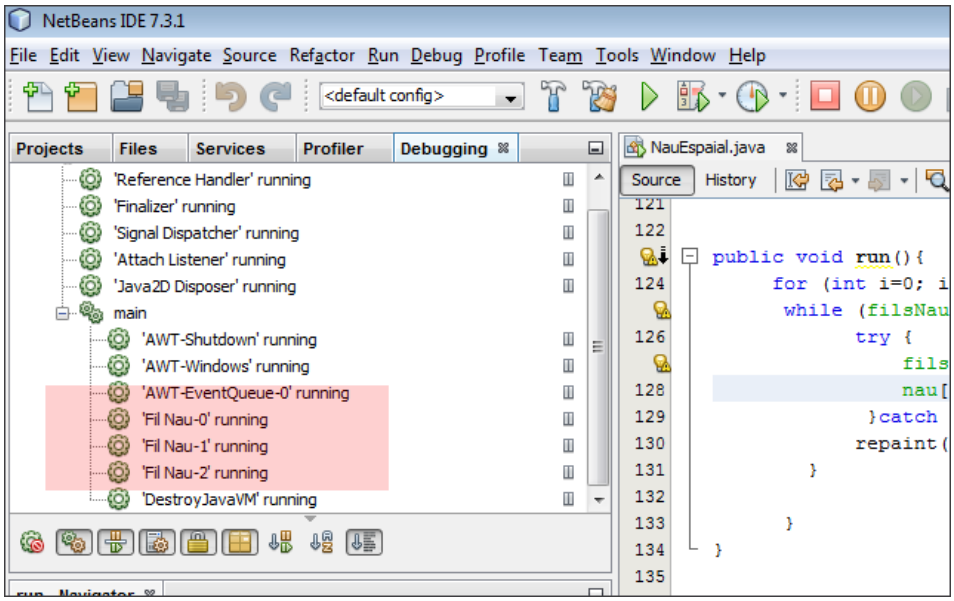
Els punts de ruptura (*breakpoints* en anglès) són marques que indiquen a l'IDE on cal aturar l'execució del codi quan es faci en mode depuració.

Figura 2.11. Aplicació de naus en funcionament



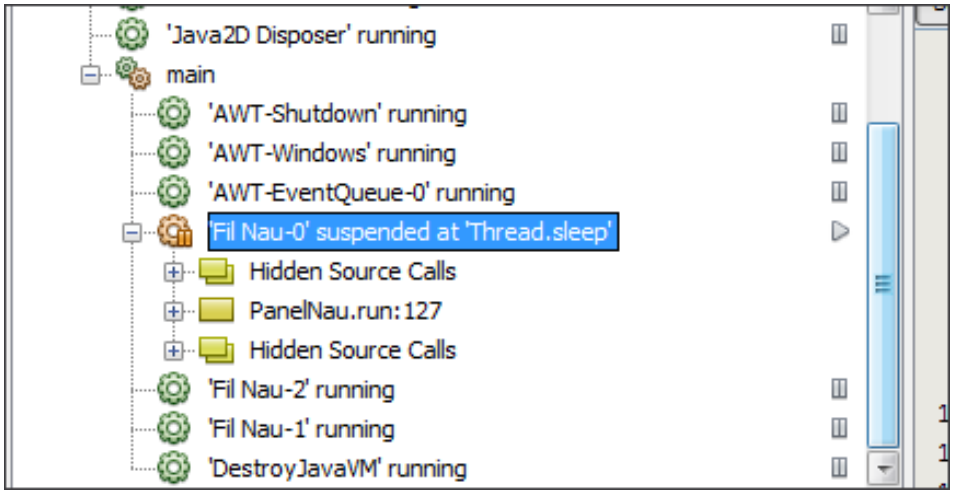
A la finestra de depuració podem veure els tres fils creats (figura 2.12), amb el nom de *Fil Nau 0*, *Fil Nau 1* i *Fil Nau 2*. La resta de fils creats són el fil principal i els del sistema.

Figura 2.12. Depuració de fils



A la dreta del nom del fil hi ha un símbol de pausa. Si el clquem el fil s'aturarà. El símbol es convertirà en el triangle de *play*. Quan el fil es troba aturat (figura 2.13) podem veure que surt una carpeta que es pot expandir. Dins hi veiem la llista de trucades del fil. Ja que el fil està aturat, la nau (la imatge que es mou pel panell) s'atura també.

Figura 2.13. Fil aturat

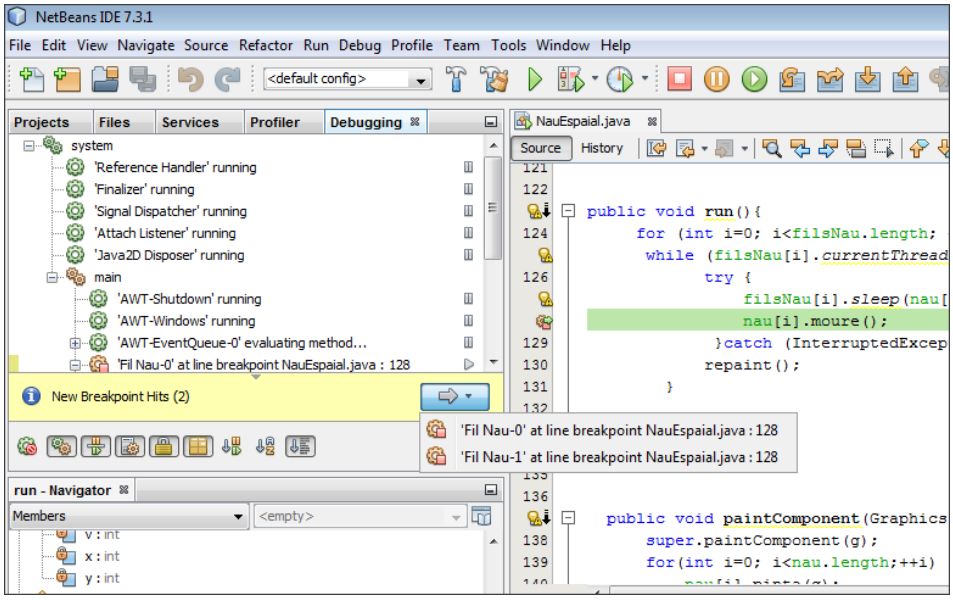


Si hem posat algun punt d'interrupció, l'aplicació es comporta com si no treballés amb fils. L'execució s'atura en aquest punt.

Per posar un punt d'interrupció hem de picar al marge esquerre de l'editor al costat del número de línia en el qual volem posar el punt d'interrupció. Apareixerà un quadrat vermell que marca el punt.

En la figura 2.14 es veu que el punt d'interrupció està posat quan el fil executa el moviment de la imatge.

Figura 2.14. Punt d'interrupció

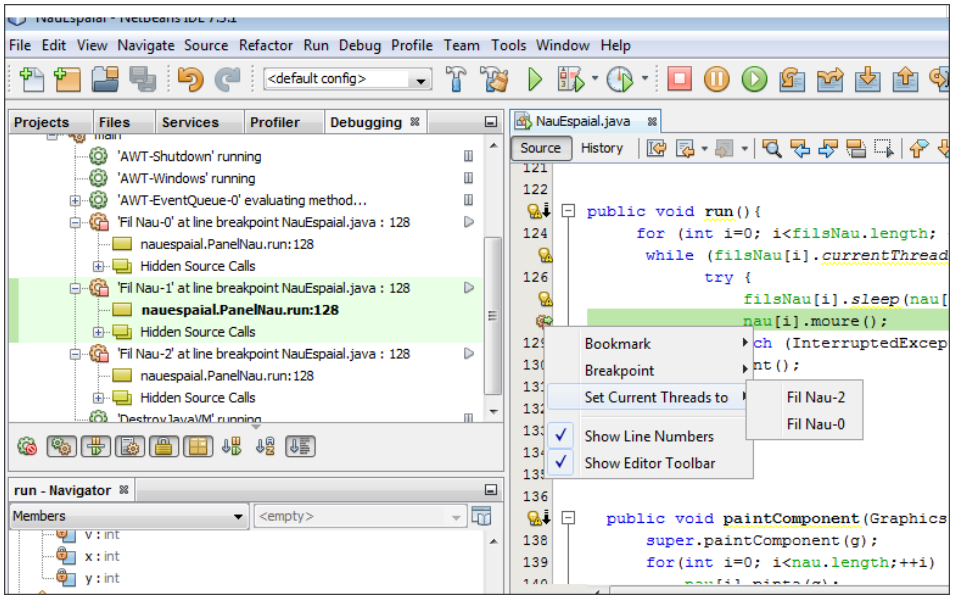


En la finestra de depuració es veu com s'està executant el primer fil que arriba. Clicant les tecles de funció F7 o F8 anirà avançant la línia de programa sobre el fil que està actiu. La resta de fils estan preparats per executar-se. Si volem que els altres fils comencin haurem de cliar sobre la fletxa que està sota la finestra

de depuració i seleccionar el fil que volem executar. Un cop tots els fils s'estan executant, podem veure l'execució d'un o altre fil picant sobre el fil a la finestra de depuració. El fil que estem controlant apareix amb un marc verd.

A l'esquerra del codi apareixen les icones de la roda dentada que ens indiquen on es troben aturats els fils que no estem depurant. Si piquem sobre la roda dentada amb al botó dret podem veure quin és el fil que està aturat en aquest punt (figura 2.15).

FIGURA 2.15. Intercanviant l'execució dels fils



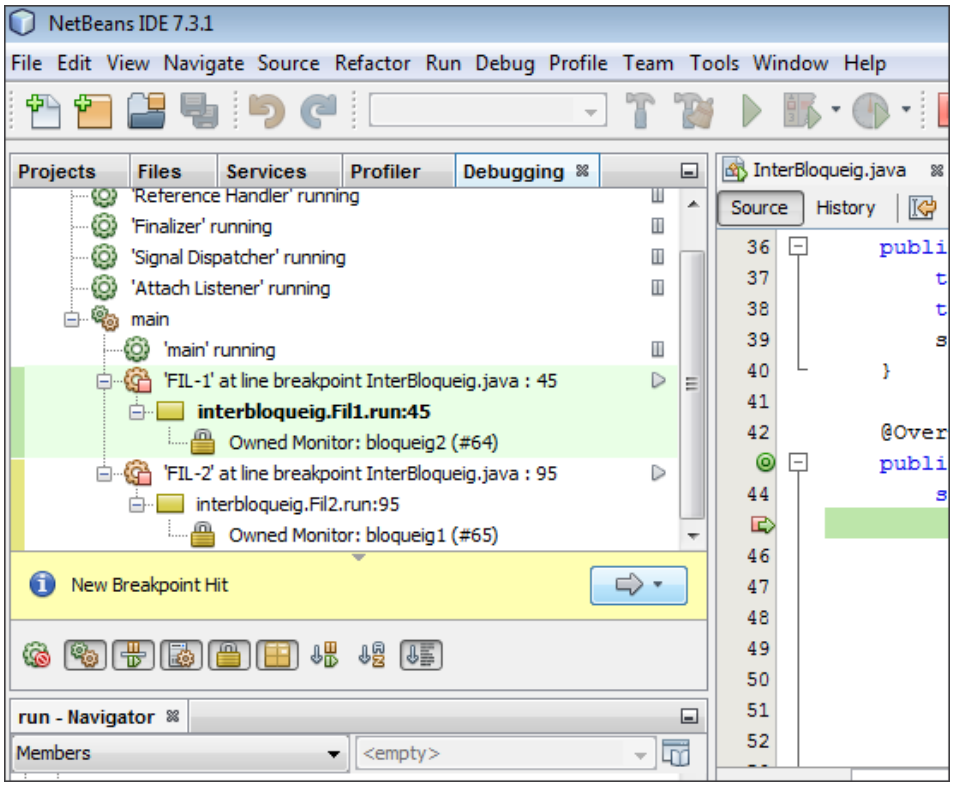
Si volem canviar per continuar depurant un altre fil el seleccionarem de la llista. Únicament podem depurar un fil amb un punt de ruptura (*breakpoint*, en anglès). Quan cliquem sobre F7 o F8 per anar avançant únicament avança un fil, els altres no s'aturen al punt d'interrupció.

Depurar aplicacions és bastant simple i molt útil. Els IDE cada vegada proporcionen eines que ens ajuden més a la correcció d'errors. Netbeans a la seva eina de depuració inclou una funció que detecta interbloquejos automàticament.

Per utilitzar-la hem de crear punts de ruptura. Els punt d'interrupció es posen on nosaltres pensem que hi ha el punt crític de la nostra aplicació i on podria ser la font dels errors. En aquest cas els posarem al mètode `run()` de la classe que hereta `threads` o implementa `runnable`.

Executem l'aplicació en mode depuració i veiem com s'atura als punt d'interrupció (figura 2.16).

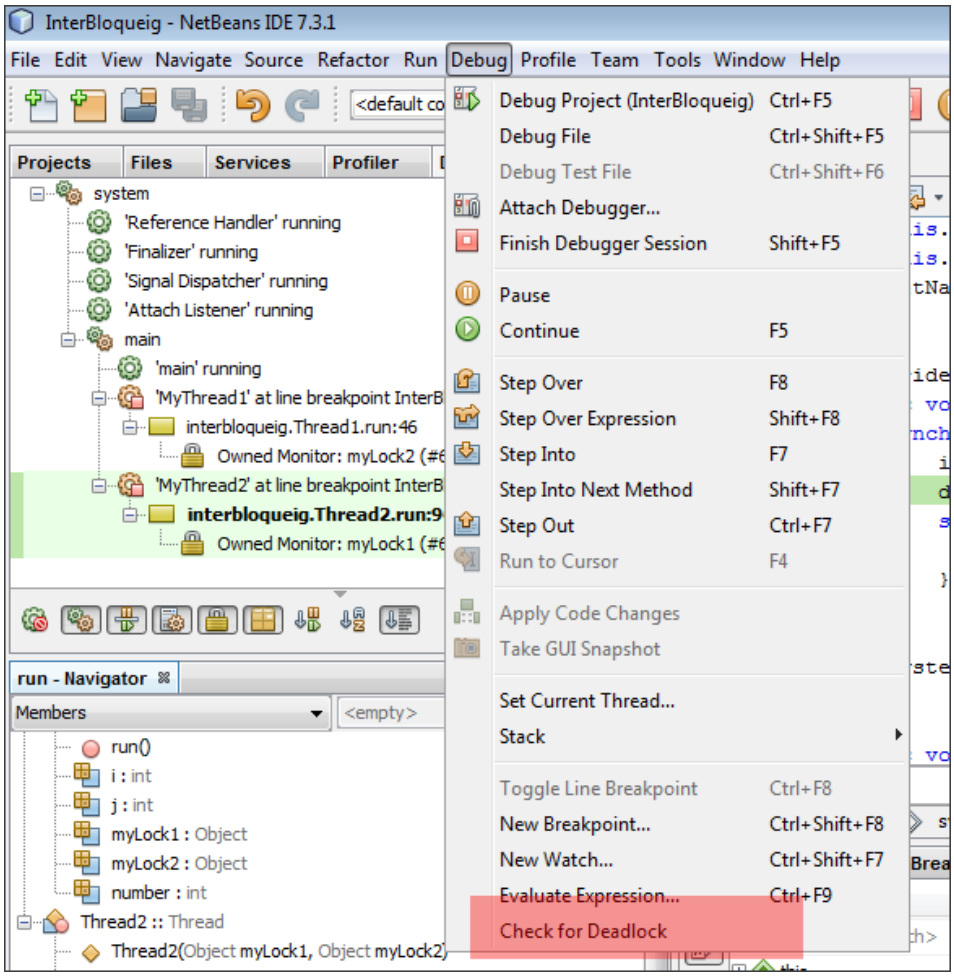
Figura 2.16. Punt d'interrupció



En aquest punt podem continuar l'execució (tecles de funció F7 o F8) i veure com avança l'aplicació i van variant els valors de les variables, la línia d'execució, etc.

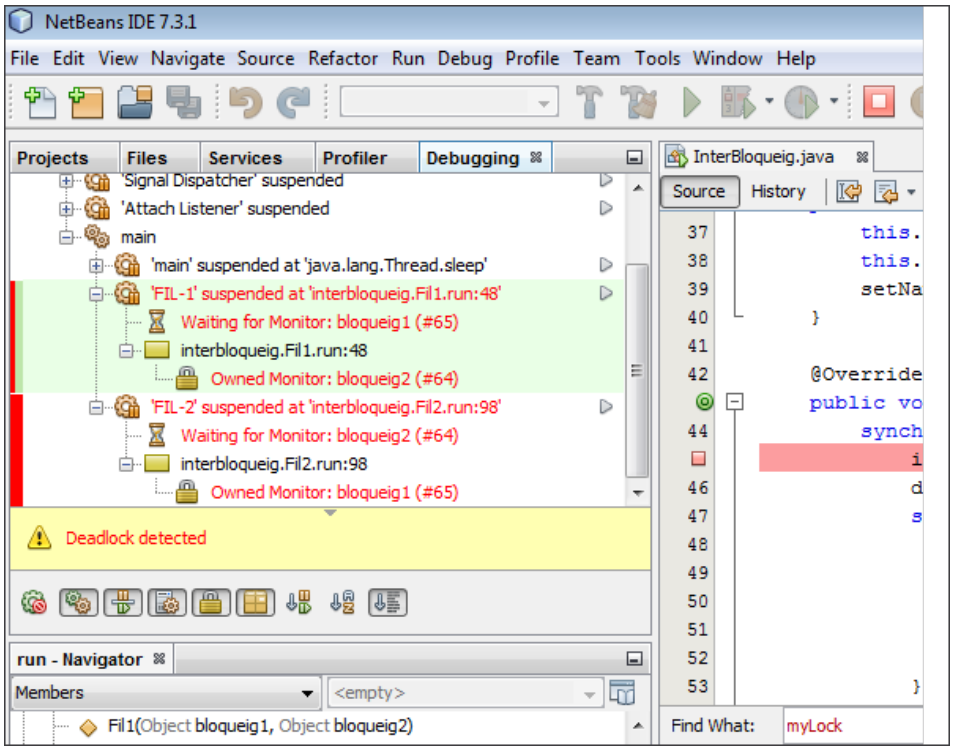
Per detectar l'interbloqueig hem d'acabar l'execució. Podem clicar sobre cada botó dret i seleccionar *resume*, podem clicar F5 o clicar sobre la icona de la barra d'eines del triangle blanc (*play*) sobre fons verd. El resultat que tindrem és l'aplicació executant-se. Per veure si s'ha produït un interbloqueig hem d'anar al menú de depuració i clicar sobre l'opció *Check for DeadLock* (figura 2.17).

Figura 2.17. Examinar interbloqueig



Automàticament Netbeans ens dirà si existeixen o no interbloquejos. Ens mostrarà un missatge dient si n'ha trobat o no (figura 2.18).

Figura 2.18. Missatge d'interbloqueig



La finestra de depuració ens diu on es troben els fils bloquejats en situació d'interbloqueig.

Les eines de depuració són molt útils per trobar possibles errades de programació. Quan programem utilitzant molts fils, la depuració de les aplicacions és fonamental.

Sòcols i serveis

Josep Cañellas Bornas

Índex

Introducció **5**

Resultats d'aprenentatge **7**

1 Programació de comunicacions en xarxa **9**
1.1 Aspectes teòrics de la comunicació . . . 10
1.2 Rols client i servidor . . . 11
1.3 Bases de la comunicació entre aplicacions . . . 12
1.3.1 Arquitectura d'Internet: una estructura de capes . . . 13
1.3.2 Capçaleres i dades . . . 15
1.3.3 Contingut de la informació a cada capa . . . 17
1.4 Elements de programació d'aplicacions en xarxa . . . 28
1.4.1 Adreçament . . . 28
1.4.2 Referències remotes i obtenció de recursos . . . 30
1.4.3 Sòcols . . . 37
1.4.4 Implementació de sòcols no orientats a connexió . . . 38
1.4.5 Implementació de sòcols orientats a connexió . . . 46

2 Generació de serveis en xarxa **55**
2.1 Serveis i protocols estàndards . . . 57
2.2 Servei de transferència de fitxers . . . 57
2.3 Serveis de correu electrònic . . . 61
2.3.1 Format del missatge de correu electrònic . . . 62
2.3.2 Adjunció de fitxers i format HTML en el cos del missatge . . . 64
2.3.3 Servei de transmissió de missatges . . . 65
2.3.4 Servei d'accés a les bústies d'usuari . . . 67
2.3.5 Implementació del client usant la biblioteca JavaMail . . . 69
2.4 Servei WWW . . . 73
2.4.1 El protocol HTTP . . . 74
2.4.2 El llenguatge HTML . . . 80
2.4.3 Implementació de clients web . . . 85
2.4.4 Implementació servidors web . . . 95
2.5 Altres serveis . . . 99
2.6 Escalabilitat dels serveis . . . 100
2.6.1 Verificació del servei i monitorització dels components . . . 101
2.6.2 Disseny basat en components . . . 103

Introducció

D'un temps ençà, han proliferat un gran nombre d'idees que aprofiten la capacitat de connexió dels dispositius informàtics per mantenir qualsevol tipus de dades tan accessible com sigui possible i de la forma més senzilla de manera que la intervenció de l'usuari sigui sempre mínima. És habitual disposar d'una petita aplicació que sincronitza part de la informació emmagatzemada en fitxers de tots els nostres dispositius de manera que un canvi en un d'ells repercuteixi en tots els altres. També sol ser habitual mantenir magatzems de dades en línia en els quals podem guardar-hi fotografies o documents. És el que es coneix com a emmagatzematge en els núvols. Fins i tot comencem a treballar amb aplicacions que s'executen en el nostre navegador i que disposen d'una potència i expressivitat semblant a les aplicacions locals. Només cal que penseu amb les aplicacions que Google, Yahoo o d'altres empreses del sector posen al nostre servei, per adonar-nos-en. Són aplicacions que no necessiten instal·lació, que no ens ocupen espai i que es poden executar des de qualsevol ordinador sempre que tingui connexió a Internet.

Però la gran revolució de les comunicacions ha vingut sobretot de la mà dels dispositius amb una quantitat de memòria limitada, ja que la gran proliferació de mòbils o tauletes ha generat també un allau d'aplicacions executades de manera distribuïda. És a dir, amb una part de codi executant-se en local que es comunica i fa peticions d'execució d'altres parts que s'executen en màquines remotes. És una manera d'incrementar la capacitat d'emmagatzematge i la capacitat de càlcul d'aquests dispositius que inunden les nostres vides.

Les aplicacions distribuïdes necessiten, però, elements específics que no trobàvem en el desenvolupament d'aplicacions locals. D'entrada caldrà possibilitar una connexió permanent entre les diferents parts de l'aplicació, però el fet que l'execució es realitzi en diferents màquines que no comparteixen el seu espai de memòria obligarà també a dissenyar protocols de comunicació que permetin a totes les parts disposar de la informació necessària per dur a terme les seves tasques malgrat que part d'aquesta informació s'hagi pogut generat en dispositius externs.

És evident que l'increment de facilitat d'ús ofert als usuaris ha d'implicar un increment de la complexitat en el desenvolupament. A la unitat anomenada *Sòcols i Serveis* estudiarem com enfrontar-nos al repte de crear aplicacions distribuïdes amb capacitat de comunicar-se i treballar de forma conjunta en pro d'un objectiu comú.

Al primer apartat anomenat "Programació de comunicacions en xarxa" descobrirem quina és la base que suporta qualsevol comunicació actual entre dispositius. Analitzarem els aspectes teòrics, les aplicacions i protocols que possibiliten la comunicació, centrant-nos sobretot en aquells que més necessitem quan desen-

volupem aplicacions que necessitin comunicar-se. Finalment, a la darrera part d'aquest apartat posarem en pràctica tot el que hem anat aprenent i veurem com es materialitza en un llenguatge concret com és Java. Estudiarem les classes i biblioteques que els llenguatge posa a la nostra disposició per aconseguir desenvolupaments distribuïts eficients i segurs.

En el segon i darrer apartat, anomenat "Generació de serveis en xarxa", donarem una ullada als principals serveis estàndards que ens ofereix Internet i analitzarem com podem dur a terme la implementació d'aquest, ja sigui des de la vessant del client que sol·licita el servei o des de la del servidor que espera, resol i contesta les peticions. El principals serveis analitzats seran els de transferència de fitxers, els de correu electrònic o els de servidors de pàgines i aplicacions web.

A més d'estudiar els serveis, en el darrer punt de l'apartat ens aturarem a veure de quina manera podem fer desenvolupaments escalables que permetin amb petites modificacions adaptar el codi al creixement que les aplicacions vagin experimentant. Sovint aquest és el punt més delicat del desenvolupament distribuït ja que sense un disseny escalable, els avantatges que suposen aquest tipus d'aplicacions es dilueixen davant la complexitat derivada d'haver d'implementar-se de bell nou cada cop que les necessitats de creixement requereixin canvis.

La unitat combina els coneixement teòrics i pràctics per tal que l'alumne sigui capaç de fer front demandes pràctiques però assentades sobre bases teòriques que l'ajudin a extrapolar els coneixements en situacions pràctiques similars que podrà trobar-se durant la seva vida laboral. Per això és important que l'alumne a més de realitzar una lectura analítica resolgui les activitats plantejades i els exercicis d'autoavaluació.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a serà capaç de:

1. Programar mecanismes de comunicació en xarxa emprant sòcols i analitzant l'escenari d'execució.

- Identificar escenaris que necessiten establir comunicació en xarxa entre diverses aplicacions.
- Identificar els rols de client i de servidor i les seves funcions associades.
- Reconèixer llibreries i mecanismes del llenguatge de programació que permetin programar aplicacions en xarxa.
- Analitzar el concepte de sòcol, els seus tipus i característiques.
- Utilitzar sòcols per programar una aplicació client que es comuniqui amb un servidor.
- Desenvolupar una aplicació servidor en xarxa i verificar el seu funcionament.

2. Desenvolupar aplicacions que ofereixen serveis en xarxa, utilitzant llibreries de classes i aplicant criteris d'eficiència i disponibilitat.

- Analitzar llibreries que permetin implementar protocols estàndard de comunicació en xarxa.
- Programar clients de protocols estàndard de comunicacions i verificar el seu funcionament.
- Desenvolupar i provar serveis de comunicació en xarxa.
- Analitzar els requeriments necessaris per crear serveis capaços de gestionar diversos clients concurrents.
- Incorporar mecanismes per possibilitar la comunicació simultània de diversos clients amb el servei.
- Verificar la disponibilitat del servei.
- Depurar i documentar les aplicacions desenvolupades.

1. Programació de comunicacions en xarxa

Actualment és impossible concebre cap dispositiu de processament de la informació (sigui ordinador, mòbil, llibre electrònic...) sense capacitat per comunicar-se. La gran majoria d'aplicacions actuals necessiten una connexió per instal·lar-se o per actualitzar-se i una bona part d'elles la necessiten també per a poder-se executar amb normalitat.

Sovint les aplicacions treballen amb recursos al núvol o extreuen les dades d'un SGBD situat en un servidor remot. Gràcies a la connectivitat dels nostres dispositius podem veure pel·lícules sense haver d'emmagatzemar-les en un disc local, podem sincronitzar els rellotges amb l'hora oficial, comprar entrades per anar al teatre sense moure'ns de casa o fer una partida en línia del nostre videojoc preferit. Tota aquesta capacitat de comunicació només és possible gràcies a les xarxes.

L'**objectiu** principal de les **xarxes** és el d'**interconnectar** diversos **dispositius** per tal de compartir totalment o parcialment els seus recursos.

Què són els protocols

Els protocols són especificacions que defineixen quin ha de ser el comportament de les diferents parts d'un sistema. En concret, els protocols de comunicació a través de la xarxa descriuen el paper de tots els elements de la xarxa (siguin físics o lògics) implicats en la intercomunicació.

Per aconseguir la interconnectivitat dels dispositius, les xarxes han anat definint elements estàndards físics (encaminadors, commutadors, concentradors...) o lògics (protocols i biblioteques de comunicació) per tal que les aplicacions puguin accedir als recursos remots fàcilment, reduint tant com sigui possible la complexitat de la implementació. Això ha permès crear aplicacions distribuïdes cada cop més transparents a l'usuari.

Entenem per **aplicació distribuïda** aquella que s'executa en més d'un dispositiu. Entenem per **aplicació distribuïda transparent** aquella aplicació distribuïda que no necessita un tractament especial pel fet d'estar distribuïda o, si més no, el redueix considerablement.

Des del punt de vista del programador el concepte de transparència s'aplica en referència a la menor quantitat de codi que caldrà escriure fent servir eines i biblioteques que automatitzin els processos bàsics de comunicació dins la xarxa.

L'evolució d'Internet ha permès també, la creació d'una única xarxa de comunicació en la qual tots els dispositius o xarxes inicialment aïllades (locals) adquireixen la capacitat d'interactuar. S'encunya així el terme de xarxa global. L'èxit de la xarxa global fa aparèixer nous elements estàndards que avancen en la línia de la programació distribuïda transparent.

1.1 Aspectes teòrics de la comunicació

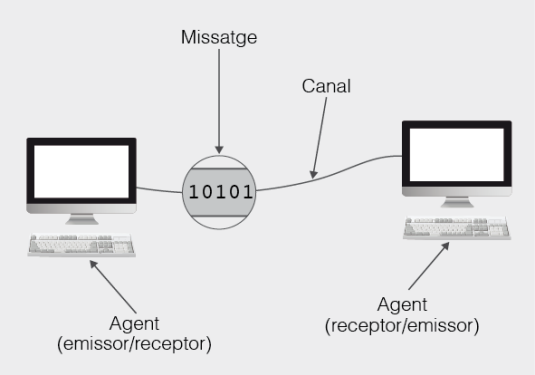
La comunicació és un procés complex en el qual es produeix una transferència d'informació entre agents independents. És important subratllar el terme *independents* perquè això implica que cada agent disposa del seu propi sistema d'informació que no comparteix de forma directa ni simple amb cap altre agent.

Per tal que la comunicació sigui possible és necessari que els agents comparteixin una mateixa manera de representar la informació tot i que aquesta no té perquè coincidir amb la representació interna que cada agent manté. Cal també que els agents tinguin òrgans o dispositius funcionals propis que els permetin d'una banda generar representacions comunes d'una part de la informació que mantenen i de l'altra interioritzar les representacions elaborades per altres agents. La part d'informació que es transmet s'anomena missatge.

El missatge es representa per mitjà de codis i símbols. Per això anomenarem codificació al procés de representar la informació d'una manera concreta. El missatge, codificat d'alguna manera comuna, arriba als agents a través del medi pel qual es transmet. Parlem de medis per referir-nos als mitjans que suporten la transmissió de la informació. L'agent que genera una representació i la traspassa al medi s'anomena emissor. L'agent que rep la informació a través d'un medi s'anomena receptor. Quan la informació elaborada per un agent emissor arriba a un agent receptor a través d'algun medi direm que entre ells s'ha establert un canal de comunicació per abstraure i emfatitzar el procés de traspàs.

El procés d'elaboració i el d'interpretació de la informació que duen a terme els agents emissors i receptors respectivament no són processos passius, requereixen de l'atenció dels agents en ambdós casos. Malgrat que depèn de les situacions, els agents implicats en la comunicació poden alternar els seus papers d'emissors i receptors donant lloc a un diàleg (vegeu la figura 1.1).

FIGURA 1.1. Esquema dels elements que intervenen en la comunicació



Els agents digitals, com ara els ordinadors o d'altres dispositius, necessiten que se'ls especifiqui una instrucció per escoltar els missatges que arribin del medi.

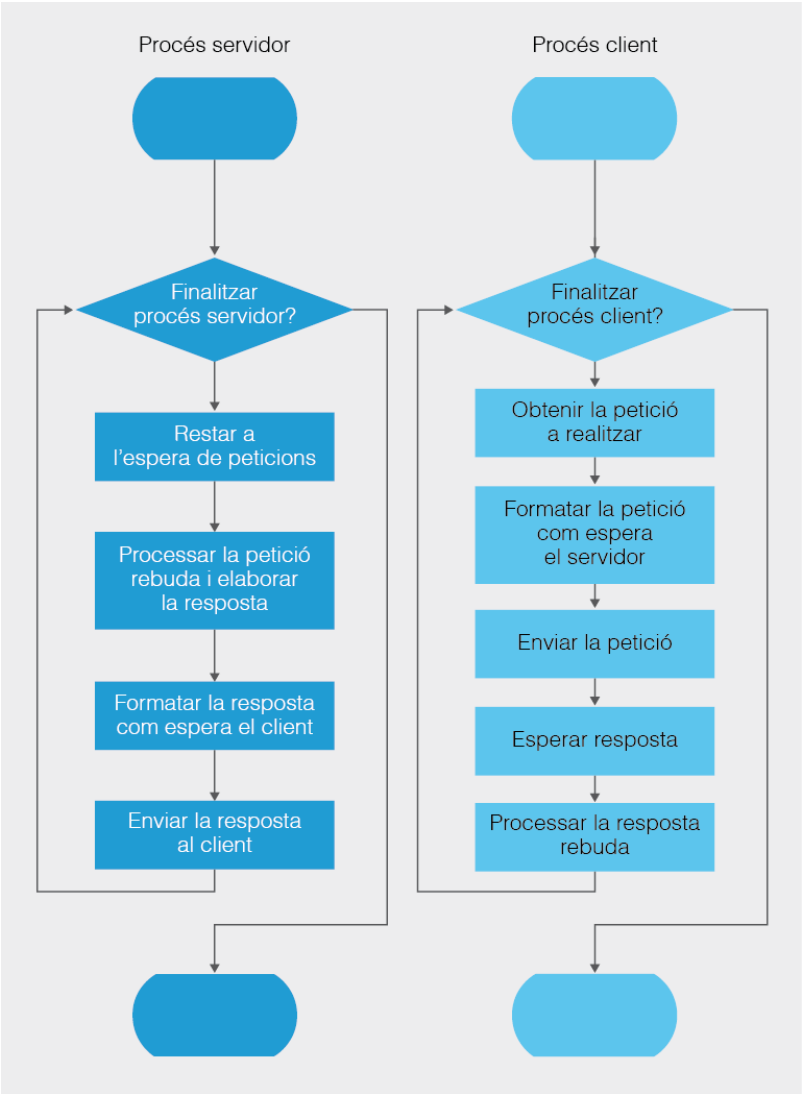
1.2 Rols client i servidor

La forma més clàssica de comunicar dispositius digitals és aplicant el model de client-servidor. El servidor és un dispositiu que conté informació a compartir amb altres agents anomenats clients, però no sap quan els clients necessitaran la seva informació. Per això el servidor haurà d'estar escoltant a l'espera que algun client li faci una petició, demanant quina part de la informació necessita. La petició és emesa per un client i codificada de tal forma que el servidor la pugui interpretar. Quan el missatge arriba al servidor, aquest detecta que la petició és per ell, la interpreta, genera la resposta adient i l'envia al client peticionari. Un cop resolta la demanda, el servidor quedarà de nou a l'espera de noves peticions.

Actualment moltes aplicacions segueixen models mixtes en què els dispositius poden fer de clients i servidors a la vegada, així permeten una major distribució de les dades i dels processos i aconsegueixen un abaratiment del maquinari. Tot i això, els papers de servidor i client no s'han perdut sinó que conviuen en un mateix dispositiu, sovint com a processos independents executats a l'hora.

No s'ha de confondre tampoc el rol del servidor amb el del receptor, ja que el servidor a més de rebre les peticions dels clients les ha de processar, obtenir-ne una resposta i enviar-la de tornada al client (vegeu la figura 1.2). El client per la seva banda, com podeu observar a la part dreta de la mateixa figura, ha de detectar quina petició necessita realitzar, ha de fer l'enviament de la petició, quedar-se a l'espera de la resposta i, quan arribi, processar-la adequadament. Noteu que aquest procés s'ha tancat dins d'un bucle continu per denotar que el client podria estar fent peticions de forma cíclica, però no ha de ser sempre així. Podria donar-se el cas d'un procés client que s'executi una única vegada. Per això a l'esquema les fletxes del bucle i el símbol condicional s'han difuminat indicant la seva opcionalitat.

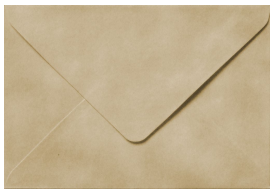
FIGURA 1.2. Esquema que representa les tasques genèriques de la majoria de processos servidor i processos client



A la part esquerra podeu observar l'esquema del procés servidor i a la dreta el del client. En aquest darrer s'ha pintat en un to més difuminat el símbol condicional i les fletxes del bucle per denotar que es tracta d'una possibilitat opcional i que hi poden haver processos client que no contemplin el bucle.

1.3 Bases de la comunicació entre aplicacions

Segurament, una de les claus que ha propiciat el creixement espectacular de l'ús de les xarxes durant el darrer quart del segle XX ha estat la senzillesa sobre la qual la commutació de paquets suporta la seva connectivitat. Aquesta tecnologia presenta certa analogia amb el sistema de lliurament de cartes postals. Les cartes s'envien dins de sobres que són els que duen la informació per fer-la arribar al seu destí.



Font: Parée

A les xarxes informàtiques, la informació que cal transmetre s'embolcalla dins de "paquets virtuals" en els quals s'hi referencien totes les dades necessàries per tal que la informació arribi al seu destí sense perdre's (dispositiu origen, dispositiu destí, aplicació a la qual va dirigida, servei demandat, etc.).

Per tal que la informació es pugui compartir a través d'una xarxa global cal que tots els dispositius i elements de la xarxa facin servir el mateix sistema d'empaquetatge. Internet fa servir un sistema molt senzill anomenat protocol IP. La gran senzillesa d'aquest sistema és el que ha permès la seva proliferació tan ràpida, ja que els costos d'implementació dels nodes intermedis de la xarxa són considerablement reduïts.

El protocol IP és tan simple que no contempla cap procés de control per verificar que la informació arriba realment al seu destí de forma complerta. És per això que la majoria de serveis i aplicacions d'Internet afegeixen un protocol extra de control anomenat TCP. Ambdós protocols es complementen tan bé que han passat a constituir el nucli central de la connectivitat d'Internet sota el nom de TCP/IP.

1.3.1 Arquitectura d'Internet: una estructura de capes

Els **processos d'una comunicació** són aquells que permeten als dispositius generar, transmetre i rebre un conjunt de senyals seqüenciats d'acord amb una convenció comuna de manera que el receptor sigui capaç d'interpretar allò que l'emissor vol comunicar.

Per a la generació, la transmissió i la recepció dels senyals es fan servir elements físics com ara les targes de xarxes o els cables. Aquests elements constitueixen la capa més baixa de la comunicació i solen anomenar-se capa física. El control dels elements físics recau sobre la capa immediatament superior, anomenada capa d'enllaç perquè també gestiona la transmissió d'informació entre dos nodes. La capa d'enllaç inclou elements lògics com ara els controladors de les targes de xarxa i també protocols d'accés, de transmissió i de recepció dels senyals. Ethernet/CSMA/CD és el protocol més estès, malgrat que podem trobar-ne d'altres.

Les dues capes esmentades, la física i la d'enllaç, aconsegueixen aglutinar bona part dels processos rutinaris de la comunicació i eliminar tota la complexitat atribuïble a la generació, recepció i transmissió dels senyals. Això allibera a les capes superiors d'una tasca de certa complexitat i permet així, simplificar la seva implementació.

Per sobre de la capa d'enllaç trobem la capa que gestiona l'adreçament entre nodes distants. Aquesta capa s'anomena també capa de xarxa i el protocol usat a Internet per gestionar l'adreçament és el protocol IP (Internet Protocol). En aquest nivell els diferents nodes de la xarxa global són capaços de decidir per quin camí cal enviar els paquets rebuts per tal que arribin al seu destí. Aquestes tres capes (física, enllaç i xarxa) han d'estar disponibles en qualsevol dels nodes de la xarxa global per tal de gestionar la transmissió d'informació entre dispositius.

El protocol IP s'encarrega només de l'adreçament dels paquets transmesos, però no pas del control de la transmissió. En una xarxa global com ara Internet, amb

milions de nodes interconnectats, les incidències durant les transmissions solen ser força habituals, ja sigui per interpretació errònia del senyal, per congestió, per caiguda o per manteniment d'algun node. Les incidències poden provocar l'arribada de dades errònies, retards o, fins i tot, la pèrdua d'alguns paquets; tot de conseqüències que poden acabar dificultant i, en darrer terme, impedint la comunicació real entre els dispositius. És per això que cal una capa més per controlar la qualitat de les transmissions. Aquesta capa s'anomena capa de transport.

La capa de transport s'implementa només sobre nodes terminals (dispositius origen i destí), mai sobre els nodes intermedis de la xarxa. Aquesta característica presenta certs avantatges i desavantatges. El principal avantatge és el fet d'abaratir els dispositius intermedis, ja que la implementació d'un protocol de control presenta un grau de dificultat realment molt més elevat que la simple implementació del protocol IP. També presenta l'avantatge de poder usar diversos protocols de control, ja que només hi estan implicats els nodes finals. És a dir, només cal posar d'acord dos nodes abans d'iniciar la transmissió. De fet, a Internet solen fer-se servir principalment dos protocols de control, el protocol UDP i el protocol TCP.

TCP és l'acrònim de Transmission Control Protocol i UDP d'User Datagram Protocol.

UDP és un protocol de control força senzill, només controla la integritat dels paquets enviats. Això vol dir que l'utilitzen aquelles aplicacions que no necessiten mantenir cap ordre en l'arribada de la informació i en la qual ni la pèrdua d'alguna dada impedeix el normal funcionament. El protocol TCP, per contra, és un protocol molt més sofisticat que gestiona el control de l'ordre i també la pèrdua de paquets a més de la integritat de la informació.

El principal desavantatge de relegar el control als nodes finals és que les errades no es detecten fins que la informació ha creuat tota la xarxa i ha arribat al node destí. Si es detecta alguna distorsió és necessari avisar a l'origen que torni a enviar de nou les dades. Com es pot imaginar, això acaba traduint-se en un augment de tràfic per la xarxa.

Malgrat que per sobre de la capa de transport poden anar-se sobreposant tantes capes com sigui necessari, aquestes depenen molt de l'aplicació executada. Podem trobar aplicacions que requereixin un canvi de format de les dades abans de ser enviades o en rebre-les o bé que necessitin afegir dades específiques pròpies del sistema o d'una configuració específica. No es pot, però, considerar un tractament universal. És per això que sol simplificar-se l'esquematització amb una única capa anomenada capa d'aplicació.

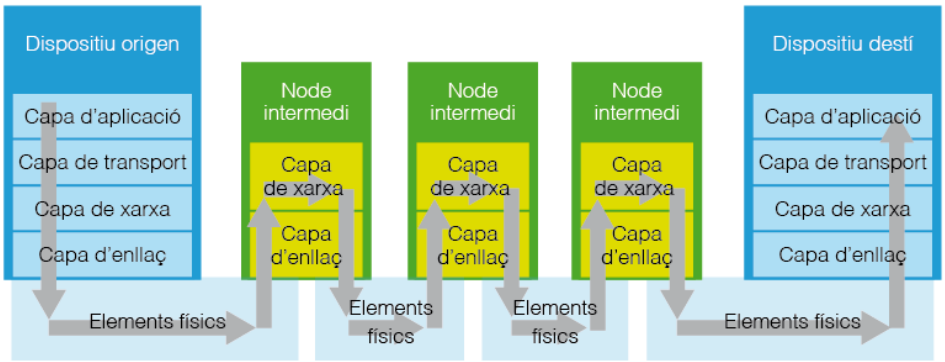
La capa d'aplicació conté la funcionalitat pròpia de l'aplicació específica que cal executar i és la capa en la qual comença i acaba tot el procés de comunicació. Quan des d'un dispositiu terminal, una aplicació necessita enviar informació a un altre dispositiu del qual suposarem que es coneix la seva adreça, la capa d'aplicació donarà el format adequat a les dades que s'han d'enviar i les passarà, juntament amb l'adreça en la qual cal fer l'enviament, a la capa de transport. Aquesta capa afegirà dades que facilitaran el control de la transmissió i deixaran les dades en mans de la capa de xarxa, en la qual s'hi escriuran les adreces d'origen i destí (la darrera, rebuda des de la capa d'aplicació a través de la capa de transport). La capa d'enllaç afegirà noves dades per aconseguir que tota aquesta informació

arribi al node més proper situat en el camí que porta al dispositiu destí i activarà els elements físics per tal que generin el senyal corresponent a tota la informació que s'ha anat generant (vegeu figura 1.3).

Un cop el senyal es troba en el medi físic, el node intermedi al qual s'ha enviat el senyal escoltarà i interpretarà la informació des de la seva capa d'enllaç. La informació es traslladarà a la capa de xarxa per tal de decidir quin serà el següent node al qual cal fer l'enviament, per tal d'aconseguir arribar al dispositiu destí. Un cop decidit, es tornarà a la capa d'enllaç des de la qual es generarà de nou el senyal adreçat al següent node intermedi. Aquest procés es repetirà tantes vegades com sigui necessari fins arribar al dispositiu destí (vegeu figura 1.3).

Quan el senyal arriba al dispositiu destinatari, es detecta com a pròpia la informació llegida comprovant l'adreça IP del destí i, en comptes de fer un nou reenviament des de la capa de xarxa, es trasllada a la capa de transport. Aquesta capa verificarà que la informació ha arribat correctament i només en cas afirmatiu pujarà les dades rebudes a la capa d'aplicació que les interpretarà i actuarà en conseqüència (vegeu figura 1.3). Si la capa de transport detecta alguna errada o distorsió en les dades rebudes, caldrà demanar a l'origen que torni a enviar les dades.

Figura 1.3. Esquema de l'arquitectura de capes d'Internet



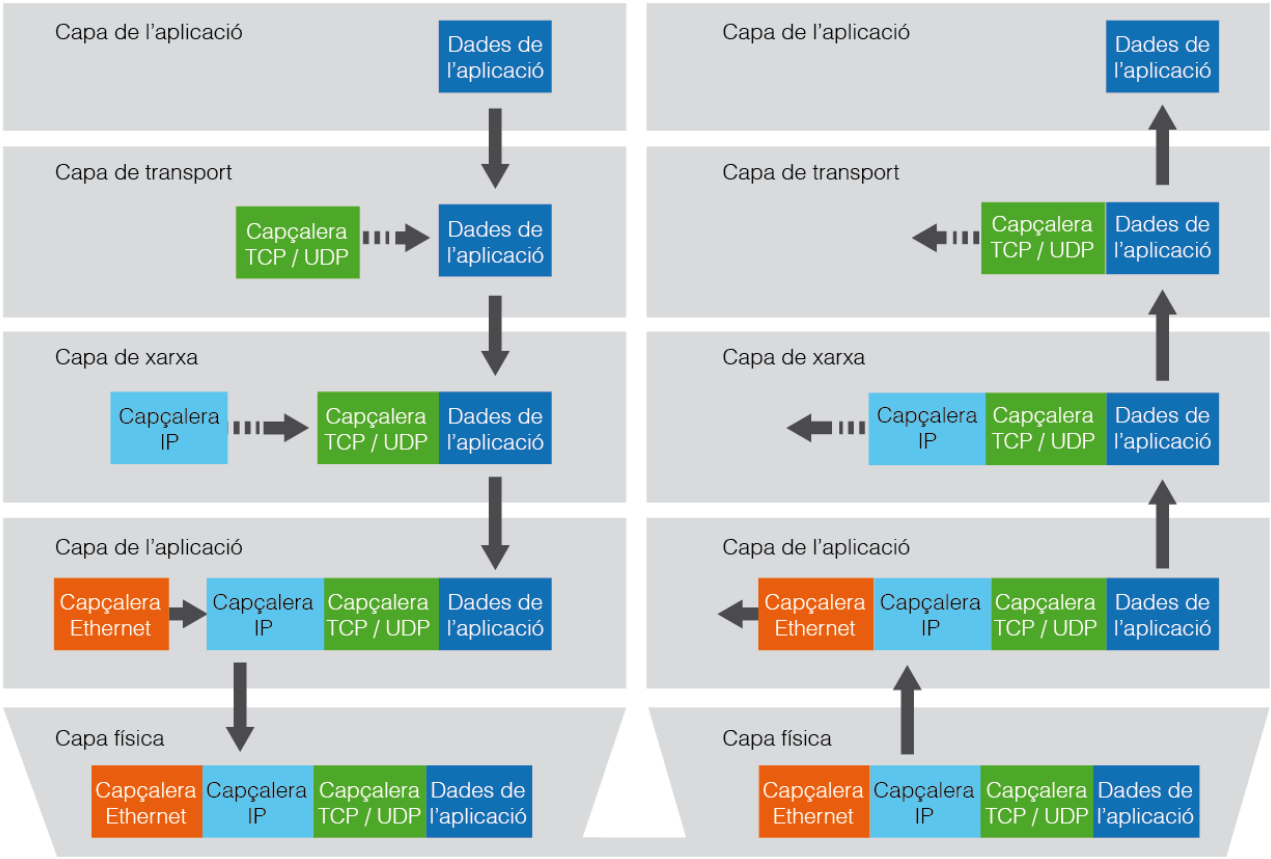
En l'esquema de la figura 1.3 es poden observar les diferents capes en què s'estructura la comunicació en una xarxa global com Internet. També es pot apreciar el flux d'informació a través de les capes des del dispositiu origen fins arribar al destí, passant per diversos nodes intermedis. Observeu com els nodes intermedis no implementen les capes superiors a la de xarxa.

1.3.2 Capçaleres i dades

Es pot veure ràpidament que la quantitat d'informació enviada pot acabar superant àmpliament la quantitat de dades originals, ja que cada capa afegeix informació extra que serà útil per la gestió. El procés invers en el qual la informació rebuda ascendeix des de les capes baixes pateix una transformació similar però a l'inrevés.

És a dir, a cada pas s'elimina la part d'informació específica del nivell en qüestió i només es passa a la capa superior la informació que li és pròpia. A la figura 1.4 podeu veure el detall d'ambdós processos (d'enviament i recepció). Les dades extres es localitzen en una capçalera per tal que sigui fàcil d'afegir, d'eliminar i de llegir.

FIGURA 1.4. Esquema del traspàs d'informació entre capes



En l'esquema de la figura 1.4 podeu observar com, per aconseguir enviar certa informació (les dades de l'aplicació), cada capa ha d'anar afegint les seves dades extres en una capçalera de mida fixa. També podeu observar el procés contrari. En rebre les dades, cada capa elimina la seva capçalera per tal de passar a la capa superior les dades corresponents netes.

Fixeu-vos que cada capa només tracta els bytes de la seva capçalera, situats sempre a l'origen de les dades. Coneixent la mida de la capçalera es pot aconseguir un tractament raonablement eficient. És important que el tractament no sigui excessivament costós, ja que, com es pot suposar, qualsevol enviament repetirà els processos de recepció/enviament a cada node pel qual passi. Sortosament, els nodes intermedis no sobrepassen mai la capa de xarxa, cosa que alleuja força el procés de transmissió. Recordeu que la capa de transport només es fa efectiva en els dispositius finals.

1.3.3 Contingut de la informació a cada capa

Sovint la capa d'aplicació formata d'una manera concreta les dades a enviar i segueix un protocol establert per tal de fer factible la comunicació entre els dispositius implicats. Es tracta que l'emissor posi en circulació la informació de tal manera que els receptors puguin interpretar les dades que els arriben. Malgrat que en aquest nivell existeixen una gran quantitat de protocols estàndard com ara HTTP, FTP, TELNET i molts d'altres, n'hi ha prou amb fer servir una convenció comuna entre dispositius finals. L'ús de protocols estàndards, però, pot arribar a facilitar molt la implementació de les aplicacions, ja que la majoria de llenguatges disposen de biblioteques i utilitats d'alt nivell específiques per aquests protocols.

Capa de transport

Els principals protocols usats en aquest nivell són UDP i TCP. Abans de poder veure els camps que composen les dades *extres* d'aquesta capa, cal analitzar quines són les seves funcions. Es tracta de la capa situada immediatament per sota de les aplicacions. Durant la recepció de les dades, necessitarà reconèixer quina és l'aplicació destinatària de la informació ja que, per descomptat, podria donar-se el cas de tenir diverses aplicacions independents esperant dades. Per convenció, la capa de transport farà servir un número per identificar les diverses aplicacions que es trobin escoltant. Habitualment es coneix aquest número identificatiu amb el nom de *port*. A cada enviament, caldrà incloure sempre el valor del port com a dada extra de la capa. Així el receptor podrà esbrinar quina és l'aplicació destinatària.

S'utilitzen 2 bytes per identificar el port. Això significa que hi ha 65.535 ports disponibles en cada dispositiu. Potencialment existeix, doncs, la possibilitat d'executar de forma concurrent la mateixa quantitat d'aplicacions en un únic dispositiu. A la pràctica el nombre de ports útils es redueix força, però continua sent més que suficient.

La reducció del nombre de ports és un conseqüència de la següent convenció:

- Els valors de ports que oscil·len entre 1 i 1.023 es reserven per aplicacions específiques i estàndards. El port que usen els servidors de pàgines web és un port 80, per exemple. De fet cada un dels serveis estàndards d'Internet (correu electrònic, FTP, DNS, etc.) tenen associats ports únics i diferents. Per això en una mateixa màquina podem instal·lar-hi tots els serveis disponibles i fer peticions des de qualsevol client sense necessitat que ens informin prèviament del port que estan escoltant.
- Els port que es troben entre el valor 1.024 i 49.151 cal destinar-los a serveis no estàndards. És a dir, a servidors que es trobin permanentment escoltant les peticions que els arribin. Per tal que el client pugui connectar-s'hi haurà de saber prèviament el número del port que el servei utilitza. Les aplicacions corporatives, com ara els sistemes gestors de bases de dades, són un clar

exemple d'aquests serveis. A les empreses no els interessa fer públic el port d'accés a les dades. Només als clients de la corporació els cal saber el port al qual han d'adreçar les seves peticions.

- Finalment els ports que es troben entre els valors 49.152 i el 65.535 es destinen a comunicacions temporals dels clients i seran assignats directament per la capa de transport en el moment d'iniciar una comunicació.

Protocol UDP

Algoritmes de suma

Un algoritme de suma és aquell que obté un resultat operant sobre una conjunt de bytes, de manera que el valor obtingut no sobrepassa mai un valor màxim i que la probabilitat d'obtenir valors idèntics a partir de conjunts de bytes diferents és el més petita possible.

Trobareu informació sobre els algoritmes de suma a l'apartat "Tecnologies de seguretat i encriptació" de la unitat *Seguretat i criptografia*.

A més a més dels ports origen i destí, la resta de dades extres d'aquest nivell vindran determinades pel protocol utilitzat. UDP és més senzill que TCP perquè únicament comprova la coherència de les dades rebudes. UDP utilitza la longitud de les dades i un valor de comprovació anomenat `checksum` per a realitzar la verificació de la coherència. El valor `checksum` s'obté a través d'un algoritme de suma que porta el mateix nom.

L'algoritme `checksum` consisteix en sumar un conjunt de bytes agrupats en blocs de 16 bits (2 bytes). En tractar-se d'una suma, el resultat que obtindríem en qualsevol dels possibles casos podria representar-se sempre amb un màxim de 17 bits. En aquells casos que el resultat de la suma necessiti 17 bits, s'elimina el bit més significatiu (el de més a l'esquerra) per tal d'ajustar sempre la mida a 16 bits i s'incrementa el valor en una unitat.

Exemple de càlcul del valor checksum d'un conjunt de byte

Tenim el següent conjunt de 8 bytes: 138,40,37,218,117,162,42,21 Quan el traslladem a binari obtenim:

```
10001010 00101000 00100101 11011010 01110101 10100010 00101010
    00010101
```

Agrupem el conjunt en blocs de 16 bits:

```
       valor binari    valor decimal
   10001010 00101000     35368
+  00100101 11011010      9690
+  01110101 10100010     30114
+  00101010 00010101     10773
 ------------------------------------
 1 01001111 10111001     85945
```

Com que el resultat de la suma hauria d'ocupar 17 bits, mantindrem els 16 bits menys significatius i incrementarem el resultat una unitat:

```
      valor binari    valor decimal
   0100111110111001     20409
+                 1         1
 ------------------------------------------------
   0100111110111010     20410
```

El resultat final després d'aplicar l'algoritme `checksum` són 2 bytes de valor:

```
01001111 10111010
```

Si usem una expressió decimal, aquest es correspondria amb: 79,186

Aquestes quatre dades que UDP afegirà, se seqüencien, tal com es mostra a la figura 1.5. Cada una d'elles ocupa 16 bits i se situen per davant de les dades a mode de capçalera.

Figura 1.5. Esquema de les dades afegides pel protocol UDP

Port origen	Port destí
Longitud total (dades + capçalera)	Checksum de dades + capçalera
Dades	

Durant la rebuda de la informació el protocol UDP extreu les dades de la capçalera i comprova si la longitud i el valor `checksum` coincideixen amb les dades rebudes. En cas afirmatiu, traspassa les dades a l'aplicació corresponent d'acord amb el port indicat com a destí. En cas que la longitud o el `checksum` siguin incoherents es descarten les dades com si mai haguessin arribat.

Tal com es pot deduir, UDP ofereix molt poques garanties de qualitat. Només garanteix que en cas de lliurar alguna dada, aquesta serà consistent. Tot i això, es tracta d'un protocol molt ràpid que necessita veritablement molt pocs recursos per mantenir una gran quantitat de comunicacions ràpides i amb múltiples clients. Per això es troba àmpliament implementat en aplicacions concorregudes que necessiten fer prevaler la velocitat per sobre de la qualitat, com ara els servidors DNS o els serveis de veu IP o l'*streaming* de vídeo.

Protocol TCP

Les aplicacions que necessitin fiabilitat en les transmissions hauran de fer servir el protocol TCP. Es tracta d'un protocol de gran complexitat amb el qual es garanteix que totes les dades enviades des de l'origen, arriben al destí íntegrament i en el mateix ordre en què han sortit de l'emissor.

Per tal d'assegurar la fiabilitat de la transmissió, el protocol TCP manté un diàleg permanent entre emissor i receptor, en el qual ambdós dispositius s'informen d'allò que van enviant i rebent. El diàleg s'inicia amb una petició de connexió i es manté obert (ambdós dispositius s'escolten mútuament) fins que un dels dos envïi un senyal per finalitzar la connexió.

> Es diu que el **protocol TCP** està **orientat a la connexió** perquè manté un diàleg permanent entre l'emissor i el receptor. En canvi el **protocol UDP no està orientat a la connexió** perquè un cop enviades les dades l'emissor es desentén de la transmissió.

L'explicació en tot detall del protocol TCP sobrepassa l'objectiu d'aquest mòdul. Sovint disposarem de biblioteques ja implementades que ens permetran gaudir d'aquesta tecnologia sense necessitat d'haver-les d'implementar. Malgrat tot, és important conèixer quines són les seves principals característiques per entendre el comportament d'aquestes biblioteques i per decidir quin protocol serà l'adequat per les aplicacions que ens toqui desenvolupar.

Característiques de la comunicació entre dos dispositius usant el protocol TCP

TCP és un protocol pensat per comunicar només dos interlocutors. Les aplicacions que necessitin transmetre a múltiples dispositius hauran d'escollir entre aplicar el protocol UDP, amb el risc de pèrdua d'informació que això suposa, o bé implementar múltiples connexions dos a dos amb les quals poder controlar a través de TCP la transmissió.

Aplicant el protocol TCP, abans d'iniciar qualsevol transmissió de dades caldrà establir una connexió entre dues màquines. Per aconseguir la connexió caldrà establir un petit diàleg entre ambdós dispositius en el qual un d'ells enviarà un senyal específic anomenat `petició`. L'inici de la connexió implicarà la reserva d'un port en cada dispositiu destinat exclusivament al diàleg i a la transmissió de les dades entre els dos interlocutors implicats. És per això que el protocol exigeix que ambdós dispositius s'enviïn mútuament un senyal d'acceptació de la connexió indicant que el port està reservat i que la comunicació pot començar. El port es mantindrà reservat fins que un dels interlocutors decideixi abandonar la comunicació iniciant un diàleg per finalitzar la connexió.

Diem que TCP és un protocol `full-duplex` perquè estableix dos canals de comunicació en cada connexió, cada un en un sentit diferent. És a dir, es manté un canal de transmissió des del dispositiu `A` al dispositiu `B` independentment del canal de transmissió que vagi del dispositiu `B` al dispositiu `A`. A la pràctica això implica que cada dispositiu manté un registre del qual envia i rep de forma independent.

Diem també que els canals de la connexió estan orientats a fluxos (`streaming-oriented`). És a dir, el registre mantingut per cada dispositiu de les dades enviades i rebudes serveix, entre d'altres coses, per determinar l'ordre en què han sortit les dades i poder organitzar les dades rebudes en el mateix ordre abans de traspassar-les a la capa d'aplicació. Sovint les limitacions de la xarxa impedeixen enviar les dades en un únic bloc. En aquests casos el protocol TCP permet dividir els enviaments en segments més petits en els quals cada un d'ells contindrà parcialment les dades a enviar. Si els dispositius es troben a certa distància, cada segment podria agafar camins diferents i arribar a destí de forma desordenada. El protocol TCP envia, juntament amb les dades, un valor *extra* que permetrà ordenar les dades de cada segment.

El valor que permet l'ordenació és un comptador dels bytes enviats de manera que el receptor pugui reconstruir el flux de dades fàcilment malgrat l'ordre d'arribada dels segments. Això implica que el receptor no pot traspassar les dades dels segments a l'aplicació conforme vagin arribant, sinó que haurà de mantenir un *buffer* en el qual reconstruirà totes les dades abans de fer el traspàs a la capa superior.

Per tal d'assegurar la fiabilitat de les dades, el protocol estableix que per cada segment enviat des d'un o altre dispositiu, el receptor contestarà amb un senyal especial de **reconeixement** que indiqui a l'emissor que les dades del segment han arribat amb èxit. El senyal de reconeixement inclou un valor que identifica la porció de bytes rebuda en segments fent servir el valor del comptador de bytes.

D'aquesta manera emissor i receptor poden anar interpretant si la transmissió va tenint èxit, o si cal fer un nou reenviament d'algun segment, ja sigui perquè no s'ha rebut el seu senyal de reconeixement, cosa que faria suposar la pèrdua parcial d'alguns segments o com a conseqüència d'haver rebut un reconeixement amb un identificador repetit, cosa que suposaria l'arribada corrupte de dades.

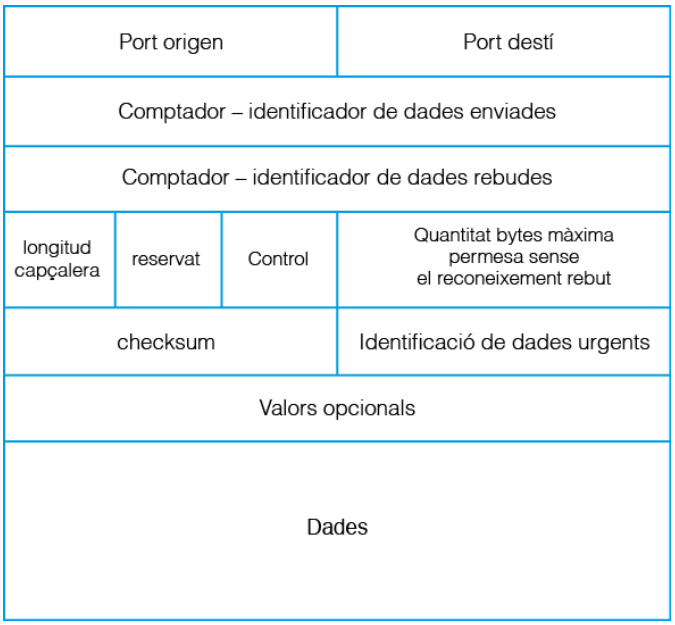
En distàncies llargues, el temps comptabilitzat des del moment de l'enviament d'un segment fins a rebre el seu reconeixement, pot arribar a ser força elevat. Si l'emissor hagués d'esperar l'arribada dels reconeixements abans d'enviar un nou segment, la transmissió podria fer-se impracticable a causa de la lentitud. Per tal d'evitar-ho es permet fer enviaments de diversos segments sense haver rebut la corresponent confirmació.

Aquest biaix entre les dades enviades i les dades confirmades obliga a disposar d'algoritmes que dedueixin en cada moment l'estat de la transmissió i puguin actuar en conseqüència per tal de fer reenviaments quan calgui, demanar una nova sincronització de les dades a transmetre a partir d'un determinat segment o fins i tot canviar la freqüència d'enviament dels segments per tal que el nombre de segments sense confirmar no augmenti perillosament.

Les dades que el protocol TCP estableix per aconseguir les característiques descrites les podeu observar a la figura 1.6. Com podeu veure, la quantitat de dades necessàries aquí és força superior a les usades en el protocol UDP. A banda dels ports origen i destí o del `checksum` de verificació de la integritat del segment, la resta de dades són específiques de TCP. Destaquem els identificadors dels bytes enviats o rebuts que també actuen com a comptadors ja que es van incrementant en funció dels bytes enviats o rebuts correctament. Destaquem també que en aquest protocol només es transmet la longitud de la capçalera ja que la longitud de les dades es pot deduir usant l'identificador-comptador de les dades enviades.

El camp de control permet donar informació extra de les dades de la capçalera o de l'estat de la transmissió. Per exemple permet indicar que certes dades són importants i cal processar-les i enviar-les a l'aplicació abans que la resta o demanar un nova sincronització de les dades i forçar un enviament a l'aplicació de les dades arribades fins el moment, etc.

FIGURA 1.6. Esquema de les dades afegides pel protocol TCP



Capa de xarxa

IP és l'acrònim d' `Internet Protocol`

Topologia de xarxa

La topologia d'una xarxa fa referència a la interconnexió directa que cada node té amb la resta. Diem que dos nodes es troben connectats directament quan poden enviar-se informació sense passar per cap altre node. La topologia influencia el trànsit que la xarxa és capaç de suportar.

La capa de xarxa fa servir el protocol IP per aconseguir connectar dos dispositius de la xarxa amb independència de la distància que hi hagi entre ells. Per aconseguir la connexió caldrà esbrinar quins nodes intermedis condueixen al dispositiu destí. En cas que hi hagi més d'un camí per arribar-hi, caldrà escollir-ne un. La decisió no és trivial ni absoluta perquè una xarxa de comunicacions no és quelcom immutable. La capacitat de connexió de la xarxes va variant a cada instant depenent d'aspectes tan volàtils com ara la quantitat de trànsit global que hi circuli o les decisions que altres nodes hagin pres amb anterioritat. Per descomptat depenen també d'aspectes estàtics com ara la topologia o l'eficiència dels recursos físics.

En una xarxa com Internet el camí no es planifica des de l'origen sinó que cada node pel qual passa el missatge a transmetre pren la seva decisió pròpia. La capa de xarxa dels nodes intermedis intenta esbrinar cap on encaminar el missatge de manera que arribi el més aviat possible al destí. En la presa de decisió caldrà tenir en compte aspectes com ara quin pot ser el camí més curt, quin camí es troba menys congestionat o quin retard porta acumulat el missatge.

Adreçament IP

Les **adreces IP** identifiquen les connexions directes d'un dispositiu de manera única. Un dispositiu pot disposar d'una o diverses connexions a través de les quals enviar i rebre els missatges. En tractar-se d'un valor únic a tota la xarxa s'utilitzen també per identificar els dispositius, malgrat que en realitat un dispositiu tindrà sempre tants identificadors com connexions tingui.

Generalment les connexions directes es coneixen com a interfícies de xarxa i s'identifiquen amb les targes de xarxa. Malgrat tot, a vegades s'opta per fer emulacions a través del programari. És el que es coneix com a interfícies de xarxa lògiques. Per exemple els sistemes operatius (SO) disposen de l'adreça 127.0.0.1 per identificar el dispositiu propi en el qual l'SO s'executi. Aquesta adreça IP s'anomena també adreça *loopback* perquè mai surt del dispositiu cap a la xarxa sinó que es retorna cap a la capa superior com si acabés d'arribar dirigida al propi dispositiu.

Col·laboració entre nodes

Els nodes col·laboren entre ells creant petits mapes a la xarxa per tal de poder decidir cap on adreçar els paquets que arriben des de diferents orígens a través de les seves interfícies de xarxa. A vegades els nodes demanen als seus veïns que s'identifiquin explicitant la seva adreça IP. Això els permet crear petites llistes associades a cada una de les interfícies. També és comú informar als veïns de les llistes d'adreces que un node és capaç de cobrir, indicant a més la llunyania a la qual es troben les diferents adreces IP de la llista. Per últim, el nodes disposen també de certes regles d'adreçament prefixades que ajuden a acabar de prendre les decisions d'adreçament dels paquets.

Una regla prefixada podria ser, per exemple, definir una interfície de xarxa per defecte, de manera que qualsevol adreçament que no figuri a cap llista es desviarà sempre per la mateixa sortida. Això té força sentit, ja que alguns nodes d'Internet són el que s'anomena *súper-hubs*, és a dir, nodes amb moltes connexions, de manera que la probabilitat que aconsegueixin resoldre l'adreçament és molt més gran que la majoria de les alternatives restants.

També és habitual definir regles que adrecin els paquets d'acord amb els bytes més significatius de les seves adreces. D'aquesta manera s'aconsegueix definir rangs d'adreces associats a una o altra interfície de xarxa. Efectivament, sovint les adreces IP que comencen per un valor es troben concentrades físicament en una regió propera o pengen d'un mateix node capaç d'adreçar-les. Les operacions necessàries per determinar si una adreça IP es troba o no dins d'una rang determinat són tan senzilles i ràpides que els dispositius són capaços d'avaluar un gran nombre de regles de forma molt eficient.

Totes aquestes accions descrites tenen com a objectiu segmentar la xarxa i crear mapes específics per a cada node per tal de poder establir un conjunt de regles senzilles que determinin quina interfície de sortida serà la millor opció per transmetre un paquet cap al seu destí.

Adreces i màscares

Les adreces IP són en el fons una seqüència de bits sobre la qual es poden fer operacions binàries. Aquesta és una característica molt útil perquè ens permet definir rangs d'adreces i determinar d'una forma molt ràpida si una adreça qualsevol es troba dins del rang. Per definir el rang utilitzarem una màscara i una adreça de xarxa base.

Una màscara és una seqüència de bits de la mateixa mida que una adreça IP, però organitzada de manera peculiar. Hi ha un nombre indeterminat de bits més significatius (els que es troben més a l'esquerra de la seqüència binària) que sempre prenen el valor 1, mentre que la resta de bits, els menys significatius (que localitzem a la dreta del conjunt d'uns) prenen sempre el valor 0.

Exemples de màscares

Podeu apreciar perfectament el conjunt d'uns que ocupen els bits més significatius i el conjunt de zeros que ocupen el bits menys significatius.

- `255.255.0.0` la representació binària del qual és `11111111 11111111 00000000 00000000`
- `255.255.192.0` la representació binària del qual és `11111111 11111111 11000000 00000000`
- `255.255.255.240` la representació binària del qual és `11111111 11111111 11111111 11110000`

Una adreça IP base és aquella que, combinada amb una màscara, permet definir un rang d'adreces IP de forma jeràrquica. És a dir, el rang de totes les adreces que comencin pel valor de l'adreça base comparant el nombre de bits de valor 1 de la màscara.

Definició de rangs a partir d'una adreça IP i d'una màscara

- El rang definit per `128.154.X.X` es pot definir fent servir una adreça base com per exemple 128.154.0.0 i una màscara 255.255.0.0
- De forma semblant, el rang definit per `144.134.12.X` es pot definir fent servir una dreça base com per exemple 144.134.12.11 i una màscara 255.255.255.0

Fixeu-vos que per definir el valor de l'adreça base només cal tenir en compte els bits significatius de la màscara, la resta podrien prendre qualsevol valor.

Per detectar si una adreça pertany o no a un rang determinat per una adreça base i una màscara, només caldrà realitzar la operació binaria `&` i comparar el resultat. És a dir, direm que una adreça IP qualsevol (`add1`) pertany al rang definit per la màscara (`masc`) i l'adreça base (`addb`) si `addb&masc=add1&masc`.

Operació & per determinar si una adreça pertany al rang format per una adreça base i una màscara

Direm que l'adreça `10000000 11010010 10011001 10101001` (en decimal `128.210.153.169`) pertany al rang d'adreces definit per l'adreça base `10000000 11010010 10011111 00000001` (en decimal `128.210.159.1`) i la màscara `11111111 11111111 11111000 00000000` (en decimal `255.255.248.0`) perquè el resultat d'operar la màscara amb ambdues adreces IP és idèntic:

```
1   10000000 11010010 10011111 10101001        10000000 11010010
        10011001 10101001
2  &11111111 11111111 11111000 00000000       &11111111 11111111
       11111000 00000000
```

```
3 ____________________________________________
    ____________________________________________
4     10000000 11010010 10011000 00000000        10000000 11010010
          10011000 00000000
```

L'ús de màscares és un recurs molt útil per jerarquitzar la xarxa, però sobretot per definir els filtres que permeten als nodes intermedis de la xarxa decidir cap on caldrà enviar els paquet en trànsit.

Adreces IP especials

Si bé és cert que les adreces IP tenen per objectiu identificar les interfícies de xarxa dels dispositius, hi ha certes adreces específiques que en comptes d'identificar centren la seva raó de ser en la funcionalitat que se'ls dóna. Ja sabeu que l'adreça 127.0.0.1 serveix per redirigir qualsevol petició de xarxa cap al propi dispositiu que fa la petició. De fet tot el rang d'adreces que comencen per `127` es reserva com a adreces de bucle o *loopback*, tal com se els coneix més popularment.

De forma semblant hi ha rangs d'adreces reservades a les xarxes locals. Això significa que no són adreces úniques sinó que es poden reutilitzar en cada xarxa local. És normal que les xarxes local petites tinguin adreces IP compreses entre els valor `192.168.0.0` i el valor `192.168.255.255`, però també es reserven a l'ús privat el rang comprès entre `172.16.0.0 i 172.31.255.255` o el rang de totes les adreces que comencin pel valor 10, és a dir que tinguin la forma `10.x.x.x`.

El valor `255` decimal equival al valor `11111111` binari. És a dir, tots els bits que composen el byte prenen el valor 1.

Dins del conjunt d'adreces especials també existeixen les adreces de difusió, conegudes popularment com a adreces *broadcast*. Es tracta d'adreces que permeten fer l'enviament d'un únic paquet, dirigit a tot el conjunt de dispositius que comparteixin un cert rang d'adreces. Les adreces *broadcast* s'identifiquen per disposar d'una o més posicions dels bytes menys significatius amb el valor `255`.

Exemples d'adreces broadcast

L'adreça `248.147.118.255` serveix per adreçar-se a tots els dispositius que tinguin una adreça compresa entre `248.147.118.1` i `248.147.118.254`. També l'adreça 154.3.255.255 aconseguirà arribar a aquells dispositius que tinguin una IP dins el rang `154.3.x.x`.

Finalment farem una menció específica a les adreces anomenades de multidifusió o *multicast*. Es tracta d'adreces a les quals es poden fer subscripcions. Són adreces que no pertanyen a cap dispositiu concret, però que són controlades per certs dispositius intermedis que anomenem encaminadors o *routers*. Aquests dispositius poden rebre la petició de qualsevol dispositiu d'afegir la seva adreça IP a la multidifusió de qualsevol paquet dirigit a l'adreça *multicast*. És a dir, que el *router*, en detectar l'enviament d'informació dirigida a una adreça *broadcast* controlada per aquest dispositiu replicarà el paquets rebuts a totes les IP associades a l'adreça *multicast*.

L'objectiu d'aquest mecanisme és reduir el trànsit dins d'una xarxa. Hi ha serveis i aplicacions que necessiten enviar massivament una gran quantitat d'informació a molts dispositius a la vegada. Si s'utilitza un paquet diferent per a cada dispositiu es corre el risc d'inundar la xarxa amb paquets que contenen les mateixes dades i que només es diferencien per les adreces destí de la capçalera. El mecanisme

multicast permet utilitzar un únic paquet durant un tram de la transmissió per fer arribar la mateixa informació als mateixos dispositius. És a dir, en el tram comprès entre el servidor i el dispositiu controlador de l'adreça *multicast*, només es farà servir un únic *datagrama*.

El *multicast* se sol reservar a empreses de comunicació que requereixin un enviament selectiu però continu de dades, com per exemple els mitjans de comunicació de pagament o fins i tot corporacions educatives que ofereixin *streaming* d'àudio o vídeo. També pot utilitzar-se en jocs MMO (sigles en anglès que signifiquen *Massively Multilayer Online*), malgrat que encara hi ha pocs servidors que facin servir aquesta tècnica. Les adreces *multicast* també solen usar-se per automatitzar processos de manteniment dins d'una xarxa com ara còpies massives de dades, clonacions o enviament d'ordres a múltiples dispositius .

Totes les adreces *multicast* comencen pel valor binari `1110`. És a dir, configuren el rang que va des de `11100000` a `11101111`, que corresponen als valor decimals compresos entre `224` i `239`.

Les adreces IP reservades al *multicasting* són les que es troben compreses entre els valors `224.0.0.0` i `239.255.255.255`. Les adreces del rang `224.0.0.x` es troben reservades per a tasques de manteniment en àmbits de xarxes d'àrea local. Generalment tots els encaminadors interpretaran aquestes adreces com a *multicast*, acceptaran peticions de subscripció dels dispositius clients i faran el repartiment pertinent dels paquets que arribin a l'adreça *multicast* a tots el dispositiu subscrits. Així tots aquells encaminadors definits com a passarel·la o *gateway* d'un conjunt de dispositius seran els encarregats de fer la multidifusió del segment de xarxa que controlin.

Noms de domini

Ja veieu que les adreces IP no identifiquen realment un dispositiu sinó les seves interfícies de xarxa. De fet, no és necessari identificar els dispositius per fer una adreçament correcte. Ara bé, per a l'ús humà les adreces IP dificulten la localització de dispositius, ja que poden ser redundants, costen de memoritzar i en no tenir significat propicien els error involuntaris d'escriptura. Per tal d'apaivagar totes aquestes complicacions els dispositius solen batejar-se amb un nom alfanumèric.

Si l'àmbit del nom no supera les xarxes locals, no és gaire difícil controlar la seva unicitat dins el mateix àmbit ja que el nombre de noms necessaris sol ser relativament reduït i es pot coordinar des de la mateixa organització propietària de la xarxa. El problema esdevé, però, quan cal garantir la unicitat del nom en xarxes de gran abast. Per això existeixen organitzacions internacionals que atorguen noms específics que les organitzacions poden usar per generar els seus noms de dispositius amb garantia d'unicitat. Ens referim als noms de dominis.

Els noms de domini tenen la garantia que són únics en tot el món, per això els propietaris poden decidir d'usar-los directament com a nom d'un dispositiu o bé crear subdominis derivats per disposar de més noms. Els subdominis s'aconsegueixen afegint prefixos separats per un punt al nom de domini. Lògicament els prefixos seran també únics per a un domini específic de manera que la concatenació del prefix i el domini acabi donant sempre subdominis únics.

Exemples de subdominis

Un dels prefixos més usats com a nom de subdomini és `www`, però cal tenir en compte que pot servir qualsevol combinació, fins i tot diverses paraules separades per punts:

TAULA 1.1.

subdomini	prefix	domini
www.google.com	www	google.com
ioc.xtec.cat	ioc	xtac.cat
es.answers.yahoo.com	es.answers	yahoo.com

Servidors DNS

Malgrat tot, si en darrer terme l'adreçament en una xarxa cal fer-lo a través d'adreces IP, de què serveix poder identificar els dispositius mitjançant un nom de domini? Tard o d'hora caldrà identificar la IP del destí si cal enviar-hi dades.

Efectivament, però d'això se n'encarrega un servei d'Internet anomenat servidor de noms de domini, habitualment conegut com a DNS (*Domain Name System*). Els servidors DNS són serveis gratuïts d'Internet, que poden ser interrogats per un nom de domini o subdomini per tal d'aconseguir les adreces IP associades al nom. La xarxa de servidors DNS constitueix una immensa base de dades distribuïda de forma jeràrquica. La responsabilitat de mantenir les dades actualitzada correspon al propietari dels dominis de manera que el manteniment és mínim.

Les aplicacions que admetin noms de dispositius, hauran de consultar el servei DNS per trobar la IP a la qual hauran de direccionar la informació.

Capa d'enllaç i transmissió física del senyal

La capa d'enllaç és la més baixa de totes abans d'iniciar la transmissió física per mitjà de senyals electromagnètics. Des del punt de vista del programador d'aplicacions, és una capa tan allunyada i el seu funcionament queda tan amagat que no n'és necessari un coneixement gaire extensiu. A més, existeixen diversos protocols d'enllaç segons el tipus de xarxa en el qual ens cal transmetre.

Un exemple d'això el podem trobar en el fet que les enormes diferències existents entre el que serien xarxes locals i xarxes de gran abast obliga a disposar de protocols radicalment diferents. Però, a més, per tal de poder treure el millor rendiment a la xarxa, existeixen diversos protocols dins d'una mateixa categoria segons l'ús que se li doni, la quantitat de trànsit que hagi de suportar, la grandària dels paquets transmesos, o el sistema físic utilitzat

Les xarxes locals solen aglutinar una nombre reduït de dispositius i acostumen a connectar-se per mitjà de cables entrellaçats o sense fils a través d'emissors i receptors de baixa potència. El principal protocol utilitzat és Ethernet en les seves versions de cablejat (IEEE 802.3) o sense fils (IEEE 802.11). A grans trets, aquest protocol es basa en identificar els elements físics directament accessibles (targes de xarxa, punts d'accés sense fils, etc.) per mitjà d'una adreça única que

els identifica anomenada MAC (*Media Access Control*). En una xarxa local la informació s'envia encapçalada per l'adreça MAC a tots els dispositius connectats. Els dispositius amb una adreça diferent descartaran la informació. En canvi el dispositiu destinatari la tractarà i la farà arribar a les capes superiors.

Les xarxes de gran abast són aquelles que aglutinen molts dispositius diferents. Per exemple les companyies de telecomunicacions disposen de xarxes de gran abast, però també els governs, les institucions i organitzacions internacionals disposen de grans xarxes interconnectades entre elles constituint el que coneixem com a Internet. Alguns dels protocols de xarxa usats aquí són PPP (*Pear to Pear Protocol*) que representen el punt d'accés a les xarxes de gran abast de dispositius particulars i xarxes d'àmbit local o ATM per aconseguir els enllaçaments intermedis.

1.4 Elements de programació d'aplicacions en xarxa

Els llenguatges d'alt nivell disposen de biblioteques especialitzades en el desenvolupament d'aplicacions distribuïdes. Malgrat que cada llenguatge contempla les seves particularitats, tots ells presenten força elements comuns que donen resposta als conceptes bàsics d'adreçament, enviament d'informació, connexió i canal de transmissió o tractament de recursos remots de la forma més transparent possible.

Implementació de capes inferiors

És important saber que Java només permet la implementació a la capa d'aplicació perquè aquesta característica descarta aquest llenguatge per implementar solucions alternatives de baix nivell. En general Java és un llenguatge destinat a implementar solucions d'alt nivell.

El llenguatge Java aprofita la riquesa que li dóna el paradigma orientat a objectes per definir una jerarquia de classes que embolcalla tota aquesta potencialitat a través de mètodes de molt alt nivell que faciliten el desenvolupament d'aplicacions distribuïdes robustes. De fet, l'API que contempla la biblioteca estàndard de Java només permet treballar a nivell de la capa d'aplicació usant la interfície d'accés exclusiu a la capa de transport. El JDK estàndard no contempla la possibilitat d'accedir a les capes de més baixes (xarxa o enllaç).

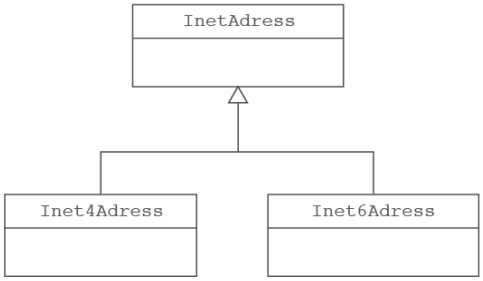
Es tracta d'una implementació realment eficient que dóna resposta a totes les necessitats originades des de la capa d'aplicació. Per tant no és necessari que ens plantegem si hem de fer servir altres alternatives, ni molt menys d'implementar-les nosaltres.

1.4.1 Adreçament

La classe `InetAddress` és una abstracció que ens permet gestionar de forma transparent adreces IP de qualsevol de les dues versions sense necessitat de tenir-ho en compte. Internament sempre es treballarà amb la classe corresponent a la versió adient (IPv4 o IPv6), que quedarà amagada al programador per la classe `InetAddress`. Això facilita molt la codificació ja que ens permet fer el mateix tractament amb independència de la versió IP usada.

Com a conseqüència d'aquesta abstracció, mai hauríem d'instanciar directament un objecte de la jerarquia invocant la sentència `new`, sinó que cal fer-ho a través d'un dels mètodes `static` que la classe `InetAddress` posa a la nostra disposició. Els més comuns són `getByName`, `getAllByName` `getByAddress` `getLoopbackAddress` i `getLocalHost`. Tots ells de cara al programador retornen una instància d' `InetAddress`, però internament si la IP utilitzada és de tipus IPv4 la instància serà de la classe `Inet4Address` però si és de tipus IPv6 serà de la classe `Inet6Address` (vegeu jerarquia a la figura 1.7).

FIGURA 1.7. Jerarquia de classes per gestionar l'adreçament a dispositius remots



El mètode `getByName` necessita un paràmetre de tipus `string`. El paràmetre podrà tractar-se d'una URL, d'un nom de xarxa identificador d'un dispositiu o d'una adreça IP en qualsevol dels formats acceptats pel protocol IP. En cas de tractar-se d'una cadena URL, la classe `InetAddress` farà una petició al servidor DNS per defecte, per aconseguir resoldre el nom subministrat i obtenir una adreça IP equivalent. Si la cadena passada conté un nom de xarxa local, la classe `InetAddress` usarà els protocols estàndards que permetin resoldre el nom i obtenir l'adreça IP associada. Les adreces IP obtingudes es convertiran a un vector de bytes de la mida corresponent a la versió de l'adreça (4 o 16 bytes). També, en cas que el paràmetre passat contingui directament una adreça IP, es farà la conversió a vector de bytes. El vector de bytes servirà per instanciar un objecte de tipus `Inet4Address` o `Inet6Address` d'acord amb la mida del vector. La instància creada es retornarà tot seguit com a resultat del mètode. En cas que el sistema usat per resoldre la URL o el nom de xarxa obtingui més d'una IP es crearà la instància d'una d'elles.

El mètode `getAllByName` rep també un paràmetre contenint una cadena amb les mateixes opcions que el mètode `getByName` i es comportarà de forma similar a l'hora de resoldre el nom passat, però si la resolució conté diverses adreces IP crearà una instància diferent per a cada una d'elles, les agruparà en un vector d'objectes `InetAddress` i les retornarà.

El mètode `getByAddress` espera rebre per paràmetre un vector amb 4 o 16 bytes. En cas que la mida del vector sigui de 4 bytes instanciarà un objecte de la classe `Inet4Address`. Seria de la classe `Inet6Address` si la mida fos de 16 bytes.

El mètode `getLoopbackAddress` obté una instància d' `InetAddress` fent servir l'adreça específica 127.0.0.1. De forma similar el mètode `getLocalHost` obté una instància de la connexió local principal de l'ordinador en el qual l'aplicació s'estigui executat.

En cas que el nom del paràmetre no es pugui resoldre o el format de l'adreça IP sigui incorrecta es llançarà una excepció del tipus `UnknownHostException`.

Les instàncies d' `InetAddress`, siguin de tipus `Inet4Address` o `Inet6Address` permeten totes obtenir informació del *host* associat a la IP. Entre d'altres el mètode `getAddress` ens retornarà un vector de bytes amb el valor de la IP associada. De forma semblant `getHostAddress` retorna també la IP però en format textual i el mètode `getHostName` obté el nom del dispositiu associat. Si durant la consulta el dispositiu no indiqués el nom, el mètode retornaria l'adreça IP en forma d'*string*.

En el següent exemple posem en pràctica l'ús d'alguns mètodes:

```
InetAddress[] addresses = new InetAddress[2];

addresses[0] = InetAddress.getLoopbackAddress();
addresses[1] = InetAddress.getByName("ioc.xtec.cat");
for(InetAddress adress: addresses){
    if(adress.isLoopbackAddress()){
        System.out.println(adress.getHostName() + " té una adreça loopback");
    }else{
        System.out.println(adress.getHostName() + " no té una adreça loopback")
            ;
    }
}
```

També cal destacar els mètodes que ens informen sobre quin tipus d'adreça IP conté l'objecte InetAddress. A l'exemple podeu veure l'ús del mètode `isLoopbackAddress` que ens informa si la IP és del tipus 127.0.0.x. De forma semblant `isSiteLocalAddress` ens indica si es tracta d'una adreça de xarxa privada com per exemple ho pot ser 192.168.4.18. El mètode `isMulticastAddress` ens indica també si la IP l'objecte és de tipus multicast.

Tim Berners-Lee. Font: Knight Foundation

1.4.2 Referències remotes i obtenció de recursos

Moltes vegades les aplicacions necessiten aconseguir recursos emmagatzemats en algun dispositiu de la xarxa. Sovint els recursos es troben emmagatzemats en forma de fitxer, però podrien també trobar-se emmagatzemats en una base de dades o obtenir-se com a resultat d'un procés.

El projecte WWW

Tim Berners-Lee és l'inventor del projecte Word Wide Web, més conegut per WWW o simplement web. Originàriament el projecte va néixer amb la intenció de permetre compartir documents i informació entre investigadors. La idea del WEB gira entorn al concepte d'*hipertex*, un enllaç capaç d'identificar a un recurs específic en un entorn web. Actualment Berners-Lee és professor del MIT i director del World Wide Web Consortium (W3C).

Berners-Lee defineix un **recurs** com «allò que té un identificador». Ens diu que malgrat que alguns recursos són virtuals i es troben a la xarxa, cal també considerar recursos les persones, les organitzacions, o d'altres entitats del món real. També aclareix que el concepte recurs no és pas sinònim d'entitat sinó més aviat una representació puntual d'una entitat o amb els seves paraules, «un mapa conceptual d'una entitat en un instant de temps». De fet els recursos poden romandre constants, les entitats no.

Per identificar un recurs del web, podem fer servir una URL (Uniform Resource Locator). Una URL és una cadena de caràcters única per a cada recurs diferent, que segueix unes determinades regles sintàctiques i conté informació d'on es troba

el recurs, de com es pot localitzar. En tractar-se d'una cadena única per a cada recurs, diem que a més de localitzar el recurs també l'identifica. Per això les URL es consideren també identificadors de recursos.

En general la sintaxi d'una URL segueix el format següent:

```
esquema://acces/nom_especific
```

En el qual `esquema` és una paraula composta exclusivament de lletres corresponents a l'alfabet anglosaxó. També admet els caràcters `:`, `.`, `-` i `+`. Representa el nom del protocol d'accés o del context en el qual es troba el recurs: HTTP, FTP, TELNET, FILE, JDBC... en són exemples.

`acces` respon a algun dels següent formats:

- dispositiu
- dispositiu:port
- nom_usuari:contrasenya@dispositiu
- nom_usuari:contrasenya@dispositiu:port

L'`acces` indica en quin dispositiu es troba el recurs i com hi podem accedir. També se'l coneix com a autoritat (en anglès *Authority*) perquè representa l'autoritat que té drets sobre el recurs. A la posició `dispositiu` podem escriure una adreça IP o el nom del domini, i subdomini si fos necessari, que identifica el dispositiu. La identificació del dispositiu és obligatòria, però tota la resta és opcional. En cas de no especificar el port pel qual accedir, per defecte es considerarà el port reservat per defecte al protocol que fem servir. Si l'accés és lliure no caldrà especificar ni el nom d'usuari ni la contrasenya. Però si cal autorització es pot afegir, tal com mostra el format.

El `nom_específic` és una cadena que identifica el recurs dins del dispositiu. Pot estar compost de dues parts separades per un caràcter `?`. La primera és obligatòria. En cas que la segona part fos opcional s'ometrà el caràcter `?`. És a dir que seguira algun d'aquest formats:

- `cami`
- `cami?altres_indicacions_de_cerca`

El `cami` estarà format per dígits i cadenes de caràcters de l'alfabet anglosaxó separades pel caràcter `/`. Representen una ubicació jeràrquica semblant als arbres de directoris dels sistemes de fitxers, perquè originàriament, el `cami` de la URL coincidia amb el `cami` del sistema de fitxers del servidor on es trobava emmagatzemat el recurs. Actualment, malgrat que pot coincidir, no sempre és així. De fet, pot només tractar-se d'una identificació lògica sense correspondència amb cap ruta real ni tan sols amb cap fitxer (les dades podrien estar emmagatzemades en una base de dades, però identificades per mitjà d'un `cami` lògic. Per indicar el

camí, s'accepten també caràcters especials com `.`, `-` o `_` a més dels dígits i l'alfabet anglosaxó. Tot i així, la necessitat d'internacionalitzar el sistema ha obligat a cercar una forma d'incorporar més caràcters. En cas que sigui necessari afegir caràcters no contemplats, podrà indicar-se el seu codi hexadecimal precedit d'un símbol `%`.

La part anomenada `altres_indicacions_de_cerca` (i que en anglès es coneix com a *query*) es troba constituïda per un nombre variable de parelles d'atribut-valor. Cada parella es trobarà separada de la següent per un símbol `&`. L'atribut apareixerà sempre en primer lloc (més a l'esquerra) i el valor en segon lloc immediatament després del símbol = que farà de separador. Veiem algun exemple:

Exemples de URLs

http://www.xtec.cat/documents/fp/dam/curriculum.html.

Es tracta d'una referència al recurs ubicat al camí *documents/fp/dam/curriculum.html* al qual podem accedir en el domini www.xtec.cat i usant el protocol HTTP. La referència usarà el port per defecte, és dir el port 80 que és el port per defecte del protocol HTTP.

ftp://admin:secret@ftp.debian.com/readme.txt.

Aquí es veu un URL que referencia el recurs *readme.txt*, usant el protocol FTP. El recurs es troba ubicat en el dispositiu que respon al domini `ftp.debian.com`. Per aconseguir el recurs cal enviar també el *login* i contrasenya que a l'exemple responen als valors `admin` i `secret` respectivament.

file:///home/user/recordatori.pdf

Aquí el recurs es troba en un disc local des del qual es fa la consulta seguint el camí *home/user/recordatori.pdf*.

http://www.servidor.com/cercar?tipus=compres&inici=2010-03-01&final=2010-03-31

Cas que cerca i obté a un recurs que representen les compres realitzades durant el mes de març registrades al servidor del domini www.servidor.com. El protocol d'accés és HTTP.

Classes per treballar amb referències a recursos

El Java Development Kit (JDK) de Java disposa de dues classes per poder treballar fàcilment amb recursos remots identificats amb una URL. Ens referim a els classes `URL` i `URLConnection`. Es tracta de classes de molt alt nivell que hauríem de situar dins la capa d'aplicació. Això permet al programador oblidar-se dels protocols més baixos i centrar la seva atenció en els protocols propis de l'aplicació.

La classe `URL` contempla dues atribucions. D'una banda representa la cadena que identifica el recurs i disposa de mètodes per tractar-la i extreure'n informació i de l'altre un punter al contingut del recurs.

Els principals mètodes que faciliten el tractament de la cadena localitzadora destriant la seva composició són:

- `String getProtocol()`: retorna una cadena amb el nom del protocol que la URL indica.
- `String getAuthority()`: obté l'autoritat extreta de la cadena URL.

- `String getPath()`: és la part que hem anomenat `cami` a la representació esquemàtica de la sintaxi de la URL.
- `String getQuery()`: invocant aquest mètode obtindrem la part anomenada `altres_indicacions_de_cerca`. En aquesta cadena s'exclou l'interrogant que la separa de `cami`.
- `String getFile()`: retorna la concatenació entre getPath() i getQuery(), mantenint entre ambdues parts el caràcter separador (un interrogant).
- `String getPort()`: en executar-lo retornarà el port especificat a la URL. Si no s'especifica port, retornarà -1.
- `String toString()`: obté tota la cadena de la URL. Aquest mètode és útil perquè la classe disposa de diversos constructors als quals se'ls pot passar la cadena sencera o per parts. Si fos així el constructor concatenarà les diverses parts passades per paràmetre i les formatarà adequadament amb els símbols adequats que les han de separar.

Els principals constructors són:

- `URL(String protocol, String acces, int port, String nomEspecificDelRecurs)`: constructor d'URL que crearà la cadena localitzadora, concatenant el protocol, l'accés, el port i el nom específic de recurs.
- `URL(String protocol, String acces, String nomEspecificDelRecurs)`: constructor d'URL que crearà la cadena de localització, concatenant el protocol, l'accés i el nom específic de recurs.
- `URL(URL context, String nomRelatiuDelRecurs)`: aquest constructor instancia la URL amb dos paràmetres i permet expressar un adreçament relatiu. Així la cadena del segon paràmetre actuarà com un localitzador relatiu al context passat en el primer paràmetre.
- `URL(String cadenaUrl)`: instanciarà un objecte URL a partir de la seva cadena passada per paràmetre.

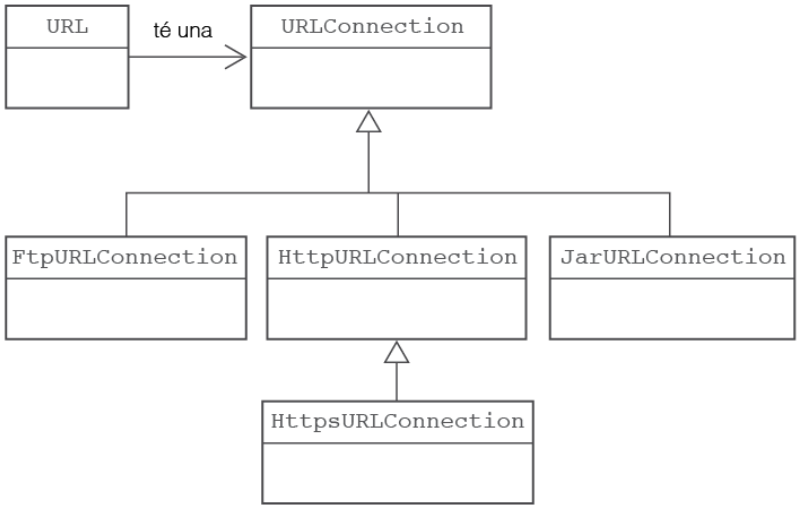
En referència a l'obtenció del contingut del recurs, podem invocar el mètode `openStream()` per aconseguir un flux d'entrada apuntant al contingut remot. És possible també obtenir directament una instància de `URLConnection` per disposar de més control sobre la connexió invocant el mètode `openConnection()`. La classe URL fa servir una instància com aquesta durant la invocació d' `openStream` per tal de poder retornar el flux de dades.

La classe `URLConnection` usarà el protocol TCP per fer la petició del recurs. Aquí el flux de dades cal obtenir-lo cridant `getInputStream()`. Amb aquesta classe tindrem a la nostra disposició alguns mètodes útils per obtenir informació del contingut. Cal però clarificar que es tracta d'informació extra (en forma de capçalera) aportada pel servidor al qual se li ha fet la petició del recurs. Podem

trobar-nos, doncs, que alguna vegada la informació no està del tot completa o fins i tot podria arribar a ser errònia. Entre d'altres, podem demanar la longitud del contingut (fent servir el mètode `getContentLength`), la codificació usada per emmagatzemar els caràcters o bé el tipus de dades que el recurs conté (imatge, text, XML, PDF, etc). La invocació necessària per obtenir el tipus de recurs és `getContentType` i per conèixer la codificació dels caràcters és `getContentEncoding`.

Tot i que la classe `URLConnection` pot suportar URLs de diversos protocols, el seu disseny està especialment pensat per donar suportar al protocol HTTP. És a dir, els seus mètodes suporten la funcionalitat descrita en el protocol HTTP, tot i que és capaç de treballar amb recursos remots apuntats per URLs que facin servir altres protocols. En realitat, `URLConnection` és una classe abstracta que delega la seva funcionalitat en les classes que l'estenen. Cada classe hereva d' `URLConnection` controlarà un protocol diferent (vegeu la figura 1.8). D'aquesta manera resulta força senzill afegir nous protocols o modificar els existents sense repercutir en la resta de classes.

Figura 1.8. Jerarquia de les classes que representen recursos remots



A nivell d'aplicació normalment treballarem sempre amb la classe `URLConnection` i no amb les seves derivades perquè així podem despreocupar-nos del protocol utilitzat i donar un tractament estàndard a totes les connexions. Quelcom semblant passa amb els objecte de tipus flux que seran retornats per aconseguir el contingut remot. La classe usada dependrà del tipus de contingut segons sigui imatge, PDF, text pla, o un document HTML, però pel programador totes elles caldrà tractar-les com una instància de la classe `InputStream`. Es tracta d'uniformitzar el tractament de tot el contingut tant com sigui possible.

Atenent que la classe `URLConnection` està orientada específicament al protocol HTTP, disposa d'un conjunt de mètodes que a més de recuperar el contingut, permeten obtenir informació extra definida en aquest protocol o bé enviar informació específica seguint les especificacions del protocol. Cal veure que es tracta de processos força genèrics comuns a diferents protocols i per tant és

possible reutilitzar la mateixa funcionalitat malgrat que estiguem fent servir altres protocols.

Recordeu que existeix una gran diversitat de formes de codificar els caràcters (ISO-8859-1, WINDOWS-1252, UTF-8...). Això és així perquè no existeix un únic model per representar qualsevol tipus de caràcter sinó que cada codificació és capaç de representar només un subconjunt més o menys ampli de caràcters. Actualment el sistema més universal és UTF.

D'acord amb l'especificació HTTP les dades viatgen embolcallades en una capçalera que conté metainformació del contingut. Generalment aquesta informació l'omple el servidor en el qual es troba ubicat el recurs. Es tracta d'informació opcional i per tant podem usar-la però cal tenir en compte que alguns recursos arribaran sense ella. Alguns d'aquests mètodes són:

Recordeu que la forma de representar una data és a partir d'un valor numèric que indica el nombre de mil·lisegons transcorreguts des del l'1 de gener de 1970.

- `String getContentEncoding()`. Obté la codificació de caràcters que el contingut del recurs fa servir.
- `int getContentLength()`. Aconsegueix la longitud del contingut que indica la capçalera.
- `String getContentType()`. Retorna el tipus de contingut d'acord amb l'estàndard HTTP. Aquesta especificació per exemple tipifica un document de text com a `text/plain`, un document HTML com a `text/html`, una imatge JPG com a `image/jpg`, etc.
  - `long getDate()`. Ofereix la data de creació/modificació del recurs con un valor numèric de tipus `long`

En cas que el tipus de contingut no es trobi informat a la capçalera podem fer servir el `guessContentTypeFromName` que intenta esbrinar el tipus a partir del nom del fitxer contingut a la URL. Aquest mètode pot servir per verificar o complementar la informació referent al tipus de dades del recurs.

Codi per comprovar si un recurs és una imatge de format GIF

Veieu a continuació un exemple que determina si un recurs en una imatge en format GIF comparant el valor retornat per els mètodes `getContentType` i `guessContentTypeFromName`. Fixeu-vos bé com se li passa el nom del fitxer fent servir el mètode de la classe URL pertinent.

```
boolean isGifFormat(URL url){
    boolean ret = false;
    try {
        URLConnection con = url.openConnection();
        String headerType = con.getContentType();
        String guessType = con.guessContentTypeFromName(url.
            getFile());
        ret = headerType.endsWith("gif") || guessType.endsWith("
            gif");
    } catch (IOException ex) {
        Logger.getLogger(URLUsage.class.getName()).log(Level.
            SEVERE, null, ex);
    }
    return ret;
}
```

`URLConnection` ens oferirà un flux de de dades per obtenir del contingut del recurs invocant el mètode `getInputStream()`. Es tracta del mateix flux de dades que obtindríem fent servir directament el mètode `openInputStream` de la classe URL perquè internament aquesta fa servir una instància d' `URLConnection` per accedir al contingut del seu recurs. És a dir, per aconseguir una instància d'un flux d'entrada podeu fer servir indiferentment qualsevol de les dues formes:

```
InputStream in = url.openStream();
```

o bé,

```
InputStream in = url.openConnection().getInputStream();
```

Vegeu l'annex "Manipulació dels fluxos" en la secció Annexos.

Cal tenir en compte a l'hora de treballar amb fluxos que només admeten de ser recorreguts en una única direcció i un cop han estat recorreguts no es poden tornar a recórrer. Normalment farem servir un *array* de caràcters o bé bytes per anar extraient de cop gran quantitat del contingut del flux fins extreure'l tot. Escollirem el tipus de dada de l'*array* segons el que contingui el recurs. Si es tracta d'un recurs de text podem convertir la instància d' `InputStream` en un objecte de tipus `InputStreamReader` per tal de disposar de mètodes més adients per tractar caràcters.

Obtenció de dades a través d'un flux de dades

Exemple de codi per extreure la informació arribada a través d'un flux de dades (`InputStream`).

```
private void printContent(URL url){
    InputStream in;
    char[] cbuf = new char[512];
    int caractersLlegits;

    if(!isText(url)){
        return;
    }

    try {
        in = url.openStream();
        InputStreamReader inr = new InputStreamReader(in);
        while((caractersLlegits=inr.read(cbuf))!=-1){
            String str = String.copyValueOf(cbuf, 0,
                caractersLlegits);
            System.out.print(str);
        }
        System.out.println();
    } catch (IOException ex) {
        Logger.getLogger(URLUsage.class.getName()).log(Level.
            SEVERE, null, ex);
    }
}
```

La classe `URLConnection` disposa també d'una connexió per enviar informació cap al recurs. De fet, HTTP especifica mecanismes per enviar dades des del client al servidor, per exemple extraient-les d'un formulari, o enviant un fitxer. Sovint les dades a enviar al servidor es concatenen juntament amb la mateixa cadena URL, constituint la part que hem anomenat `altres_indicacions_de_cerca` i que en anglès es coneix com a `query`. Malgrat tot HTTP també és capaç d'acceptar dades posteriors a la connexió a través d'una cadena URL. És el que es coneix com a mètode POST perquè primer s'estableix la connexió i immediatament després es comencen a enviar dades. El protocol HTTP estableix la seqüència fixa de passes que cal seguir per fer l'enviament amb èxit.

Imaginem que disposem d'una URL vàlida anomenada *url*. En primer lloc, caldrà obtenir la connexió fent,

```
URLConnection con = url.getConnection();
```

Després, cal indicar a la connexió que tenim intenció d'enviar dades invocant el mètode `setDoOutput`

```
con.setDoOutput(true);
```

i començar el procés d'enviament,

```
/*Suposarem que volem enviar caràcters i per tant usarem el tipus
 *OutputStreamWriter per facilitar el tractament*/
OutputStreamWriter out = new OutputStreamWriter(con.getOutputStream());
out.write("nomPropietat=" + valorPropietat);
```

En acabar cal tancar el flux de sortida per assegurar que s'envien totes les dades i s'alliberen els recursos,

```
out.close();
```

Com és natural, aquest sistema por fer-se servir també quan treballem amb altres protocols, perquè el diàleg específic propi del protocol es realitzarà internament dins de la classe específica (`HttpURLConnection`, `HttpsURLConnection`, `FtpUrlConnection`...) en el moment de fer la connexió i l'enviament de les dades.

1.4.3 Sòcols

API és l'acrònim d'Application Programming Interface.

Els sòcols, en anglès *sockets*, són una Interfície de Programació d'Aplicacions (API) que permet a dues aplicacions intercanviar informació malgrat que s'executin en dispositius diferents. Representen la porta d'entrada i sortida a la xarxa i constitueixen la base de qualsevol aplicació distribuïda.

> La documentació oficial de Java defineix els **sòcols** com el punt final de la comunicació bidireccional entre dos programes que s'executen a la xarxa.

Recordeu que l'arquitectura de les xarxes defineixen una pila de capes des de l'aplicació al medi físic per tal de fer factible la transmissió de dades entre programes. Els sòcols estarien situats a la capa de transmissió i representarien el punt d'accés de les aplicacions a les capes inferiors del sistema.

Per motius de seguretat i robustesa, Java no disposa de cap més utilitat que permeti treballar directament a les capes inferiors, per tant els sòcols es converteixen en la utilitat de programació de més baix nivell del llenguatge Java. Això significa que no és possible fer implementacions de protocols que es trobin per sota de la capa de transmissió.

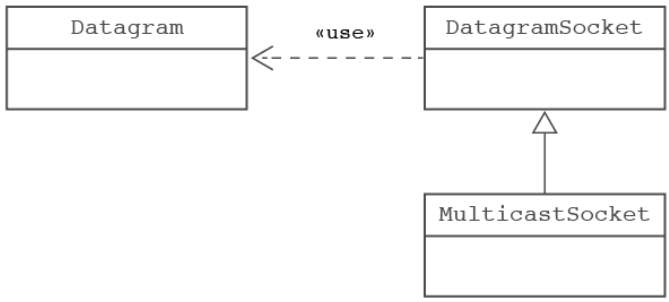
Els sòcols es troben associats a una IP i a un port de forma que sigui possible adreçar-hi informació a través de la xarxa fent servir algun dels protocols sobre IP disponibles (TCP o UDP).

Segons el protocol utilitzat parlarem de sòcols **no orientats a connexió** quan utilitzin el protocol UDP i de sòcols **orientats a connexió** quan utilitzin el protocol TCP.

1.4.4 Implementació de sòcols no orientats a connexió

El llenguatge Java contempla 3 classes a l'hora de fer implementacions de comunicació no orientada a connexió. `DatagramSocket` i `DatagramPacket` constitueixen les classes bàsiques de la comunicació a través d'UDP. La tercera, deriva de `DatagramSocket` i dóna suport específicament a les comunicacions de tipus multicast (vegeu la figura 1.9).

Figura 1.9. Jerarquia de classes per implementar sòcols no orientats a connexió



DatagramSocket

Els sòcols creats usant `DatagramSocket` són capaços d'enviar i rebre els paquets especificats pel protocol UDP. En crear una instància podem especificar un port concret que el sòcol escoltarà quan sigui necessari atendre algun servei (estàndard o no) associat al port. Pels sòcols temporals, en canvi, no caldrà establir el valor del port. Durant la creació de la instància la classe cercarà el primer port lliure dins el rang disponible per atendre comunicacions temporals.

- `DatagramSocket()`. Genera una instància de `DatagramSocket` associada a un port temporal.
- `DatagramSocket(int port)`. Genera una instància de `DatagramSocket` associada al port que s'indica en el paràmetre.
- `DatagramSocket(int port, InetAddress)`. En dispositius que tinguin més d'una IP es podrà especificar a través del segon paràmetre la IP que es vincularà la instància generada.

No cal especificar sempre la IP en dispositius amb diverses adreces, la classe en seleccionarà una per defecte durant la creació de la instància.

Els objectes de la classe `DatagramPacket` representen paquets de dades d'acord a l'especificació definida pel protocol UDP. La classe és capaç d'afegir per defecte gran part dels camps obligatoris del paquet. Tot i així cal indicar els bytes de dades a enviar, la longitud d'aquestes i també l'adreça i el port destí.

- `DatagramPacket(byte[] bufferDeDades, int longitudDades, InetAddress adreca, int port)`. Genera una instància de `DatagramPacket` que tindrà com a dades els bytes continguts al vector del primer paràmetre en la quantitat que indiqui el segon. Òbviament la longitud especificada no podrà superar mai la quantitat de bytes continguts al vector. El segon i tercer paràmetres permetran definir el destí de les dades.

Els objectes `DatagramSocket` disposen del mètode `send` per enviar un objecte `DatagramPacket` a l'adreça establerta en el propi paquet. A continuació veiem un exemple complet de creació d'instàncies i enviament d'un missatge de caràcters.

```
//bytes del missatge a enviar
byte[] missatge = "Salutacions".getBytes();
//adreça IP del destí
InetAddress adrecaDesti = InetAddress.getByName("localhost");
//port destí
int portDesti = 5555;
//creació del paquet a enviar
DatagramPacket packet = new DatagramPacket(missatge,
                                           missatge.length,
                                           adrecaDesti,
                                           portDesti);
//creació d'un sòcol temporal amb el qual realitzar l'enviament
DatagramSocket socket = new DatagramSocket();
//Enviament del missatge
socket.send(packet);
```

El procés de recepció de dades és molt similar, tot i que a l'hora de generar el paquet UDP cal crear-lo buit, però amb suficient capacitat per poder-hi copiar les dades rebudes.

```
//port a escoltar el servei que estem implementant
int portAEscoltar = 5555;
//vector de bytes en el qual rebre el missatge amb una capacitat de 1.024 bytes
byte[] missatge = new byte[1024];
//creació del paquet en el qual rebre les dades de 1.024 bytes com a màxim
DatagramPacket packet = new DatagramPacket(missatge,
                                           missatge.length);
//creació d'un sòcol que escolti el port passat per paràmetre
DatagramSocket socket = new DatagramSocket(portAEscoltar);
//recepció d'un paquet
socket.receive(packet);
```

Heu de tenir en compte, però, que rarament caldrà implementar un dispositiu que només hagi de fer enviaments o recepcions. Normalment caldrà definir petits diàlegs en els quals els dispositius hagin de rebre i enviar missatges successives vegades fins a completar l'objectiu de la comunicació. Els `DatagramSockets`, però, no són `full-duplex`, és a dir no poden processar al mateix temps un

enviament i una recepció, sinó que ho han de realitzar de forma seqüenciada, primer un procés i després l'altre.

Normalment un dels dispositius restarà escoltant a l'espera que d'altres iniciïn el diàleg. En general el dispositiu que escolta de manera indefinida fa el paper de servidor i els altres dispositius de client. Generalment una classe que implementi un servidor haurà de tenir algun mètode de configuració que permeti assignar el port que haurà d'atendre. També haurà de tenir un mètode que iniciï l'escolta del port, esperi la recepció de dades, les interpreti, n'obtingui la resposta, l'enviï al client que hagi fet la demanda i torni de nou a esperar una nova petició.

La implementació en Java del que acabem de descriure seria:

```
1  public class DatagramSocketServer {
2      DatagramSocket socket;
3
4      public void init(int port) throws SocketException{
5          socket = new DatagramSocket(port);
6      }
7
8      public void runServer() throws IOException{
9          byte [] receivingData = new byte[1024];
10         byte [] sendingData;
11         InetAddress clientIP;
12         int clientPort;
13
14         //el servidor atén el port indefinidament
15         while(true){
16             //creació del paquet per rebre les dades
17             DatagramPacket packet = new DatagramPacket(receivingData, 1024);
18             //espera de les dades
19             socket.receive(packet);
20             //processament de les dades rebudes i obtenció de la resposta
21             sendingData = processData(packet.getData(), packet.getLength());
22             //obtenció de l'adreça del client
23             clientIP = packet.getAddress();
24             //obtenció del port del client
25             clientPort = packet.getPort();
26             //creació del paquet per enviar la resposta
27             packet = new DatagramPacket(sendingData, sendingData.length,
28                                         clientIP, clientPort);
29             //enviament de la resposta
30             socket.send(packet);
31         }
32     }
33
34     private byte[] processData(byte[] data, int length) {
35         //procés diferent per cada aplicació
36         ...
37     }
38 }
```

Fixeu-vos que es tracta d'una classe prou genèrica per adaptar-se a la majoria d'aplicacions, tan sols caldria adaptar el mètode `processData` als requeriments específics de cada implementació.

Fixeu-vos també, que la classe `DatagramPacket` disposa de mètodes per obtenir la informació continguda en el paquet. Això, a banda d'aconseguir les dades rebudes, permet també saber a quina adreça i port caldrà enviar la resposta generada.

De forma semblant al servidor, es pot definir un client de manera genèrica tenint en compte que és el client el que iniciarà la comunicació i per tant cal un procés

específic que aquí anomenarem `getFirstRequest` per tal d'aconseguir les dades inicials que caldrà enviar al servidor. A més, cada cop que el servidor ens enviï la resposta al nostre requeriment, caldrà fer-la arribar als interessats a través del mètode `getDataToRequest` que rebrà o bé un nou requeriment pel servidor o bé el senyal per finalitzar la connexió i abandonar l'execució de l'aplicació client.

Podeu imaginar que segons de quina aplicació client, `getDataToRequest` haurà de fer diferents coses amb les dades rebudes i obtenir de diferent manera les noves dades a enviar. Per exemple algunes aplicacions requeriran només mostrar les dades per pantalla i esperar la resposta de l'usuari, però d'altres potser requeriran també emmagatzemar les respostes en una fitxer o base de dades, obtenir la resposta mitjançant un procés automatitzat, etc. Sigui com sigui, el procés caldrà realitzar-lo sempre a través del mètode `getDataToRequest` per tal d'aconseguir un client prou adaptable.

El client també ha de ser capaç de reconèixer el senyal de finalització indicat per l'usuari, per exemple, en una de les seves respostes o com a resultat d'un procés que avalua que ja no cal continuar més la comunicació amb el servidor. En el nostre exemple aconseguim un procés neutre incloent el mètode `mustContinue` al qual se li passarà les dades retornades per `getDataToRequest` i ens retornarà si cal o no continuar la comunicació. Usarem aquest valor *booleà* per mantenir-nos dins del bucle o bé per abandonar-lo.

Abans d'iniciar la comunicació caldrà haver configurat el client indicant el nom del servidor i el port on es trobarà escoltant. Per això usarem el mètode `init`, malgrat que també es podria implementar en el propi constructor.

```
1  public class DatagramSocketClient {
2      InetAddress serverIP;
3      int serverPort;
4      DatagramSocket socket;
5
6      public void init(String host, int port) throws SocketException,
7                                                      UnknownHostException{
8          serverIP = InetAddress.getByName(host);
9          serverPort = port;
10         socket = new DatagramSocket();
11     }
12
13     public void runClient() throws IOException{
14         byte [] receivedData = new byte[1024];
15         byte [] sendingData;
16
17         //a l'inici
18         sendingData = getFirstRequest();
19         //el servidor atén el port indefinidament
20         while(mustContinue(sendingData)){
21             DatagramPacket packet = new DatagramPacket(sendingData,
22                                                        sendingData.length,
23                                                        serverIP,
24                                                        serverPort);
25             //enviament de la resposta
26             socket.send(packet);
27
28             //creació del paquet per rebre les dades
29             packet = new DatagramPacket(receivedData, 1024);
30             //espera de les dades
31             socket.receive(packet);
32             //processament de les dades rebudes i obtenció de la resposta
```

```
            sendingData = getDataToRequest(packet.getData(), packet.getLength()
                );
        }
    }

    private byte[] getDataToRequest(byte[] data, int length) {
        //procés diferent per cada aplicació
        ...
    }

    private byte[] getFirstRequest() {
        //procés diferent per cada aplicació
        ...
    }

    private boolean mustContinue(byte[] sendingData) {
        //procés diferent per cada aplicació
        ...
    }
}
```

En aquesta implementació s'ha optat per rellançar tots els llançaments d'excepcions per tal de deixar el tractament d'errors pel darrer nivell d'implementació quan es pugui decidir què s'ha de fer amb el errors. Tot i així, hi ha una excepció que caldria controlar aquí. En cas que el servidor tingui problemes i no sigui capaç de contestar el requeriment d'un client, aquest podria romandre esperant la resposta indefinidament. És possible evitar aquesta situació fent servir el mètode `setSoTimeout` que limitarà el temps d'escolta només a la quantitat de mil·lisegons que se li passi com a paràmetre. Si el `DatagramSocket` sobrepassa el temps d'espera es llançarà una excepció de tipus `SocketTimeoutException`. Ara bé, no ens interessa que el llançament provoqui abandonar el bucle i per tant la finalització de l'execució, sinó que és preferible fer una notificació sense sortir del bucle `while`. El codi de l'interior del bucle podria quedar com segueix:

```
DatagramPacket packet = new DatagramPacket(sendingData,
                                           sendingData.length,
                                           serverIP,
                                           serverPort);
//enviament de la resposta
socket.send(packet);

//creació del paquet per rebre les dades
packet = new DatagramPacket(receivedData, 1024);
//espera de les dades, màxim 5 segons
socket.setSoTimeout(5000);
try{
    //espera de les dades
    socket.receive(packet);
    //processament de les dades rebudes i obtenció de la resposta
    sendingData = getDataToRequest(packet.getData(), packet.getLength());
}catch(SocketTimeoutException e){
    sendingData = timeoutExceeded(packet);
}
```

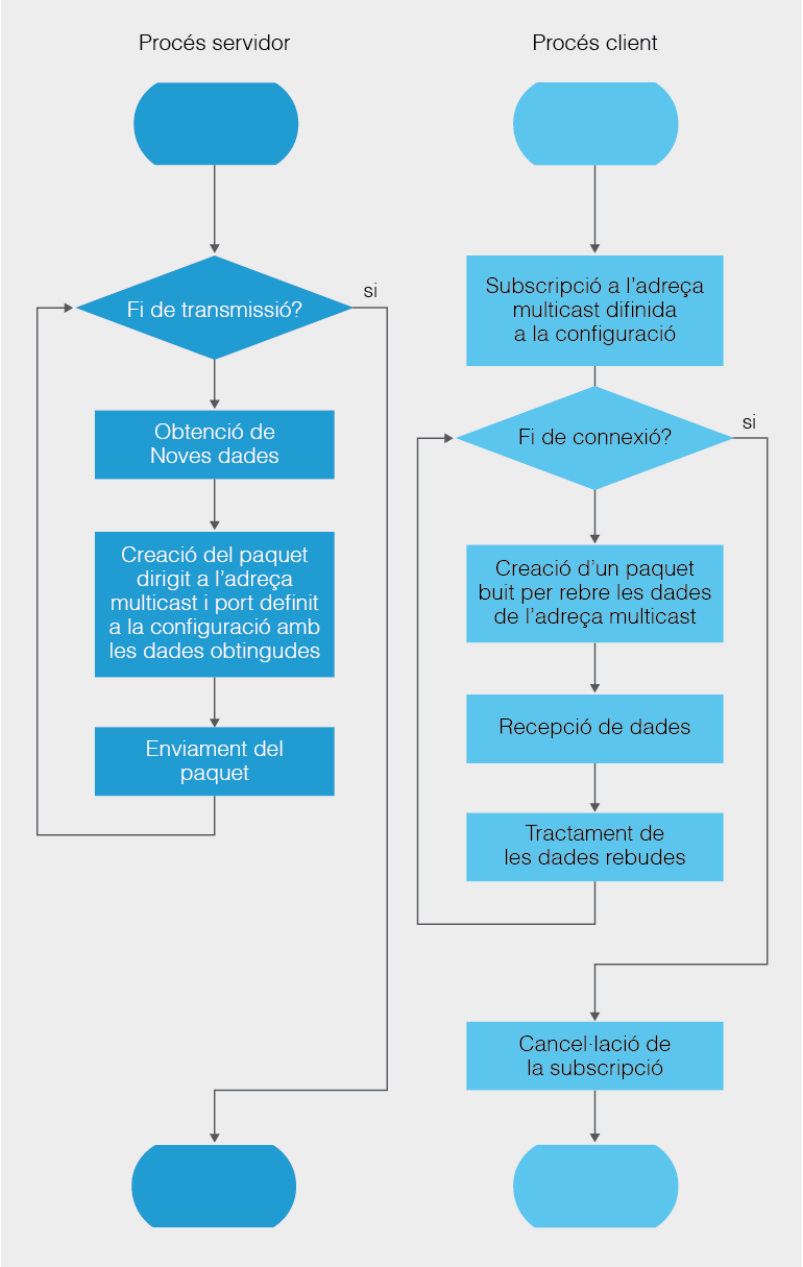
Per últim ambdues classes haurien de tenir també un mètode que permeti fer el tancament de l'objecte `DatagramSocket` abans d'acabar amb l'execució del servidor o del client. Per exemple:

```
public void close(){
    if(socket!=null && !socket.isClosed()){
        socket.close();
    }
}
```

MulticastSocket

La jerarquia de sòcols UDP de Java contempla també la classe `MulticastSocket`. Amb aquesta classe és possible crear aplicacions que usin adreces multicast. Recordeu que les adreces multicast permeten als dispositius subscriure's a la difusió de tot el que arribi per l'adreça a la qual s'ha subscrit. Qualsevol objecte `MulticastSocket` podrà fer enviaments a tots els dispositius subscrits i rebre tot allò que vagi dirigit a l'adreça associada. Malgrat tot, generalment les aplicacions `multicast` fan els enviaments des d'un servidor i els clients subscrits es limiten a escoltar i processar les dades, de forma semblant al que es veu a la figura 1.10.

FIGURA 1.10. Diagrama de flux del procés d'un servidor multicast



L'esquema simula un servidor que només envia dades d'una manera contínua

L'ús de la classe `MulticastSocket` és molt semblant a la classe `DatagramSocket`, pel que fa a la recepció i l'enviament de dades perquè ambdues instancien els paquets com a objectes `DatagramPaquet` i fan servir el protocol UDP per a la transmissió. Tot i així, cal assenyalar que la classe `MulticastSocket` disposa d'un mètode per realitzar la subscripció (`joinGroup`) i un altre per cancel·lar-la (`leaveGroup`). A més destacarem que els sòcols Multicast han de treballar tots en el mateix port, ja siguin clients o servidors. Anem a veure com podem plasmar l'esquema anterior amb Java. Començarem codificant un servidor genèric. El mètode `init` permetrà configurar les instàncies per mitjà dels paràmetres, l'adreça multicast representada com un *string* i el port comú en el qual enviar totes les dades.

```
public class MulticastSocketServer {
    MulticastSocket socket;
    InetAddress multicastIP;
    int port;
    boolean continueRunning = true;

    public void init(String strIp, int portValue) throws SocketException,
                                                          IOException{
        socket = new MulticastSocket(portValue);
        multicastIP = InetAddress.getByName(strIp);
        port = portValue;
    }

    public void runServer() throws IOException{
        DatagramPacket packet;
        byte [] sendingData;

        //el servidor fa enviaments continus mentre calgui
        while(continueRunning){
            //obtenció de les dades a enviar. Suposarem l'existència d'un
            //mètode específic per a les diferents aplicacions
            sendingData = getNextData();
            //creació del paquet per enviar les dades obtingudes
            packet = new DatagramPacket(sendingData, sendingData.length,
                                        multicastIP, port);
            //enviament de les dades
            socket.send(packet);

            //mirem si cal acabar la transmissió
            continueRunning = !transmissionFinished();
        }
    }

    public void close(){
        if(socket!=null && !socket.isClosed()){
            socket.close();
        }
    }
}
```

Destacarem que la funcionalitat del mètode `getData` dependrà de cada aplicació. També veiem que es contempla el mètode `close` per a preveure el tancament del sòcol abans de finalitzar l'execució del servidor.

El client, abans d'iniciar el bucle per rebre totes les dades, haurà de subscriure's a l'adreça multicast i en finalitzar el bucle caldrà cancel·lar la subscripció per reduir despeses inútils de recursos i tràfic a la xarxa innecessari.

```
public class MulticastSocketClient {
    MulticastSocket socket;
    InetAddress multicastIP;
    int port;

    public void init(String strIp, int portValue) throws SocketException,
                                                        IOException{
        multicastIP = InetAddress.getByName(strIp);
        port = portValue;
        socket = new MulticastSocket(port);
    }

    public void runClient() throws IOException{
        DatagramPacket packet;
        byte [] receivedData = new byte[1024];
        boolean continueRunning = true;

        //activem la subscripció
```

```
19          socket.joinGroup(multicastIP);
20
21          //el client atén el port fins que decideix finalitzar
22          while(continueRunning){
23              //creació del paquet per rebre les dades
24              packet = new DatagramPacket(receivedData, 1024);
25              //espera de les dades, màxim 5 segons
26              socket.setSoTimeout(5000);
27              try{
28                  //espera de les dades
29                  socket.receive(packet);
30                  //processament de les dades rebudes i obtenció de la resposta
31                  continueRunning = getData(packet.getData(), packet.getLength())
                        ;
32              }catch(SocketTimeoutException e){
33                  //s'ha excedit el temps d'espera i cal saber què s'ha de fer
34                  continueRunning = timeoutExceeded();
35              }
36          }
37
38          //es cancel·la la subscripció
39          socket.leaveGroup(multicastIP);
40      }
41
42
43      public void close(){
44          if(socket!=null && !socket.isClosed()){
45              socket.close();
46          }
47      }
48
49  }
```

1.4.5 Implementació de sòcols orientats a connexió

Els sòcols orientats a connexió fan servir el protocol TCP. Recordeu que en aquest cas les dades es passen a l'aplicació en el mateix ordre en què han sortit i s'assegura que no es perd informació durant la transmissió. El protocol TCP defineix que abans de començar la transmissió de dades cal fer una petició de connexió que l'altre part haurà d'acceptar. Un cop acceptada la connexió, en ambdós costats es reservarà un port de xarxa exclusivament per a la transmissió de dades en qualsevol dels dos sentits. Recordeu que TCP és un protocol que defineix una comunicació *full-duplex* exclusiva entre dos dispositius.

Java té una manera pròpia d'implementar aquestes característiques. Dels dos dispositius a posar en contacte, un d'ells farà el paper de servidor esperant la demanda d'algun dispositiu, que jugarà el paper de client. El procés del servidor usarà una instància de la classe `ServerSocket` per restar escoltant fins que un client faci una petició, moment en què acceptarà la connexió i generarà una instància de tipus sòcol per mantenir la comunicació *full-duplex*, des d'un port temporal, durant la transmissió. El procés dels clients en canvi, usarà directament un objecte de la classe sòcol per fer la petició i també per mantenir la connexió i la transmissió de dades.

Per a la transmissió de dades es fa servir el concepte de flux de dades. Es considera que les dades flueixen de forma contínua i seqüencial des de l'origen fins al destí.

La característica de *seqüencial* és essencial aquí, ja que garanteix que l'ordre d'arribada a la destinació és el mateix que l'ordre de sortida des de l'origen.

El concepte de flux (*stream* en anglès) és una abstracció relacionada amb qualsevol procés de transmissió d'informació entre un origen i un destí.

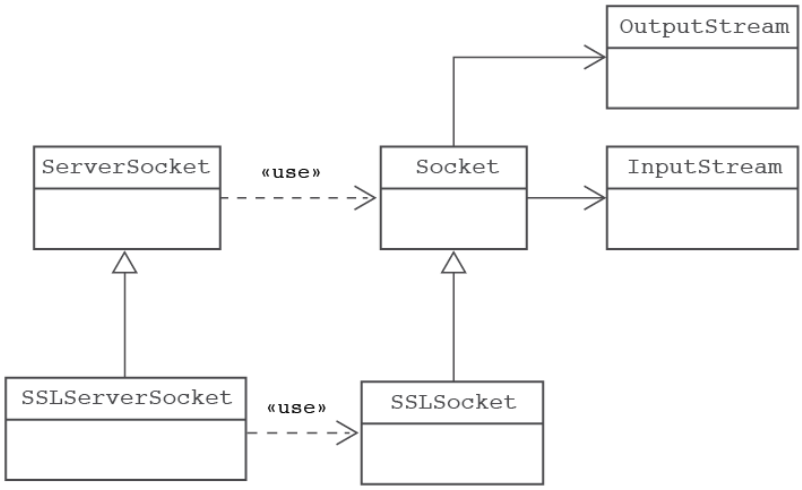
El flux de dades segueix un únic sentit (de l'origen al destí), per tant les aplicacions que necessitin transmetre i rebre hauran de fer servir dos fluxos, un en cada sentit.

El flux que rep dades des d'un origen extern s'anomena **flux d'entrada** (*input stream* en anglès) i el que envia informació a un destí exterior s'anomena **flux de sortida** (*output stream* en anglès).

Els objectes de tipus sòcol contenen dos fluxos de dades a través d'un d'ells es rebran les dades (flux d'entrada) i a través de l'altre s'enviaran (flux de sortida). En aplicacions client servidor, el flux de sortida del servidor equival al flux d'entrada del client i el flux d'entrada del servidor coincidirà amb el de sortida del client.

A la figura 1.11 podeu veure el diagrama de les classes que entren en joc en una transmissió TCP de Java. Les classe `SSLServerSocket` i `SSLSocket` ofereixen la mateixa funcionalitat que `ServerSocket` i `Socket` respectivament però estan especialitzades en transmetre de forma segura usant un protocol de seguretat anomenat SSL.

Figura 1.11. Jerarquia de classes per implementar sòcols orientats a connexió



La implementació que el llenguatge Java fa del protocol TCP passa per disposar en un dels dispositius a comunicar, el que faci el paper de servidor, d'una instància de `ServerSocket` que tindrà per objectiu únicament esperar que algun client decideixi fer una petició de comunicació, utilitzant els mínims recursos possibles mentre duri l'espera. El port a atendre, s'indicarà en el moment de creació de la instància de ServerSocket com a paràmetre del constructor.

Creació d'un ServerSocket que atendrà el port 7777 a l'espera de clients que demanin de comunicar-se

```
1        ServerSocket serverSocket = new ServerSocket(7777);
```

El client necessitarà una instància de Socket, la qual disposarà des del primer moment dels recursos propis del protocol TCP, ja que les instàncies de Socket solen crear-se indicant l'adreça i el port amb el que caldrà comunicar-se i des de la creació que es realitza la petició inicial per poder generar els recursos pertinents com els fluxos de dades que simularan els dos canals de comunicació que el protocol TCP demana.

Creació d'un sòcol en un client que vulgui comunicar-se amb un servidor que es trobi atenent el port 7777 a l'adreça 192.168.4.1

```
1        Socket socket = new Socket(new InetAddress.getByName("
             192.168.4.1"), 7777);
```

Durant la creació del sòcol del client s'enviarà al servidor la petició, el qual en rebre-la l'acceptarà creant un sòcol específic per comunicar-se amb el client acceptat a la banda del servidor. Recordeu que la comunicació TCP és exclusiva per dos dispositius, per això el servidor ha d'anar creant un sòcol específic per cada client que faci la petició.

La creació del sòcol a la banda del servidor es troba automatitzat i s'obté com a retorn del mètode de `ServerSocket` que inicia l'escolta del port fins l'arribada de la petició. Ens referim al mètode `accept`.

Mètode accept

```
1        Socket socket = serverSocket.accept();
```

El mètode accept resta a l'espera d'una petició i en el moment de produir-se crea una instància específica de sòcol per suportar la comunicació amb el client acceptat.

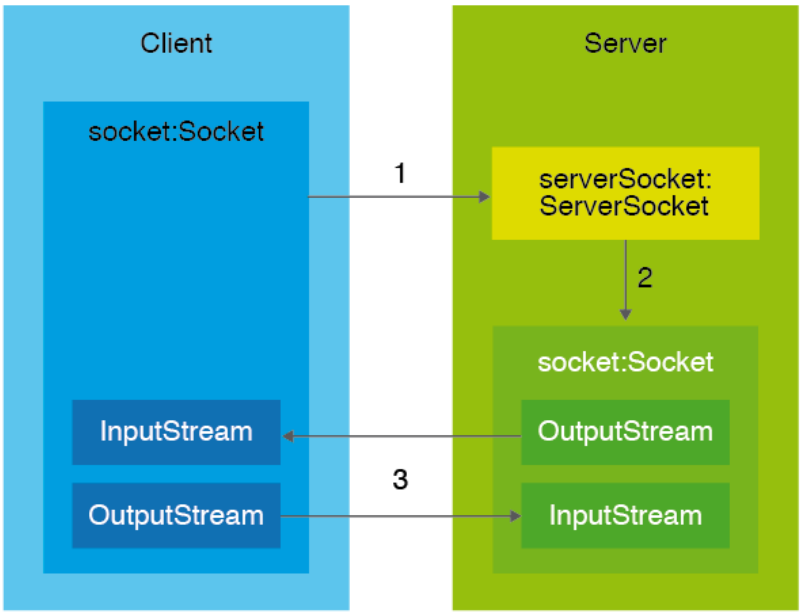
A partir d'aquí ambdós sòcols, el de la part client i el de la part servidor, disposaran d'un canal d'entrada i un de sortida adequadament connectats. La implementació dels canals es realitzarà a través d'instàncies internes de fluxos d'entrada i sortida. Per obtenir els fluxos d'un sòcol caldrà invocar els mètodes `getInputStream` i `getOutputStream` per obtenir respectivament els fluxos d'entrada i sortida.

Usarem els mètodes propis del fluxos per enviar o rebre dades a través d'ells. L'enviament a través del flux de sortida d'un dels sòcols implicarà la recepció de les mateixes dades enviades a través del flux d'entrada de l'altre sòcol. Podeu veure l'esquema del procés descrit a la figura 1.12 on:

- 1: El sòcol de la part client demana connectar-se a l'adreça en la qual es trobi el servidor i al port que escolti una instància de `ServerSocket` del servidor.
- 2: El `ServerSocket` accepta la connexió i crea un sòcol a la part del servidor amb dos canals, un d'entrada i un de sortida.
- 3: l' `InputStream` (flux d'entrada) del sòcol de la part client es troba connectat a l' `OutputStream` (flux de sortida) del sòcol de la part servidor

i el seu flux d'entrada connecta amb el flux de sortida del sòcol del client. D'aquesta manera és possible la comunicació en ambdues direccions amb la qualitat exigida pel protocol TCP.

Figura 1.12. Esquema del procediment seguit per connectar amb una servidor TCP de Java



A continuació veurem un exemple senzill de la implementació d'un diàleg entre un servidor i un client en el qual alternen consecutivament petició i resposta. Hem pres la consideració que tant les peticions com les respostes s'inclouen en una única línia de caràcters. D'aquesta manera cada part pot saber quan acaba l'enviament remot per tal de processar les dades i generar una resposta o una nova petició.

A la classe `TcpSocketServer` que representarà una aplicació de tipus servidor destacarem el mètode `listen` en el qual implementarem un bucle continuu en el qual una instància de ServerSocket anirà escoltant a cada iteració el port a atendre esperant clients que es connectin i acceptant les seves peticions a instanciar un sòcol. La comunicació entre ambdós objectes de tipus sòcol (el del client i l'acabat de generar) la implementarem en un mètode a part que anomenarem `proccesClientRequest`.

```
1  public class TcpSocketServer {
2      static final int PORT=9090;
3      private boolean end=false;
4
5      public void listen(){
6          ServerSocket serverSocket=null;
7          Socket clientSocket = null;
8          try {
9              serverSocket = new ServerSocket(PORT);
10
11             while(!end){
12                 clientSocket = serverSocket.accept();
13                 //processem la petició del client
14                 proccesClientRequest(clientSocket);
```

```
15                  //tanquem el sòcol temporal per atendre el client
16                  closeClient(clientSocket);
17              }
18              //tanquem el sòcol principal
19              if(serverSocket!=null && !serverSocket.isClosed()){
20                  serverSocket.close();
21              }
22
23          } catch (IOException ex) {
24              Logger.getLogger(TcpSocketServer.class.getName()).log(Level.SEVERE,
                     null, ex);
25          }
26      }
27
28      ...
29  }
```

El mètode per gestionar les peticions del client (`proccesClientRequest`) constarà d'un bucle continu que iterarem fins que el client ens indiqui que no té més peticions a realitzar. Per facilitar la dinàmica de pregunta resposta, convindrem que tant la petició com la resposta s'inclouran en un sola línia de caràcters. D'aquesta manera és fàcil detectar el final del missatge en qualsevol dels dos costats.

```
1   ...
2   public void proccesClientRequest(Socket clientSocket){
3       boolean farewellMessage=false;
4       String clientMessage="";
5       BufferedReader in=null;
6       PrintStream out=null;
7       try {
8           in = new BufferedReader(new InputStreamReader(clientSocket.
                getInputStream()));
9           out= new PrintStream(clientSocket.getOutputStream());
10          do{
11              //processem el missatge del client i generem la resposta. Si
12              //clientMessage és buida generarem el missatge de benvinguda
13              String dataToSend = processData(clientMessage);
14              out.println(dataToSend);
15              out.flush();
16              clientMessage=in.readLine();
17              farewellMessage = isFarewellMessage(clientMessage);
18          }while((clientMessage)!=null && !farewellMessage);
19      } catch (IOException ex) {
20          Logger.getLogger(TcpSocketServer.class.getName()).log(Level.SEVERE,
                null, ex);
21      }
22  }
23  ...
```

Per més detalls, vegeu l'annex "Manipulació de fluxos" de l'apartat Annexos.

En tractar-se d'un diàleg basat en cadenes de caràcters delimitades per un final de línia, podem fer servir els fluxos específics orientats a caràcters que disposa el llenguatge. Recordeu que Java permet convertir qualsevol flux en un flux específic orientat a caràcters durant el procés de construcció.

El mètode `isFarewellMessage` indicarà si el missatge del client es correspon amb una petició de finalització de la comunicació. Si no s'hi correspon, caldrà processar la petició i obtenir la resposta mitjançant el mètode `processData`. La resposta obtinguda s'enviarà al client a través d'un *stream* de tipus `PrintStream` que afegirà al final de l'enviament un símbol de final de línia.

Forçarem l'enviament de les dades invocant el mètode `flush` i seguidament esperarem un nova petició del client, començant de nou l'anàlisi i si cal processant una nova iteració del bucle.

Per tal de gestionar de forma eficient els recursos del servidor, és important tancar les instàncies de sòcol dels clients que hagin demanat la finalització de la comunicació. Implementarem el tancament en un mètode específic anomenat `closeClient` que invocarem des del mètode `listen` tot just en acabar de retornar de la invocació del mètode `processClientRequest`.

```
...
private void closeClient(Socket clientSocket){
    //si falla el tancament no podem fer gaire cosa, només enregistrar
    //el problema
    try {
        //tancament de tots els recursos
        if(clientSocket!=null && !clientSocket.isClosed()){
            if(!clientSocket.isInputShutdown()){
                clientSocket.shutdownInput();
            }
            if(!clientSocket.isOutputShutdown()){
                clientSocket.shutdownOutput();
            }
            clientSocket.close();
        }
    } catch (IOException ex) {
        //enregistrem l'error amb un objecte Logger
        Logger.getLogger(TcpSocketServer.class.getName()).log(Level.SEVERE,
             null, ex);
    }
}
...
```

És important seguir la seqüència d'instruccions del mètode `closeClient` si volem assegurar una tancament correcte. Els mètodes `shutDownInput` i `ShutDownOutput` engeguen ambdues peticions de tancament necessàries en el protocol TCP. La invocació del mètode `shutDownInput` enviarà la petició de tancament al sòcol remot i esperarà la confirmació de la mateixa. El sòcol remot en rebre una demanda de tancament del seu canal de sortida forçarà un darrer enviament de les dades contingudes al *buffer* abans d'enviar la confirmació. Quan el servidor local rebi la confirmació alliberarà el bloqueig de lectura del seu `InputStream`. De forma semblant, la invocació de `ShutDownOutput`, provocarà el forçament de l'enviament de les darreres dades contingudes al *buffer* de sortida local i enviarà la petició de tancament del canal d'entrada del sòcol remot. El sòcol remot abans de confirmar el tancament alliberarà el bloqueig de lectura dels seu `InputStream`.

Un cop fetes les peticions de tancament i rebudes les confirmacions podrà processar-se el tancament local dels fluxos i seguidament el tancament del sòcol. Cal tenir en compte que els tancament dels objectes sòcol no impliquen el tancament del servidor, sinó que aquest restarà disponible per acceptar noves peticions ja que el "serverSocket" restarà obert en el bucle inicial del mètode "listen". Quan decidim fer el tancament de tot el servidor caldrà tancar també la instància del "serverSocket". A la part client, a la classe `TcpSocketClient` implementarem la connexió en el mètode `connect`. Aquí iniciarem un sòcol per tal que demani una petició de connexió TCP a l'adreça i port passada per paràmetre

i iniciarem un bucle continu que permeti anar fent peticions i rebent les respostes aplicant la mateixa consideració que ambdues aniran delimitades per un final de línia.

```
public class TcpSocketClient{

    public void connect(String address, int port) {
        String serverData;
        String request;
        boolean continueConnected=true;
        Socket socket;
        BufferedReader in;
        PrintStream out;
        try {
            socket = new Socket(InetAddress.getByName(address), port);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream
                ()));
            out = new PrintStream(socket.getOutputStream());
            //el client atén el port fins que decideix finalitzar
            while(continueConnected){
                serverData = in.readLine();
                //processament de les dades rebudes i obtenció d'una nova
                    petició
                request = getRequest(serverData);
                //enviament de la petició
                out.println(request);//assegurem que acaba amb un final de lí
                    nia
                out.flush(); //assegurem que s'envia
                //comprovem si la petició és un petició de finalització i en
                    cas
                //que ho sigui ens preparem per sortir del bucle
                continueConnected = mustFinish(request);
            }

            close(socket);
        } catch (UnknownHostException ex) {
            reportError("Error de connexió. No existeix el host", ex);
        } catch (IOException ex) {
            reportError("Error de connexió indefinit", ex);
        }
    }
    ...
}
```

Per tal que funcioni correctament aquest algoritme serà necessari disposar d'un mètode especific anomenat `getRequest` que processi la resposta rebuda i aconsegueixi una nova petició que acabarà enviant-se al servidor. L'algoritme fa servir també el mètode `mustFinish` per detectar si la petició enviada implica una finalització de la connexió per tal de forçar la sortida del bucle.

Fora del bucle caldrà fer el tancament del sòcol de forma semblant de com s'ha realitzat en el servidor, assegurant l'ordre adequat de les peticions. És important comprovar si ja s'ha realitzat la petició del tancament de la connexió TCP usant els mètodes `isShutDownInput` o `isShutDownOutput` perquè no sabem quina de les parts (el servidor o el client) farà primer la petició.

```
...
    private void close(Socket socket){
        //si falla el tancament no podem fer gaire cosa, només enregistrar
        //el problema
        try {
            //tancament de tots els recursos
            if(socket!=null && !socket.isClosed()){
                if(!socket.isInputShutdown()){
```

```
9                   socket.shutdownInput();
10              }
11              if(!socket.isOutputShutdown()){
12                  socket.shutdownOutput();
13              }
14              socket.close();
15          }
16      } catch (IOException ex) {
17          //enregistrem l'error amb un objecte Logger
18          Logger.getLogger(TcpSocketClient.class.getName()).log(Level.SEVERE,
                null, ex);
19      }
20  }
21  ...
```

2. Generació de serveis en xarxa

L'èxit de la xarxa global no ha estat només el desenvolupament d'una connectivitat barata i eficient. De fet, des dels inicis, a la connectivitat van afegir-se un nombre creixent d'aplicacions, suportades pels operadors d'Internet, que intentaven donar respostes genèriques a les necessitats d'intercanvi d'informació demandades pels usuaris.

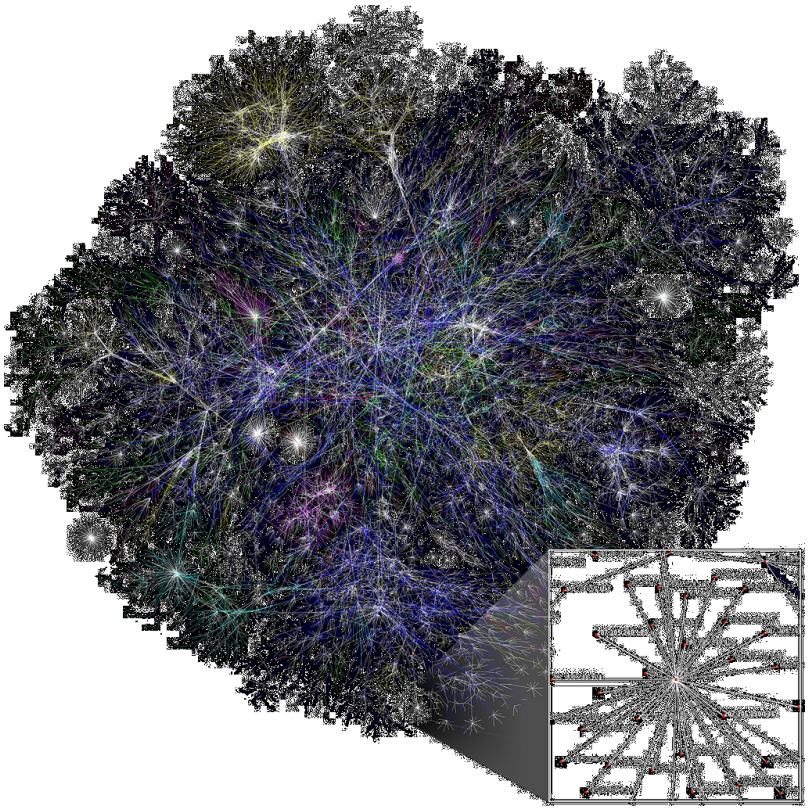
Es tracta d'aplicacions distribuïdes més o menys especialitzades en algun tipus d'intercanvi, que segueixen el model client-servidor i que s'han popularitzat amb el nom de **serveis de xarxa**.

> Podem dir que un **servei de xarxa** és aquell procés que executa una tasca predeterminada i ben definida dirigida a respondre les peticions concretes realitzades des de dispositius remots que actuen a mode de clients.

L'encert d'Internet ha estat l'estandardització d'un conjunt de serveis que podríem anomenar "bàsics", que són capaços de suportar nous serveis de més alt nivell, garantint així la interoperabilitat de tots ells, sense necessitat de noves instal·lacions ni canvis de programari en cap dels clients que hagi d'utilitzar-los.

L'estandardització ha permès un creixement exponencial d'Internet amb una infraestructura comuna relativament reduïda i totalment descentralitzada en subxarxes propietat d'empreses i organitzacions sense ànim de lucre. D'aquí que Internet rebi el nom de xarxa de xarxes. Podeu veure un mapa que il·lustra l'estructura d'Internet a la figura 2.1.

FIGURA 2.1. Esquema visual que mostra el mapa de les connexions d'Internet



Cada color representa una subxarxa dins la xarxa global. Aquí podeu observar la gran quantitat de graus d'interacció de cada node i l'organització de la xarxa. Font: Wikimedia Commons

La majoria dels serveis estàndards existeixen des dels inicis de la xarxa i responen a funcionalitats específiques d'intercanvi d'informació. Alguns poden facilitar l'ús de la pròpia xarxa com per exemple els serveis de **sistemes de noms de domini** anomenats també serveis **DNS** que permeten obtenir les adreces IP associades a un nom de domini concret, o com el servei de **sincronització horària** que aconsegueix mantenir en hora els dispositius que ho sol·licitin. D'altres serveis permeten que diferents usuaris intercanviïn informació, com ara els serveis de **correu electrònic**, els **fòrums** o els serveis de **xat**. Internet també disposa de serveis per obtenir recursos emmagatzemats en línia. Ens referim als serveis **d'aplicacions web** o als serveis **FTP** que permeten gestionar l'intercanvi de fitxers. Existeixen també serveis prou oberts com ara **telnet** sense funcionalitat específica definida. De fet el servei emula una consola de text a través de la qual el client pot rebre missatges i enviar comandes que seran executades remotament en el servidor.

Els serveis d'aplicacions web utilitzen tecnologies específiques, com ara el protocol de comunicació HTTP, el llenguatge HTML, etc.

Actualment telnet està en desús en favor dels serveis d'aplicacions web, perquè disposen de bones interfícies gràfiques, d'un temps de resposta eficient i de la possibilitat de realitzar crides realment complexes, cosa que els ha permès transformar-se en l'eina base en la qual fer créixer altres serveis de nivell més alt que puguin adaptar-se a les limitacions que la tecnologia implicada imposa.

Avui trobem molts serveis basats en les aplicacions web, perquè així s'asseguren l'accés de qualsevol client que disposi d'un navegador. Tots els cercadors ofereixen serveis d'indexació i cerca basats en web. De forma similar molts llocs i plataformes web enriqueixen la seva potencialitat amb xats o fòrums basats també en aquestes tecnologies. Les tecnologies web són tan versàtils que darrerament

s'està popularitzant la implementació de serveis que fan servir HTTP com a protocol de transferència i els llenguatges associats a la tecnologia web, com ara XML, per donar estructura a la informació transmesa.

2.1 Serveis i protocols estàndards

D'entre la gran quantitat de serveis d'Internet destacarem, per la seva importància, 3 serveis que sens dubte han marcat i encara marquen l'evolució de la xarxa de xarxes. Per ordre d'importància són el servei WWW, els serveis de correu electrònic i els serveis de transferència de fitxers. La importància esdevé sobretot per la utilització que se'ls dóna en el desenvolupament d'aplicacions més grans, ja sigui com a complements o com a suport base de la seva implementació.

Podeu trobar una llista de tots els documents RFC existents a http://www.ietf.org/download/rfc-index.txt

La documentació que descriu els serveis estàndards d'Internet és aprovada i publicada per un organisme anomenat IETF (Internet Engineering Task Force) format per voluntaris que discuteixen i regulen totes les regles i missatges utilitzats a Internet. La documentació estàndard s'identifica mitjançant les sigles RFC (*Request For Comments*) seguides d'un valor numèric únic per a cada especificació. Per exemple, el format dels missatges de correu electrònic es defineixen al document anomenat RFC-2822, mentre que el protocol que regula la consulta i gestió dels missatges arribats a les bústies dels destinataris és el RFC-1939.

Només **el compliment de la documentació estàndard** assegura la compatibilitat entre clients i servidors i la plena interoperativitat de la xarxa.

2.2 Servei de transferència de fitxers

Aquest servei, tal com indica el seu nom, permet gestionar la transferència de fitxers entre dos llocs situats en diferents dispositius. El protocol estàndard utilitzat aquí és l'anomenat FTP (File Transfer Protocol) que es troba especificat al document RFC-959. El protocol corre sobre TCP. Es basa en un model client-servidor i contempla un conjunt de comandes que permetran al client gestionar un sistema de fitxers remot. El protocol contempla comandes a validar i autenticar, per demanar informació referent al sistema de fitxers controlat pel servidor, per fer peticions de modificació del sistema de fitxers remot o per iniciar una transferència d'un fitxer en qualsevol de les dues direccions possibles. L'especificació descriu també els paràmetres obligatoris i opcionals de les comandes, la seva funcionalitat i les possibles respostes que es poden esperar dels servidors.

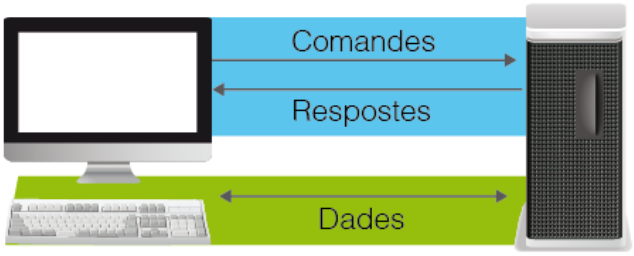
En general podem dir que se segueix el patró *petició-resposta* de la majoria d'aplicacions **client-servidor**. En aquest cas les peticions són les comandes que el client envia i les respostes les genera el servidor associant un codi i un missatge

que informa de si la comanda ha tingut o no èxit.

Cal tenir en compte, però, que algunes comandes, a més de la resposta, requeriran un traspàs extra d'informació que pot arribar a ser força considerable (imagineu la transferència d'un fitxer gran). També cal considerar que les comandes i les respostes associades s'envien sempre en forma de cadenes de caràcters, però les dades extres procedents de fitxers poden tenir una gran varietat de formats. Finalment cal tenir en compte que els tractaments usats per enviar gran quantitat d'informació poden resultar poc eficients en aquells casos que sigui necessari fer enviaments curts de forma puntual.

Són arguments amb els quals el protocol justifica la raó per la qual FTP estableix l'ús de canals de comunicació diferents pel tractament de les comandes pròpies del protocol i per la transferència d'informació extra derivada de la resolució d'algunes d'aquestes comandes (vegeu la figura 2.2).

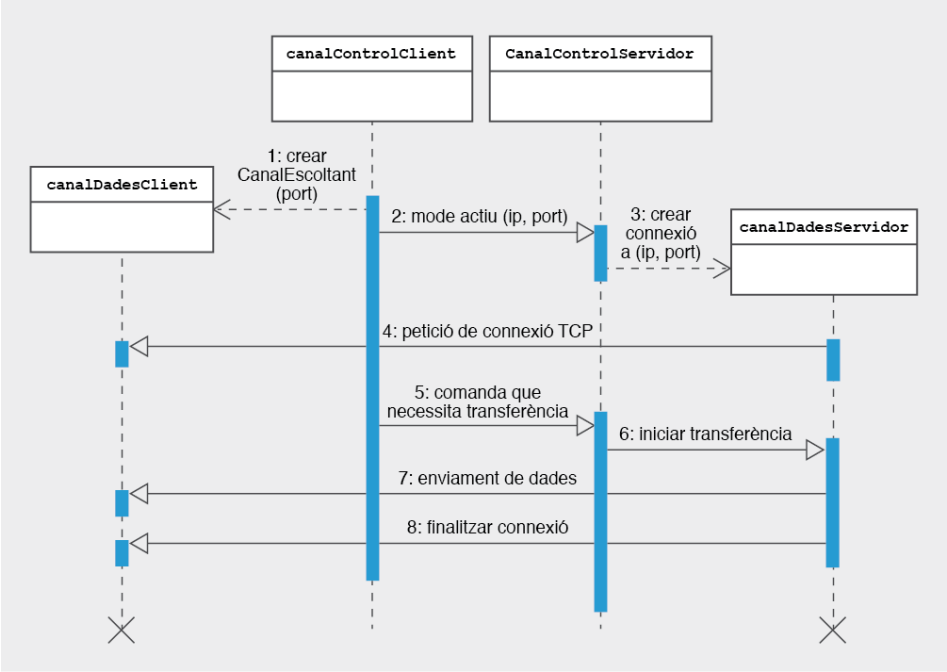
Figura 2.2. Esquema dels canals usats en un servei FTP



El canal usat pel diàleg comandes-respostes s'anomena també canal de control. Es tracta d'una connexió que romandrà oberta mentre el client es trobi connectat. Per contra el canal de dades és temporal, només es crearà quan sigui necessari iniciar una transferència i es tancarà automàticament en el moment en què totes les dades hagin estat transmeses.

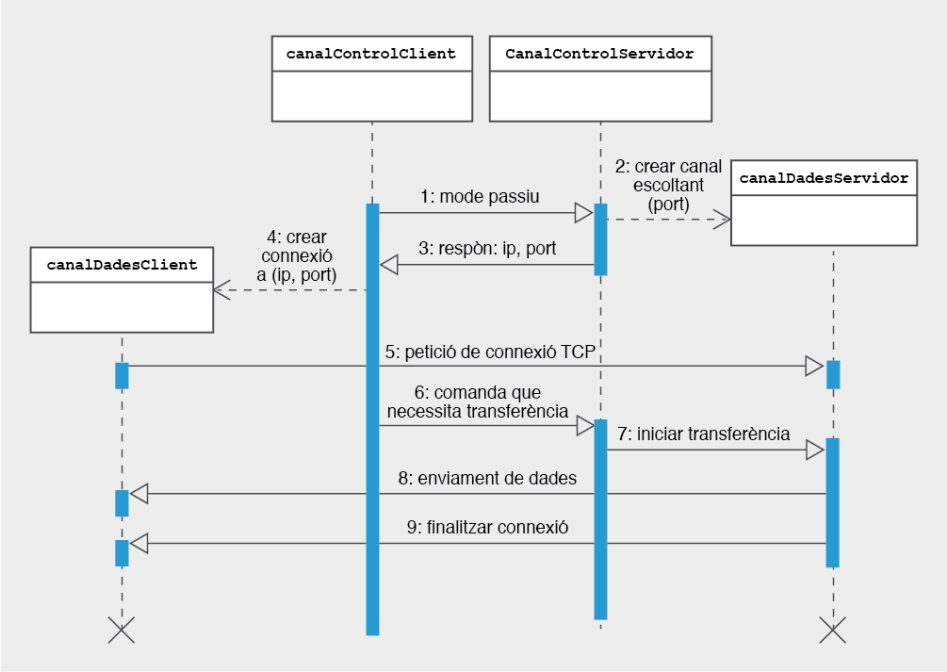
El servidor acostuma a fer servir el port 21 pel canal de control. Pel canal de dades, en canvi, el protocol dóna la possibilitat de treballar en dos possibles modes que anomenen respectivament mode actiu i mode passiu. El mode actiu preveu que el client indiqui al servidor l'adreça IP i el port en el qual desitja iniciar la transferència i sigui el servidor l'encarregat de fer la connexió (vegeu la figura 2.3). En canvi el mode passiu implica que el client sol·licita al servidor les dades de connexió, amb la resposta del servidor serà el client l'encarregat de fer la connexió (vegeu la figura 2.4).

Figura 2.3. Seqüència que mostra com es gestiona la petició d'una connexió activa



En aquest esquema no es fan servir les comandes FTP. Per raons didàctiques s'han substituït per frases de text amb sentit

Figura 2.4. Seqüència que mostra com es gestiona la petició d'una connexió passiva



En aquest esquema no es fan servir les comandes FTP. Per raons didàctiques s'han substituït per frases de text amb sentit

És important entendre la seqüència de cada mode si hem d'implementar el servei ja que el bon funcionament dependrà de la correcta coordinació entre client i servidor. Fixeu-vos que el client, abans de realitzar una comanda que requereixi transferència de dades, indicarà al servidor el mode amb el qual vol treballar. Només un cop finalitzada la petició del mode i amb el canal de dades creat, el client demanarà l'execució de la comanda. En cas de fer la petició d'una comanda que requereixi transferència sense demanar el mode (passiu o actiu) el servidor enviarà una resposta d'error amb el codi 425.

L'ús de connexions actives i passives permet gestionar la còpia entre dos servidors des d'un client. Només caldrà que el client demani un connexió passiva a un dels servidors. Això li farà arribar al client la IP i el port que el servidor estarà preparant per a la connexió. Amb la IP i el port rebut indicarà al segon servidor que desitja un connexió activa, la qual cosa farà que el segon servidor es connecti al primer. Finalment el client enviarà una comanda a cada servidor per indicar a l'un que ha d'iniciar la transferència del fitxer desitjat i a l'altre que ha d'emmagatzemar les dades rebudes.

Les comandes FTP són cadenes de text que simbolitzen el nom d'una operació i poden anar seguides dels paràmetres que la comanda necessiti. Els paràmetres se separen amb comes. La comanda es finalitza i s'envia amb un final de línia per tal que el servidor detecti el final de la petició. Les respostes segueixen el mateix format de text. El servidor envia una línia de text en la qual la primera paraula serà un codi numèric de 3 xifres seguit d'una cadena informativa.

Així per exemple, en sol·licitar la connexió inicial al servei del port 21, el servidor respondrà amb el codi 220 indicant que cal que ens autentiquem. FTP disposa de les comande *user* <nom_suauri> i *pass* <contrasenya> per poder realitzar l'autenticació. El codi 230 ens informarà que la validació del nom i contrasenya de l'usuari ha tingut èxit o, en cas contrari, arribarà el valor 530.

Els servidors FTP solen donar accés a una zona limitada del sistema de fitxers i acostumen a treballar amb els permisos propis del sistema operatiu, executant les comandes com l'usuari autenticat. Tot i així, el servidor pot mantenir també un sistema de permisos propi i decidir el que cada usuari pot fer o no.

Les comandes FTP ens ajuden a moure'ns pel sistema de fitxers remot, obtenir informació, transferir fitxers d'una banda a l'altre, eliminar-los, reanomenar-los, crear nous directoris, etc. A la taula taula 2.1 trobareu una relació de les principals comandes.

Taula 2.1. Taula de les principals comandes FTP

Comanda	Paràmetres	Acció
PWD	-	Mostra el nom del directori actual
CWD	nom_directori	Canvia al directori indicat en el paràmetre
CDUP	-	Canvia al directori pare
PORT	bytte_1_IP, byte_2_IP, byte_3_IP, byte_4_IP, byte_1_port, byte_2_port	Demana iniciar una transferència de dades en mode actiu a la IP i port indicats. Tant la IP com el port s'expressen com a cadenes de bytes
PASV	-	Demana iniciar una transferència de dades en mode passiu. Si tot va bé, es rebrà una resposta amb el codi 227 que indicarà la IP i port de connexió seguint el format (n1, n2, n3, n4, n5, n6) i que indicarà els 4 bytes de la IP i el dos del port
NLST	-	Envia pel canal de dades la llista de noms dels fitxers continguts al directori actual

Taula 2.1 (continuació)

Comanda	Paràmetres	Acció
LIST	-	Envia pel canal de dades la llista de fitxers continguts al directori actual i informa dels seus atributs
STOR	nom_fitxer	Emmagatzemarà les dades rebudes a través del canal de dades en un fitxer que anomenarà com indica el paràmetre
STOU	-	Emmagatzemarà les dades rebudes a través del canal de dades en un fitxer que anomenarà automàticament de forma única
APPE	nom_fitxer	Emmagatzemarà les dades rebudes al final d'un fitxer existent anomenat com indica el paràmetre
RETR	nom fitxer remot	Envia pel canal de dades el fitxer remot indicat per paràmetre
ABOR	-	Interromp la transferència en curs
DELE	nom_fitxer	Elimina el fitxer indicat pel paràmetre
MKD	nom_directori	Crea un nou directori situat al directori actual amb el nom del paràmetre
QUIT	-	Es demana abandonar el servei i tancar la connexió

2.3 Serveis de correu electrònic

Quan parlem de serveis de correu electrònic, bàsicament ens referim a dos serveis diferents que col·laboren per aconseguir transmetre un missatge digital des del dispositiu de l'autor (remitent) al dispositiu del destinatari. Autors i destinataris són particulars que no mantenen el seu dispositiu permanentment obert i connectat, cosa que podria impedir la recepció si el dispositiu destinatari es trobés desconnectat en el moment de la transmissió. La solució passa per emmagatzemar els missatges en dispositius als quals s'hi pugui accedir des del destinatari per acabar de realitzar la transmissió. Fent un símil amb la vida real, els dispositius d'emmagatzematge s'anomenen bústies de correu electrònic. Cada destinatari ha de tenir una bústia diferenciada en la qual trobar només els seus missatges.

La transmissió efectiva dels missatges electrònics s'aconseguirà doncs realitzant una transmissió des del dispositiu de l'autor fins a la bústia del destinatari, en una primera fase, i des de la bústia fins al dispositiu del destinatari en una segona. La primera fase dóna lloc al servei de transferència de missatges. La segona al servei d'accés a les bústies i recepció del contingut.

El format del missatge juga un paper fonamental en l'automatització d'ambdós serveis. Es troba especificat al document RFC-2822, en el qual s'indica que

els missatges de correu electrònic es composaran de dues parts, la capçalera de caràcter obligatori i el cos del missatge de caràcter opcional.

2.3.1 Format del missatge de correu electrònic

La **capçalera** conté la informació que ajudarà a automatitzar la transmissió. De fet, seria equivalent a la informació que en una carta postal se situa en el sobre, nom i adreça del destinatari, nom i adreça del remitent, etc. En la versió electrònica, la informació es troba estructurada en un conjunt de camps específics, organitzats com a parelles nom-valor.

El **cos** del missatge conté només la informació que el remitent vol fer arribar al destinatari. Seria doncs l'equivalent a l'escrit que trobem dins del sobre en una carta postal.

A la taula 2.2 trobareu els principals camps de capçalera definits a l'RFC-2822.

Taula 2.2. Principals camp de capçalera dels missatges de correu electrònic

Nom	Funció
Date	Informa de la data en la qual s'ha enviat el missatge. La data presenta la forma: DD MMM AAAA hh:mm:ss <latex>\pm</latex>hhmm on DD són els dos dígits del dia del mes, MMM l'abreviació de tres lletres del nom del mes en anglès, AAAA, els 4 dígits de l'any, hh:mm:ss l'hora i <latex>\pm</latex>hhmm, 4 dígits precedits d'un signe més o un signe menys indicant la diferència horària respecte l'hora UTC (hh són hores i mm, minuts)
From	Indica l'adreça del remitent i opcionalment també el seu nom. Si s'indica el nom, l'adreça es delimitarà amb els símbols "menor que" < marcant l'inici i "major que" > el final
To	Informa de l'adreça del destinatari (o adreces si n'hi ha més d'una) i opcionalment també del seu nom. Segueix el mateix format que el camp *From*. Si hi ha més d'un destinatari se separaran usant el símbol coma ","
Cc	Especifica nom i adreça de destinataris als quals se li ha d'enviar una còpia del missatge. A la pràctica l'enviament és idèntic al dels destinataris, però permet informar als usuaris del servei que aquests no són els destinataris. És un camp opcional
Bcc	Aporta la mateixa informació que Cc, però el valor d'aquest camp mai es farà arribar als usuaris. Per això aquest camp s'anomena còpia oculta (*blind carbon copy* en anglès). És tracta d'un camp opcional
Subject	És un text explicatiu que resumeix el contingut del missatge. És opcional i té per objectiu informar al destinatari del que es trobarà quan llegeixi el contingut del missatge

Taula 2.2 (continuació)

Nom	Funció
Reply-to	Permet indicar l'adreça a la qual cal contestar el correu en cas que no coincideixi amb l'adreça expressada al camp *From*

Entenem per **espai en blanc** qualsevol caràcter que representin un o més espais tipogràfics de comportament horitzontal o vertical. Així, són espais en blanc l'espai, el tabulador, el salt de línia, etc.

Els camps de la capçalera separen les parelles nom-valor intercalant el caràcter dos punts ":". El nom del camp no pot contenir espais en blanc, ni abans, ni després. Els valors, en canvi, sí que poden contenir espais en blanc en qualsevol posició. En general cada camp ocuparà una única línia, però si convé podem fer que ocupin diverses línies posant un espai després de cada salt de línia. Un salt de línia sense cap espai, significarà que comença un nou camp de capçalera.

Exemples de camps de capçalera

El següent camp:

```
Date: 29 Apr 2014 19:30:09 +0200
From: your@domain.org
```

és equivalent a:

```
Date: 29 Apr 2014
  19:30:09 +0200
From: your@
   domain.org
```

El cos del missatge se separarà de la capçalera afegint una línia de text buida. Els missatges de correu electrònic fan servir com a salts de línia la combinació de dos caràcter especials CR (de codi ASCII 10) i CF (codi ASCII 13), per tant entre el darrer camp de la capçalera i l'inici del missatge trobarem sempre 4 bytes corresponents a la seqüència CR-LF-CR-LF.

Dins del missatge poden haver-hi tants salts de línia seguits com sigui necessari. Només el primer parell de salts separen la capçalera del missatge. Veieu l'exemple:

Exemples de missatge de correu electrònic

```
Date: 10 Jun 2013 18:01:23 +0200
From: professor@ioc.cat
To: El Teu Nom <your@domain.org>, Més Gent <another@domain.org>
Subject: Exemple que il·lustra el format d'un correu electrònic
Cc: El Cap <boss@domain.org>
Bcc: El Cap Major <big_boss@domain.org>

Aquest és un exemple en el qual es pot veure el format en text
    pla d'un correu
electrònic, tal com s'envia per la xarxa.

A dalt podeu veure la capçalera i aquí teniu el cos, separat de
    la capçalera
amb un salt de línia.

Cordialment
El Professor
```

2.3.2 Adjunció de fitxers i format HTML en el cos del missatge

L'enviament de missatges de text presenta moltes limitacions. Sovint tenim dades a enviar que es troben en altres fitxers, en formats diversos o volem expressar èmfasi al nostre escrit afegint recursos estilístics que l'ajudin a llegir i interpretar. Aquesta limitació, va forçar a crear un sistema independent capaç de suportar qualsevol dada en qualsevol format. El resultat es va materialitzar en l'estàndard que es coneix com a MIME (*Multipurpose Internet Mail Extension*). Aquest estàndard identifica i anomena tots els tipus de formats existents, per tal que puguin identificar-se sense confusió usant el nom.

Podeu trobar una llista dels tipus MIME a wikipedia.

L'especificació de MIME no modifica l'estàndard RFC-2822 sinó que l'estén. Bàsicament afegeix nous camps de capçalera que ajudaran a identificar el tipus de contingut, el joc de caràcters usat quan es tracti de text, el tipus de codificació utilitzada per enviar la informació quan aquesta s'enviï codificada, etc. Així, quan el client de correu rebi un missatge amb una camp a la capçalera especificant "Content-Type: text/html", ja sabrà que el cos del missatge es trobarà formatat usant HTML i podrà fer-ne una interpretació correcta.

L'extensió MIME, a més, ofereix una forma de poder enviar informació de diversos tipus dins d'un mateix missatge. Es tracta de la mateixa fórmula que permet l'adjunció de fitxers dins un missatge. MIME preveu un tipus de contingut especial anomenat *multipart*, compost de blocs d'informació de diferents tipus (els fitxers a adjuntar), els quals es concatenaran l'un darrere l'altre formant el cos del missatge. Cada bloc estarà format per una capçalera de bloc en la qual es descriurà el tipus de contingut i l'altra informació necessària per interpretar el bloc. Entre bloc i bloc, MIME preveu intercalar una cadena de text única que faci de separador i que s'anomena frontera o *boundary*. Els blocs es trobaran delimitats per davant i per darrera per dos guionets "–" seguits de la cadena frontera. La darrera cadena frontera haurà de finalitzar també amb dos guionets (a més dels dos que la precediran) per tal de reconèixer el final del cos.

La cadena frontera pot ser qualsevol valor. Cal assegurar, això sí, que es tracta d'un valor únic i que no serà possible trobar la mateixa seqüència repetida entre les dades. Per tal que el servidor sàpiga quina cadena farem servir com a frontera, inclourem el valor de la mateixa com a extensió del camp Content-Type, sota el nom de `boundary`.

Exemples de missatge de correu electrònic amb fitxers adjunts

```
Date: 10 Jun 2013 18:01:23 +0200
From: professor@ioc.cat
To: El Teu Nom <your@domain.org>, Més Gent <another@domain.org>
Subject: Exemple que il·lustra el format d'un correu electrònic
    amb fitxers adjunts
Cc: El Cap <boss@domain.org>
Bcc: El Cap Major <big_boss@domain.org>
Content-Type: multipart/mixed; boundary=_123_abc_987_zyx_

--_123_abc_987_zyx_
Content-Type: text/html
Aquest és un exemple en el qual es pot veure el <em>format d'un
```

```
    correu
electrònic</em> que segueix l'especificació MIME per donar format
     d'estil al
seu missatge i per adjuntar fitxers independents.

El tipus del missatge ha de ser <i>multipart/mixed</i>. Després
    cada bloc
contindrà una capçalera pròpia en la qual s'indicarà el tipus de
    cada fitxer i també el del missatge.

Cordialment
El Professor
——_123_abc_987_zyx_
Content-Type: image/gif; name="ioc_logo.gif"
Content-Transfer-Encoding: base64

PGh0bWw+CiAgPGhlYWQ+CiAgPC9oZWFkPgogIDxib2R5PgogICAgPHA+
    VGhpcyBpcyB0aGUg
Ym9keSBvZiB0aGUgbWVzc2FnZS48L3A+CiAgPC9ib2R5Pgo8L2h0bWw+Cg==
——_123_abc_987_zyx_——
```

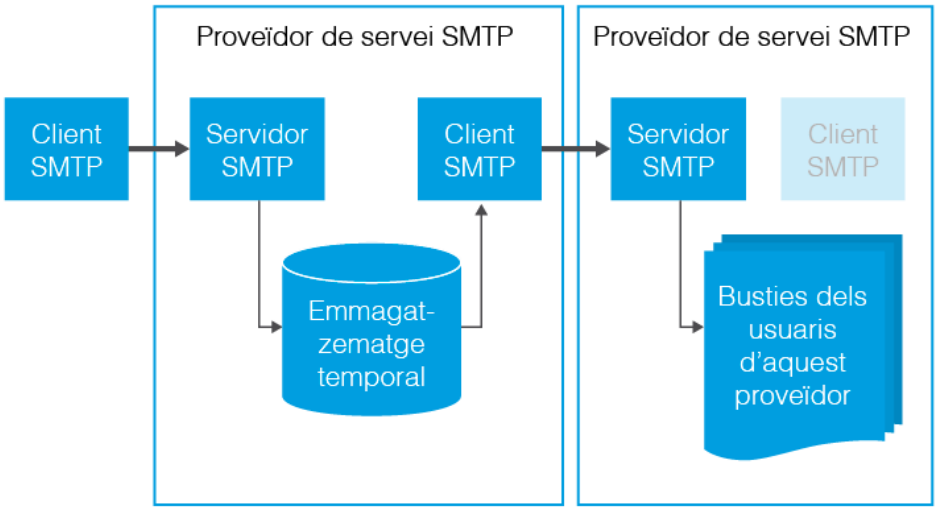
Per tal d'assegurar la compatibilitat de la transmissió, els fitxers que no siguin de caràcter, normalment es codifiquen usant el mecanisme anomenat **base64** que transforma la informació binària en seqüències de 7 bits (codi ASCII).

2.3.3 Servei de transmissió de missatges

És el servei encarregat de transmetre els missatges de correu electrònic. Fa servir el protocol anomenat SMTP (*Simple Mail Transfer Protocol*), especificat en el document estàndard RFC-2821. El port assignat a aquest servei sol ser el 25. Generalment, cada usuari d'Internet fa servir un únic servidor SMTP que sol coincidir amb el proveïdor que li subministra la bústia de correu electrònic.

Per dur a terme la transmissió d'un missatge entren en joc 3 actors diferents, l'usuari remitent, un servidor de correus (habitualment el del proveïdor del remitent) i el proveïdor de correus del destinatari. El remitent usarà un emissor SMPT (inclòs en el client de correus local) per enviar el missatge al servidor SMTP del seu proveïdor (receptor). En ser rebut, serà reenviat als proveïdors de correus als quals estiguin subscrits els destinataris. El servidor del destinatari es reconeix analitzant la part dreta de la seva adreça de correu, la que es troba a continuació del símbol @. Aquesta part de l'adreça es pot resoldre contra un servidor DNS especificant que es tracta de registres MX. Quan els servidors SMPT reben correus adreçats als seus abonats analitzen la part esquerra de l'adreça, la qual els indicarà el nom de la bústia en la qual caldrà emmagatzemar-los. Mireu l'esquema de la figura 2.5.

Figura 2.5. Esquema que mostra com actuen els diferents actors durant la transmissió d'un missatge de correu electrònic



La connexió entre client i servidor es realitza també a través de comandes de text acabades amb un salt de línia. Cada comanda es contesta amb una resposta composta d'un codi de 3 xifres i una frase informativa que acabarà també amb un salt de línia. Quan la resposta a una comanda ocupi més d'una línia caldrà incloure un guió entre el codi i la frase excepte en la darrera línia que hi apareixerà un espai, tal com es mostra al patró que segueix:

```
CODI-frase1
CODI-frase2
CODI frase3
```

A la taula 2.3 trobareu les principals comandes dels protocol SMTP.

Taula 2.3. Principals comandes especificades al protocol SMTP

Comanda	Paràmetres	Descripció
HELO	Nom domini	Cal usar-la sempre a l'inici de la connexió i informa al servidor del domini des del qual el client s'està connectant
MAIL FROM	Adreça del remitent	Informa al servidor de l'adreça del que envia el missatge
DATA	Missatge de correu electrònic (capçaleres i cos) a enviar	Per finalitzar el missatge caldrà escriure la seqüència formada per un salt de línia un punt i un altre salt de línia
QUIT	-	Indica al servidor que l'emissor vol tancar la connexió

Exemple que il·lustra un connexió entre un client i un servidor SMTP

Comença el servidor

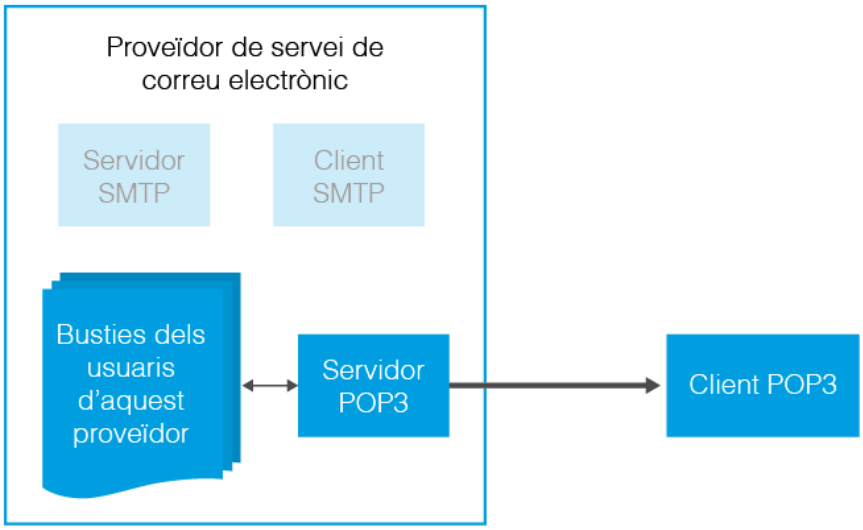
```
220 smtp.xtec.cat SMTP server ready
HELO ioc.xtec.cat
250 Hello, please to meet you
MAIL FROM: <ioc@xtec.cat>
```

```
5  250 Ok
6  RCPT TO: <destinatari1@undomini.org>
7  250 Ok
8  RCPT TO: <destinatari2@unaltredomini.org>
9  250 Ok
10 DATA
11 354 End data with <CR><LF>.<CR><LF>
12 Date: 10 Jun 2013 18:01:23 +0200
13 From: professor@ioc.cat
14 To: El Teu Nom <your@domain.org>, Més Gent <another@domain.org>
15 Subject: Exemple que il·lustra el format d'un correu electrònic
16 Cc: El Cap <boss@domain.org>
17 Bcc: El Cap Major <big_boss@domain.org>
18
19 Aquest és un exemple en el qual es pot veure el format en text
       pla d'un correu
20 electrònic, tal com s'envia per la xarxa.
21
22 A dalt podeu veure la capçalera i aquí teniu el cos, separat de
       la capçalera
23 amb un salt de línia.
24
25 Cordialment,
26 El Professor
27
28 .
29 250 Ok: Mail accepted
30 quit
31 221 Bye
```

2.3.4 Servei d'accés a les bústies d'usuari

Aquest servei s'encarrega només de gestionar l'accés a les bústies i la recepció dels correus emmagatzemats. Actualment molts proveïdors ofereixen accés web a través d'una aplicació que retorna les dades en una pàgina HTML i que no necessita que el servidor es descarregui cada correu. Tot i així continuen vigents com a alternativa les aplicacions d'accés a les bústies amb l'objectiu de fer una descàrrega local. El protocol més popular d'aquestes característiques és l'anomenat POP3. En aquest servei només intervenen l'usuari abonat i el proveïdor del servei POP3. Mireu l'esquema de la figura 2.6.

Figura 2.6. Esquema que mostra com actuen els diferents actors durant l'accés a les bústies de correu electrònic



Aquest servei preveu sempre l'autenticació en realitzar la connexió, ja que cada usuari ha de poder accedir només a la seva bústia. L'autenticació es realitzarà amb les comandes `USER` i `PASS`. Un cop l'usuari aconsegueixi autenticar-se, podrà consultar el nombre de missatges que té a la bústia. El servei POP3 no sap diferenciar entre un missatge llegit i un no llegit, per això aquesta informació ha de córrer a càrrec del client que mantindrà una referència per poder fer-ne la distinció. Tot i així, val a dir que POP3 no està pensat per mantenir a les bústies un gran volum de missatges i per tant és convenient que l'usuari es descarregui el correu en local i l'esborri del servidor.

L'usuari podrà també demanar una llista dels missatges de la bústia usant la comanda `LIST`. Això retornarà un identificador numèric per a cada missatge juntament amb el nombre de bytes que ocupen. La comanda `TOP` permetrà rebre la capçalera d'un missatge i el nombre de línies especificat per paràmetre i la comanda `RETR` permetrà recuperar tot el missatge sencer. A més, amb la comanda `DELE` aconseguirem eliminar el missatge indicat de la bústia.

Les respostes del servidor començaran sempre per la seqüència `+OK` si la comanda s'ha executat amb èxit o `-ERR` si hi ha hagut algun error. Generalment les respostes contindran també frases informatives per l'usuari.

Exemple d'un diàleg entre client i servidor de POP3

```
+OK POP3 server ready.
USER alumne
+OK password required. Send PASS command.
PASS 1234
+OK alumne you has 3 messages
LIST
+OK 3 messages
1 513
2 1825
3 1019
.
TOP 1 0
```

```
13 +OK Top of message (1) whith 0 lines of body
14 Date: 10 Jun 2013 18:01:23 +0200
15 From: professor@ioc.cat
16 To: El Teu Nom <your@domain.org>, Més Gent <another@domain.org>
17 Subject: Exemple que il·lustra el format d'un correu electrònic
18 Cc: El Cap <boss@domain.org>
19 Bcc: El Cap Major <big_boss@domain.org>
20 .
21 RETR 1
22 +OK 1801 octets
23 Date: 10 Jun 2013 18:01:23 +0200
24 From: professor@ioc.cat
25 To: El Teu Nom <your@domain.org>, Més Gent <another@domain.org>
26 Subject: Exemple que il·lustra el format d'un correu electrònic
27 Cc: El Cap <boss@domain.org>
28 Bcc: El Cap Major <big_boss@domain.org>
29
30 Aquest és un exemple en el qual es pot veure el format en text
     pla d'un correu
31 electrònic, tal com s'envia per la xarxa.
32
33 A dalt podeu veure la capçalera i aquí teniu el cos, separat de
     la capçalera
34 amb un salt de línia.
35
36 Cordialment
37 El Professor
38 .
39 DELE 1
40 +OK message (1) deleted
41 QUIT
42 +OK bye alumne. POP3 server signing off.
```

2.3.5 Implementació del client usant la biblioteca JavaMail

> És probable que ja tingueu la versió J2EE activada, però si no fos així, podeu consultar l'annex `Instal·lació de J2EE 7` per saber com poder activar-la.

Per implementar clients de correu Java posa a la nostra disposició la biblioteca JavaMail. La biblioteca només es troba disponible si usem la plataforma J2EE que, tal com sabeu, és una extensió del JDK. JavaMail és una biblioteca que treballa sobre la majoria de protocols, ja sigui de transmissió o d'accés a les bústies. Per descomptat reconeix els protocols SMTP i POP3 i aconsegueix una programació força neta en ambdues funcionalitats.

L'estructura de la biblioteca és de molt alt nivell i amb això aconsegueix independitzar la programació del protocol realment utilitzat. La classe `Session` contribueix a aquesta abstracció. Es tracta de la classe que dóna accés a un dels protocols disponibles i representa una sessió de correu oberta contra un proveïdor dels serveis de correu. La classe `Session` es configura a través de `Properties` ja que resulta una forma molt versàtil d'associar atributs i valors que es pot adaptar a gairebé qualsevol necessitat.

Exemple de configuració d'una sessió

Disposeu de la relació de propietats acceptades a: http://connector.sourceforge.net/doc-files/Properties.html

```
1         Properties props = new Properties();
2
```

```
3         props.put("mail.smtp.host", hostName);
4         props.put("mail.smtp.port", 459);
5         //Crea una instància que emmagatzema les dades de
              configuració
6         Session session = Session.getInstance(props);
```

Si ens cal realitzar transferència de correus, `Session` treballarà amb una instància de la classe `Transport`. Aquesta és una classe abstracta materialitzada per una classe que s'adapta a un protocol concret. Per defecte, la biblioteca JavaMail suporta `SMTPTransport`, però el fet de treballar amb la classe abstracta permet en un futur incorporar noves classes. És important, doncs, que totes les instàncies que calgui crear estiguin declarades com a `Transport` si volem construir una aplicació escalable i de llarga durada.

Les classes de la jerarquia `Transport` bàsicament tenen la funcionalitat d'enviar missatges seguint el protocol que suportin i fent servir una instància oberta de `Session`. Per evitar crear instàncies temporals que ens puguin carregar la memòria, `Transport` té el mètode *static* `send` per aconseguir realitzar l'enviament sense haver de crear cap instància. Aquest mètode obre i tanca una connexió cada cop que l'invoquem.

Exemple d'enviament d'un missatge usant el mètode estàtic anomenat send

El mètode `send` accepta diversos paràmetres. Per defecte enviarà el missatge passant en el primer paràmetre les adreces incloses en el missatge, però també es pot especificar les adreces com a segon paràmetre i en aquest cas no es consultarà el missatge.

```
1     InternetAddress[] addresses = {new InternetAddress("
          unaAdreca@xtec.cat"),
2                                    new InternetAddress("
                                         dosAdreces@xtec.cat")};
3
4     ...
5
6     Transport.send(emailMessage1);
7
8     Transport.send(emailMessage2, addresses);
```

L'objecte de l'enviament, que haurem d'haver construït prèviament, és una instància de la classe `Message`, que és estesa per la classe `MimeMessage`. La classe `Message` és una classe abstracta que no podem instanciar. Usarem sempre la classe `MimeMessage` per instanciar l'objecte. `MimeMessage` suporta l'especificació MIME. És a dir, podem enviar missatges de contingut divers i adjuntar fitxers, d'acord amb aquesta especificació.

Aquesta classe disposa dels mètodes adequats per crear un missatge compatible amb l'especificació RFC-2822. Per exemple, disposa del mètode `setFrom` per afegir el remitent, dels mètodes `setRecipient`, `setRecipients` o `addRecipient` per afegir destinataris de qualsevol dels camps que el format permet (To, Cc o Bcc). Per evitar problemes de sintaxi, les dades de les adreces de correu dels paràmetres dels mètodes esmentats seran del tipus `InternetAddress`, una classe que representa una adreça de correu electrònic però que analitza la cadena passada i llança excepcions quan detecta incorreccions en la cadena.

`Message` disposa també d'altres mètodes d'entrada de dades com ara `setSendDate`, `setSubject`, `setText` o `setReplyTo` per facilitar la composició.

Exemple de creació d'un missatge de correu simple

Per instanciar el missatge cal passar-li una sessió per paràmetre en el constructor.

```

    Message emailMessage = new MimeMessage(session);

    ...

    emailMessage.setFrom(new InternetAddress("lameva@ioc.cat"));
    InternetAddress[] address = {new InternetAddress("secretaria@ioc.cat" ),
                                 new InternetAddress("direcciofp@ioc.cat")};
    emailMessage.setRecipients(Message.RecipientType.TO, address);
    emailMessage.setRecipient(Message.RecipientType.CC,
                              new InternetAddress("secretaria@ioc.cat" ) );
    emailMessage.setSubject("Prova de construcció amb JavaMail");
    emailMessage.setSentDate(new Date());

    emailMessage.setText("Contingut del missatge en text pla i sense " +
                         "usar adjuncions.\n\nAtentament\nEl professor");
```

JavaMail també simplifica molt la creació de missatges MIME de tipus multipart, per exemple per adjuntar documents.

Exemple de creació d'un missatge de correu multipart

```

    Message emailMessage = new MimeMessage(session);

    ...

    emailMessage.setFrom(new InternetAddress("lameva@ioc.cat"));
    InternetAddress[] address = {new InternetAddress("secretaria@ioc.cat" ),
                                 new InternetAddress("direcciofp@ioc.cat")};
    emailMessage.setRecipients(Message.RecipientType.TO, address);
    emailMessage.setRecipient(Message.RecipientType.CC,
                              new InternetAddress("secretaria@ioc.cat" ) );
    emailMessage.setSubject("Prova de construcció amb JavaMail");
    emailMessage.setSentDate(new Date());

    //Part corresponent al cos del missatge
    MimeBodyPart p1 = new MimeBodyPart();
    p1.setText("Aquí <em>escrivim</em> el <i>cos del missatge</i> "
            + "usant UTF-8 i indicant el tipus de contingut",
            "utf-8", "text/html");


    // Adjuntem un fitxer en una segona part. Per a tipus coneguts, aquest
    // mètode afegeix de forma automàtica el content type i si cal codifica
    // en base64
    MimeBodyPart p2 = new MimeBodyPart();
    p2.attachFile("nomFitxer.jpg");

    // Adjuntem un fitxer en una tercera part. En aquesta adjunci
```

```
        ó especifiquem
    // el tipus i la codificació desitjada
    MimeBodyPart p3 = new MimeBodyPart();
    p2.attachFile("fitxerFormatPropi.meu", "application/octet-
        stream", "base64");

    // Ajuntem totes les parts.
    Multipart mp = new MimeMultipart();
    mp.addBodyPart(p1);
    mp.addBodyPart(p2);
    mp.addBodyPart(p3);

    // Assignem el contingut multipart creat
    emailMessage.setContent(mp);

    //Podem enviar el missatge
    Transport.send(emailMessage);
```

L'accés a la bústia de correu personal i la seva gestió s'aconsegueix treballant amb una instància d' `Store`. La classe `Store` seria l'equivalent a `Transport` però pel procés d'accés a les bústies. Igual que `Transport`, és una classe que es materialitzarà amb extensions que implementin un protocol específic. Així trobem `Pop3Store` i `IMAPStore`. De forma semblant al que fèiem amb `Transport`, caldrà configurar la sessió indicant almenys el nom del *host* i el port si el servidor no utilitzés el port 110.

La classe `Store` representa la bústia remota i gestionarà la connexió puntual invocant el mètode `connect`, al qual caldrà passar-li per paràmetre el nom d'usuari i contrasenya per autenticar-se en el servidor. A partir d'un objecte `Store` aconseguirem un objecte `Folder` que representa una carpeta de la bústia. Cal explicar que POP3 no permet gestionar carpetes però altres protocols com ara IMAP4 si que ho fan. JavaMail és una biblioteca genèrica que ha d'adaptar-se a qualsevol protocol. Per això els correus es recullen sempre en una carpeta (objecte `Folder`).

Treballant amb el protocol Pop3, caldrà recupera la capeta usant el nom *INBOX* i seguidament obrir-la en mode de lectura o en mode d'escriptura. El mode de lectura no permetrà realitzar cap alteració de la bústia. En canvi el mode d'escriptura ens deixarà eliminar remotament els missatges.

Obtenció i obertura d'un objecte Folder

```
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_WRITE);
```

La carpeta ens permetrà recuperar els missatges invocant `getMessages`. També podrem fer una recuperació parcial o eliminar aquells missatges que indiquem. El missatges arriben com a objectes `Message` i, per tant, el seu accés al contingut i capçaleres resultarà ràpid o fàcil.

2.4 Servei WWW

Aquest servei, que normalment es troba associat al port 80, ha esdevingut el més important dels tres en els darrers anys, ja que ha acabat convertint-se en el servei que sustenta la comunicació i presentació de les dades d'una gran quantitat d'aplicacions distribuïdes i fins i tot de nous serveis que se sobreposen al primer. La gran versatilitat d'aquest servei ha acabat desbancant totalment altres serveis, com ara Telnet, pensats específicament per donar suport a l'execució d'aplicacions remotes i de serveis de nivell superior.

HTML és l'acrònim de *Hipertext Makup Language* (Llenguatge de marcatge d'hipertetx) i HTTP el de *Hipertext Transfer Protocol* (Protocol de transferència d'hipertext)

L'objectiu originari del servei web era només facilitar la difusió de documents tècnics entre la comunitat científica. Per aconseguir-ho van idear un llenguatge de presentació del contingut, que van anomenar HTML i un protocol per formalitzar la localització, la petició i la transferència de documents, l'HTTP. Cada lloc d'Internet podia disposar d'un servei web i controlar un conjunt limitat de documents HTML. L'usuari podia adreçar-se als diferents documents indicant el nom del servei i el nom del document. És el que ha acabat derivant en el que es coneix com a URL o *Unified Resource Location*. És a dir, un sistema per identificar de manera unívoca la localització dels documents. El protocol HTTP definia la manera d'interaccionar amb els serveis WWW a l'hora de requerir l'obtenció d'un document localitzat per mitjà de la seva URL. El servei rebia les peticions de l'usuari, cercava el document que acabava enviant de nou al client seguint les indicacions del protocol HTTP. A la banda client, la rebuda del contingut es realitzava amb una aplicació especial capaç d'interpretar el llenguatge HTML i de presentar-lo adequadament a l'usuari.

El llenguatge HTML deu el seu nom a l'ús de vincles específics que permetien adreçar d'altres documents i van acabar per batejar-se amb el nom d'hipervincles. La facilitat de passar d'un document a un altre, sense necessitat de fer noves peticions, va propiciar que les aplicacions encarregades de la visualització dels document formatats usant el metallenguatge de l'hipertext s'acabessin denominant navegadors.

La meteòrica evolució de l'HTML ha propiciat el canvi d'orientació del servei que n'ha determinat l'èxit rotund. Cap a les darreries del segle XX van començar a desenvolupar-se servidors de pàgines HTML que es generaven dinàmicament. El funcionament era ben simple, s'aprofitava la infraestructura dels serveis web, afegint una aplicació extra capaç d'interpretar les peticions del protocol HTTP i generar a partir d'aquesta una pàgina inexistent amb dades extretes d'una base de dades i formatar-les d'acord amb el llenguatge HTML per tal de ser enviades igual com s'enviaria un document HTML existent.

A ulls de l'usuari final no hi havia diferència entre ambdós sistemes. La petició o la visualització es realitzava des del mateix navegador i la presentació de les dades en fer servir també el mateix llenguatge que els documents estàtics resultava molt similar. Els servidors dinàmics de pàgines web podien coexistir amb els serveis antics de les pàgines estàtiques.

Els llenguatges de programació script són llenguatges de programació interpretats que no requereixen compilació.

Aviat va arribar la revolució a la banda dels navegadors. Van independitzar la visualització de determinats estils i formats de text especificant un sistema de creació d'estils de format anomenat CSS i van enriquir la interpretació del llenguatge de marques HTML, amb un llenguatge de programació script que permetia processar dades, atendre esdeveniments i modificar dinàmicament la pròpia estructura HTML, des de la banda del client, sense necessitat de baixar ni generar noves pàgines remotes. La combinació de tots aquest elements (Javascript, CSS i HTML) han permès la creació d'interfícies gràfiques d'usuari cada cop més potents i ràpides. Això ha fet decantar la balança a favor d'aquest servei a l'hora de desenvolupar aplicacions remotes en detriment de Telnet que només suportava formats textuals.

2.4.1 El protocol HTTP

El protocol HTTP és l'estàndard usat per establir la comunicació entre clients i servidors d'un servei web. Es troba especificat en el document RFC-2616. Es tracta d'un protocol que pot funcionar tant sobre UDP com sobre TCP, malgrat que la majoria d'implementacions es troben realitzades sobre TCP perquè ofereix una major qualitat.

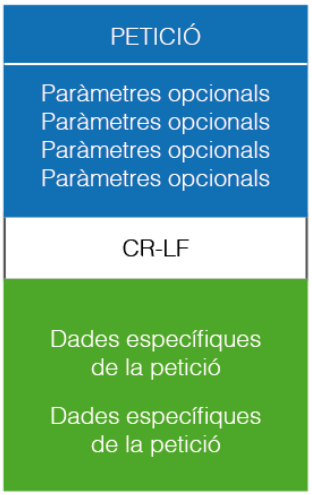
El protocol HTTP estableix que el final de línia es genera usant dos caràcters especials, el caràcter simbolitzat amb CR (*carriage return*) i el caràcter simbolitzat per LF (*line feed*). Respectivament responen als valors ASCII 10 i 13 i usant el llenguatge Java s'aconsegueixen també amb els literals '\r' i '\n'.

El protocol defineix que la comunicació s'estructura sempre en dues fases, la petició i la resposta. Els clients realitzen les peticions i els servidors els tornen la resposta. D'acord amb el protocol, les peticions dels clients han de ser independents entre sí, de manera que no hauria de ser necessari que el servidor hagi de recordar les peticions realitzades amb anterioritat per poder respondre'n una. Si el servidor al qual el client envia la petició manté aixecat el servei WWW, generarà sempre una resposta amb les dades demanades o amb el motiu pel qual no s'ha pogut satisfer la demanda.

Tant les peticions com les respostes són missatges de text estructurats en dos blocs i separats cada un d'ells per una salt de línia. El primer bloc s'anomena capçalera i el segon bloc cos. El cos del missatge és opcional, la capçalera és obligatòria. La mida de les capçaleres és variable. Per tant, cal enviar sempre els separadors CRLF per indicar al servidor que ja s'ha acabat la capçalera malgrat que el missatge no disposi de cos.

Les peticions HTTP contenen almenys una línia que explicita la petició del client, situada a la primera línia de capçalera del missatge. Opcionalment a la capçalera s'hi afegiran camps que podran matisar la resposta del servidor. El format del cos dependrà del tipus de petició. A la figura 2.7 podeu veure un diagrama que esquematitza aquesta estructura.

Figura 2.7. Esquema d'una petició HTTP



La petició pròpiament dita, situada a la primera línia, contindrà sempre 3 elements separats entre sí per espais i acabarà amb un final de línia (CR-LF). La primera paraula coincidirà amb la denominació d'un dels possibles mètodes acceptats pel protocol: `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `OPTIONS` o `TRACE`. Cal escriure la denominació dels mètodes en majúscula. El segon element indicarà la ruta absoluta en la qual es troba el recurs demanat a la petició dins el servidor. La ruta s'expressarà com una URL, a la qual s'eliminarà la denominació del *host* o domini. El tercer element indicarà el protocol i la versió utilitzada. Normalment usarem sempre HTTP/1.1.

Exemple de petició HTTP

GET /index.html HTTP/1.1

Fixeu-vos com la URL no conté el nom del domini ni el port. Per fer la petició del recurs www.xtec.cat/index.html caldria connectar-nos a través d'un sòcol al port 80 de l'adreça www.xtec.cat i un cop connectats enviar la petició de l'exemple.

Tipus MIME

L'especificació dels tipus MIME defineix una forma única de denominar cada un dels diferents tipus de recursos que es poden localitzar o enviar a través Internet. Podeu trobar una llista dels tipus MIME a wikipedia.

Els camps suportats a la capçalera són comuns a tot els mètodes. Cada camp ocupa una línia de la capçalera i cal especificar-los escrivint el nom del camp i el valor del mateix, separats ambdós pel caràcter dos punts `:`. Cada parella camp valor donarà al servidor informació per poder elaborar una resposta ajustada al client peticionari. Per exemple, el camp `Accept` permet indicar quins tipus de documents acceptarà el client. El contingut d'aquest camp ha de seguir l'especificació dels tipus MIME. Cada tipus acceptat anirà separat per un caràcter coma `,`. De forma semblant, el camp `Accept-Language`, permetrà informar de la llista de llenguatges acceptats pel client. D'aquesta manera, si el servidor disposa de recursos internacionalitzats, podrà escollir l'adequat. El camp `Host` informa del nom del servidor o domini al qual el client està adreçant la petició, el camp `User-Agent` informarà del tipus de

navegador que disposa el client, i amb el camp `Cookie` serà possible fer arribar al servidor les galetes (*cookies*) del client. Cal prestar atenció també als camps condicionals que permeten oferir una dada crítica que pugui fer decidir al servidor sobre si fer certa acció. Per exemple, el camp `if-Modified-Since` indicarà que només caldrà fer efectiva la petició si el recurs s'ha modificat posteriorment a la data indicada en el valor del camp.

Pel que fa a mètodes, els més usats són `GET` i `PUT`.

El **mètode** `GET` indica al servidor que el client demana la informació del recurs referenciada per la URL que es troba a continuació. És el mètode amb el que el navegador farà la petició quan escrivim directament l'adreça a la barra d'eines. El mètode `GET` no usa cos de missatge. Podeu provar aquesta sintaxi obrint una aplicació client de tipus Telnet, realitzant la connexió a un servei WWW d'algun servidor existent i seguidament, un cop connectat, fer algunes peticions per observar la resposta dels servidor. Recordeu que un cop introduïda la darrera línia de la capçalera cal confirmar amb un nou salt de línia per tal que el servidor sàpiga que no s'enviarà cap més dada i que ha de processar la petició rebuda.

Exemple de peticions GET fent servir un client Telnet

```
usuari:~$ telnet
telnet> open ioc.xtec.cat 80
Trying 213.176.163.218...
Connected to ioc.xtec.cat.
Escape character is '^]'.
GET /educacio/en/cicles-formatius HTTP/1.1
Host: ioc.xtec.cat
Accept-Language: CA-ES, en-us
```

Obtindrem una resposta semblant a:

```
HTTP/1.1 200 OK
Date: Wed, 17 Jul 2013 00:30:08 GMT
Server: Apache/2.2.22 (Unix) PHP/5.3.10 mod_ssl/2.2.22 OpenSSL
    /0.9.7d
...

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
    http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-gb" lang
    ="en-gb" dir="ltr" >
  <head>
    <base href="http://ioc.xtec.cat/educacio/en/cicles-formatius"
         />
          <meta http-equiv="content-type" content="text/html;
               charset=utf-8" />

    ...

 </body>
</html>
```

El **mètode** `POST` en canvi, serveix per enviar dades específiques al servidor en el cos del missatge. Aquest sol ser el mètode escollit per enviar les dades d'un formulari, per pujar un fitxer o per enviar qualsevol altra dada des del client al servidor. El mètode admet també els camps opcionals de la capçalera i el seu ús indica al servidor que haurà d'esperar dades al cos del missatge. Cal informar de la mida en bytes de les dades del cos fent servir el camp de la capçalera anomenat

`Content-Length` i del tipus de contingut usant el camp `Content-Type` amb un dels tipus especificats per la norma MIME. Això permetrà al servidor saber quants bytes li cal esperar en el cos, un cop passat el salt de línia de separació posterior a la capçalera. Si les dades del cos provenen d'un formulari, aniran aparellades de dos en dos. El primer element de la parella representarà el nom del paràmetre i el segon el seu valor. Ambdós elements aniran separats per un símbol = i entre paràmetre i paràmetre (parella de nom valor) s'intercalarà un salt de línia (CR-LF). El tipus de contingut que caldrà especificar és `application/x-www-form-urlencoded` i s'ha de tenir en compte les limitacions dels caràcters especials com ara l'espai que caldrà canviar pel seu valor numèric (expressat en hexadecimal) precedit del caràcter %.

Exemple de petició HTTP amb el mètode POST

```
1
2  POST /cgibin/escolx.pgm HTTP/1.1
3  Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;
       q=0.8
4  Accept-Language:ca,es;q=0.8,en-US;q=0.6,en;q=0.4,en-GB;q=0.2
5  Content-Length:36
6  Content-Type:application/x-www-form-urlencoded
7  Host:www.grec.cat
8
9  GECART=protocol
10 USUARI=dictioman
```

Cal explicar també que és possible enviar dades al servidor fent servir el mètode `GET`. L'especificació URL (*Uniform Resource Locator*) preveu afegir immediatament després de la ruta que identifiqui un recurs en una domini, el símbol ? seguit de la concatenació de les parelles de paràmetres (nom valor) separades per el símbol `&`. Per tant, usant el mètode `GET`, l'exemple anterior hauríem d'escriure'l:

Exemple de petició HTTP amb el mètode GET i enviament de dades

```
1  GET /cgibin/escolx.pgm?GECART=protocol&USUARI=dictioman HTTP/1.1
2  Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;
       q=0.8
3  Accept-Language:ca,es;q=0.8,en-US;q=0.6,en;q=0.4,en-GB;q=0.2
4  Content-Length:36
5  Content-Type:application/x-www-form-urlencoded
6  Host:www.grec.cat
```

Com a darrera característica del mètode `POST`, cal comentar que MIME preveu un tipus de dada especial anomenat `multipart` per poder concatenar gran quantitat d'informació de diversos tipus provinent de diferents fonts. La informació s'organitza en blocs i és possible especificar el tipus de cada bloc.

Trobareu més informació sobre el tipus MIME multipart al subapartat anomenat `Serveis de correu electrònic` de l'apartat `Generació de serveis en xarxa`.

Malgrat que no és exclusiu, aquest sistema sol fer-se servir per enviar gran quantitat de dades recollides des d'un formulari HTML o per enviar fitxers des del client al servidor. Els formularis HTML associen un valor a un paràmetre, però si es tracta d'una interfície per seleccionar un fitxer entraran en joc 3 elements: el nom del paràmetre, el nom del fitxer i el seu contingut. En aquests casos es preveu l'ús de del camp `Content-Disposition` amb el valor `form-data` i enriquit amb els complements `name` i opcionalment `filename`. El cos de cada bloc es destinarà al valor. Heu de tenir en compte que aquí no es barregen diferents tipus de contingut, de manera que si un valor és molt llarg i necessita definir-se com

a multipart, la resta de valors també es formataran de la mateixa manera. Un exemple ens ho clarificarà:

Exemple de petició HTTP amb el mètode POST usant el tipus MIME multipart

En el següent exemple s'envia al recurs `/copy/copier` tres fitxers. Un fitxer de text pla anomenat res.txt, una imatge anomenada logo.jpg que s'ha codificat usant l'algoritme Base64 que transforma qualsevol seqüència de bytes en una tira de caràcters alfanumèrics, i un HTML anomenat hola.html. A més s'envien les dades d'un camp simple anomenat *overwrite* que pren un valor lògic.

Observeu que els paràmetres que enriqueixen els camps de la capçalera són parelles de nom valor separades pel símbol = i entre elles pel símbol ;.

```

POST /copy/copier HTTP/1.1
Accept:text/html,application/xhtml+xml,application/xml
Accept-Language:ca,es, en-US, en, en-GB
Content-Type:multipart/mixed; boundary=78942734
    hdfkipiriepiryewi7979rb213cxzb
Host:www.securityCopy.com

--78942734hdfkipiriepiryewi7979rb213cxzb
Content-Disposition:form-data; name="file1"; filename="res.txt"
Content-Type: text/plain

Aquest és el contingut del fitxer res.txt
--78942734hdfkipiriepiryewi7979rb213cxzb
Content-Disposition:form-data; name="file2"; filename="logo.jpg"
Content-Type: image/jpg
Content-Transfer-Encoding: base64

PGh0bWw+CiAgPGhlYWQ+CiAgPC9oZWFkPgogIDxib2R5PgogICAgPHA+
    VGhpcyBpcyB0aGUg
Ym9keSBvZiB0aGUgbWVzc2FnZS48L3A+CiAgPC9ib2R5Pgo8L2h0bWw+Cg==
--78942734hdfkipiriepiryewi7979rb213cxzb
Content-Disposition:form-data; name="file3"; filename="hola.html"
Content-Type: text/html

<html><body>HOLA</body></html>
--78942734hdfkipiriepiryewi7979rb213cxzb
Content-Disposition:form-data; name="overwrite"

true
--78942734hdfkipiriepiryewi7979rb213cxzb--
```

Podeu consultar la llista de codis de l'estatus i les seves definicions a: http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

D'altra banda, les respostes HTTP generades pel servidor han de seguir també un format determinat. L'estructura genèrica del missatge és la mateixa que la usada per a les peticions (vegeu figura 2.7). Ara bé, en aquest cas la primera línia indicarà l'estatus de la petició per tal que el client pugui saber en rebre la resposta si s'ha pogut satisfer i en cas que no s'hagi pogut, quin n'ha estat el motiu. La línia d'estatus conté també tres elements. El primer es correspon amb la versió del protocol utilitzat. De la mateixa manera que a la petició, caldria fer servir el valor actual HTTP/1.1. Separat per un espai seguirà un codi de 3 xifres corresponent al tipus de resposta generada. Així el codi 200 indica que la petició s'ha pogut servir amb èxit, el codi 301 significa que el recurs ha canviat d'adreça, el 403 que l'usuari no té suficients permisos i el 404 que no s'ha trobat el recurs demanat. Podeu trobar el detall de tots els codis al document RFC corresponent. Finalment el tercer element inclourà una frase explicativa del codi de format lliure per tal que pugui utilitzar-se per visualitzar l'estatus en cas que fos necessari.

Podeu consultar la llista de camps de capçalera i les seves definicions a: http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html

La resposta tindrà també camps de capçalera similars als de la petició que permetran al client reconèixer el tipus de contingut, la mida o d'altra informació que pot ser utilitzada per facilitar la presentació, valorar si cal emmagatzemar una còpia en local (memòria cau), etc. Els principals camps de la capçalera seran: `Server` per informar de l'aplicació que executa el servei WWW, `Set-Cookie` per mantenir actiu l'estat d'una connexió entre peticions consecutives, `Content-Length`, per indicar la mida del cos, `Content-Type` per determinar-ne el tipus, `Last-Modified` per indicar la data de la darrera modificació, `Expires` per controlar la data màxima que es podria emmagatzemar a la memòria cau, etc.

El cos es reserva pel contingut demanat o per la informació generada pel servei WWW amb indicacions a l'usuari. Cal clarificar que els documents HTML que contenen incrustades dades multimèdia, no solen subministrar-se al mateix temps que el document malgrat que es trobin emmagatzemades en el mateix servidor. De fet és el client el que decideix de fer les crides necessàries per acabar obtenint tot el document complert.

Exemple de resposta HTTP

Seguidament podeu veure dues respostes d'un servei WWW. La primera obtinguda amb una petició que ha tingut èxit i la segona amb una petició que contenia una URL inexistent.

Resposta amb èxit:

```
HTTP/1.1 200 OK
Date: Wed, 17 Jul 2013 18:32:48 GMT
Server: Apache/2.2.22 (Unix) PHP/5.3.10 mod_ssl/2.2.22 OpenSSL
    /0.9.7d
Set-Cookie: 5170f26bccc1e56c8e292c74405ffb45=en-GB; expires=Thu,
    17-Jul-2014 18:32:48 GMT; path=/
Cache-Control: no-cache, max-age=604800
Expires: Wed, 24 Jul 2013 18:32:48 GMT
Content-Type: text/html; charset=utf-8
Set-Cookie: BIGipServerphp-ioc23-pool=346921738.20480.0000; path
    =/

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
    http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-gb" lang
    ="en-gb" dir="ltr" >
 <head>
   <base href="http://ioc.xtec.cat/educacio/en/cicles-formatius"
       />
          <meta http-equiv="content-type" content="text/html;
              charset=utf-8" />
          ...
    </head>
    <body>
    ...

 </body>
</html>
```

Recurs no trobat:

```
HTTP/1.1 404 Not Found
Date: Wed, 17 Jul 2013 22:38:37 GMT
Server: Apache/2.2.22 (Unix) PHP/5.3.10 mod_ssl/2.2.22 OpenSSL
    /0.9.7d
Set-Cookie: 5170f26bccc1e56c8e292c74405ffb45=en-GB; expires=Thu,
    17-Jul-2014 22:38:37 GMT; path=/
Status: 404 Article not found
```

```
6  Cache-Control: no-cache, max-age=604800
7  Content-Length: 5537
8  Content-Type: text/html; charset=utf-8
9
10 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "
       http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
11 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-gb" lang
       ="en-gb" dir="ltr">
12 <head>
13 <meta http-equiv="content-type" content="text/html; charset=utf
       -8" />
14 <meta name="language" content="en-gb" />
15
16 <title>404 - Error: 404</title>
17 ...
18 </head>
19
20 <body>
21  <div id="wrapper2">
22  <div id="errorboxbody">
23  <h2>An error has occurred.<br />
24  The requested page cannot be found.
25  </h2>
26  <div id="searchbox">
27  <h3 class="unseen">Search</h3>
28  <p>You may wish to search the site or visit the home page.</p>
29  </div>
30  <div>
31  <p><a href="/educacio/index.php" title="Go to the Home Page">
       Home Page</a></p>
32  </div>
33  </div>
34  </div><!-- end wrapper -->
35 </body>
36 </html>
```

2.4.2 El llenguatge HTML

HTML és un llenguatge de marques que permet crear documents web. Va ser ideat per l'equip de Tim Berners-Lee amb la intenció de facilitar la compartició de la gran quantitat de documentació científica i tècnica que s'anava generant al CERN. Es va dissenyar un llenguatge d'etiquetes de text prou senzill com perquè els autors poguessin crear els documents amb un editor de text. Com que els autors no eren editors, ni havien de tenir coneixements d'edició, es va optar per un llenguatge de marques eminentment semàntic en el qual aquestes representaven parts lògiques del document com ara els títols o els paràgrafs.

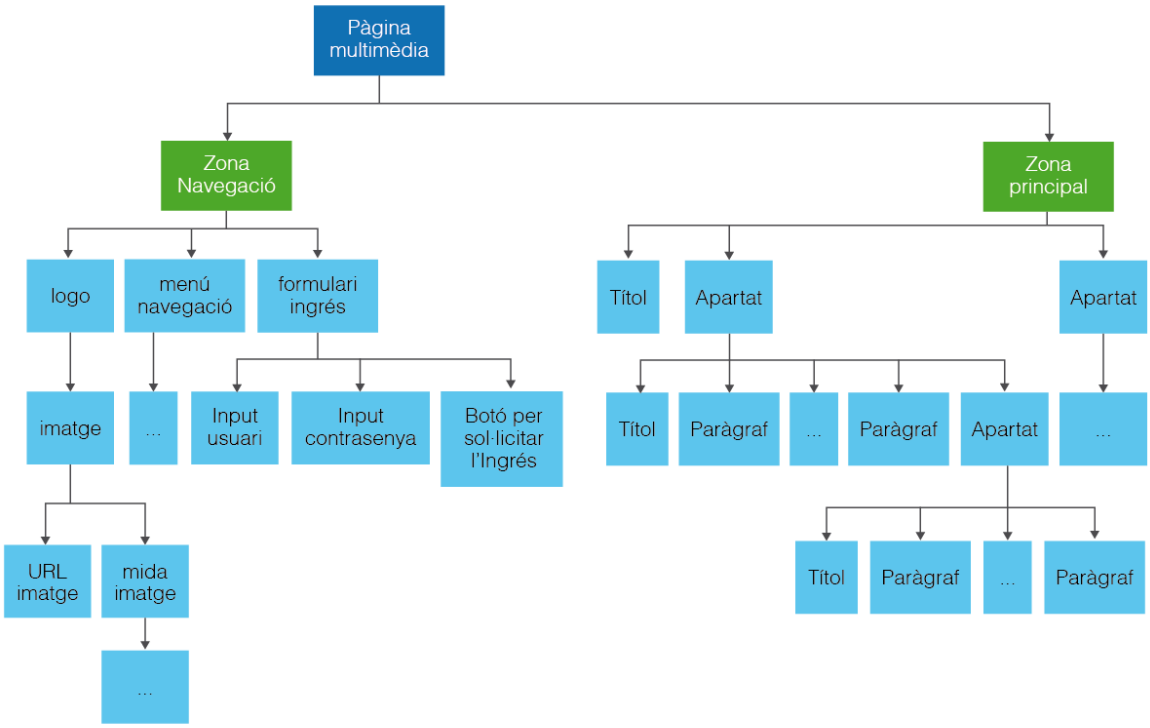
Des que va veure la llum a principis dels anys 90, fins avui, l'HTML ha patit molts canvis, ha incrementat intensament la seva funcionalitat i s'ha consolidat com una clara iniciativa, no només de creació de documents web, sinó també de definició d'interfícies gràfiques d'usuari. En la darrera versió publicada i recomanada pel W3C recentment (2012), es fa una aposta forta per una especificació clarament estructural (semàntica) capaç però d'absorbir múltiples tecnologies perifèriques (CSS, SVG, MathMl, llenguatges script, connexions asíncrones, etc.) que li donen una gran projecció de futur i una alta modularització. Això facilita, sens dubte, l'escalabilitat de la proposta.

En general, qualsevol document necessita una estructura interna que li doni coherència per poder ser llegit i entès. HTML permet definir aquesta estructura, identificant cada una de les seves parts. Es tracta d'una composició recursiva perquè a la vegada algunes d'aquestes parts poden definir-se també com una composició, formada d'altres parts més petites. El resultat pot visualitzar-se com una estructura jeràrquica en forma d'arbre en la qual cada node conté altres nodes (composició) a excepció de les fulles, en les quals trobarem el contingut propi del document, sigui contingut textual, gràfic o d'un altre tipus.

Estructura d'un document

Qualsevol document o pàgina multimèdia pot estructurar-se en diversos components, inclosos uns dintre d'altres seguint un esquema en forma d'arbre en el qual l'arrel es correspon amb la pàgina sencera, les fulles amb el contingut pròpiament dit i els nodes intermedis amb els contenidors que configuraran l'estructura interna del document (figura 2.8).

FIGURA 2.8. Esquema gràfic que exemplifica l'estructura d'un document multimèdia



Consulteu el mòdul anomenat *Llenguatges de marques i sistemes de gestió d'informació* de DAM per conèixer més detalls sobre els llenguatges de marques.

En l'argot dels llenguatges de marques, els components de l'estructura, els nodes de l'arbre, s'anomenen elements. Els llenguatges de marques com HTML disposen d'etiquetes de text per poder marcar els límits de cada element, no els límits físics sinó els estructurals. És a dir, quins elements estaran inclosos dins de quins altres. Les etiquetes determinaran l'estructura tancant el contingut entre una marca que indicarà l'inici de l'element i una que n'indicarà el final. El client decidirà la millor manera de mostrar el contingut a l'usuari donant forma a cada element. La funcionalitat i el comportament visual de les etiquetes pot modificar-se usant atributs i definicions d'estils de text (CSS).

Malgrat que el llenguatge HTML deriva originàriament de l'SGML (*Standard Generalized Markup Language*), hi ha una clara intenció d'anar-lo acostant a la forma més restrictiva de l'XML (*eXtensible Markup Language*). XML és una variant de l'SGML més restrictiva que simplifica el procés d'anàlisi dels documents que segueixen les seves especificacions. Per exemple, les etiquetes XML són sensibles a majúscules, és obligatori incloure sempre una única etiqueta que faci d'arrel de l'estructura, no es poden cavalcar etiquetes, cal deixar ben clar si un element es troba o no inclòs dins d'un altre, etc. Quan un document compleix les restriccions imposades per l'XML direm que es troba ben format. Els documents HTML ben formats tenen totes les seves etiquetes en minúscula i segueixen les seves restriccions, com les esmentades.

En versions anteriors a l'HTML 5, l'etiqueta `DOCTYPE` pot canviar el seu atribut ja que és necessari indicar quin és el DTD que segueix el document. La versió 5 ha intentat simplificar-ho ja que es tractava d'una dada molt infrautilitzada perquè els navegadors intenten sempre acceptar qualsevol document malgrat no compleixi el DTD indicat.

Per tal de poder reconèixer de forma senzilla un document HTML d'altres documents de text, caldrà començar indicant el tipus amb la etiqueta `<!DOCTYPE HTML>`. Es tracta d'una etiqueta opcional que ha d'estar situada, si hi és, a la primera línia i que no necessita tancament. L'etiqueta arrel de qualsevol document HTML s'anomena `html` i conté dos elements opcionals, `head` i `body`. L'element `head` conté metainformació adreçada a l'aplicació encarregada d'analitzar els documents. No és informació que calgui mostrar a l'usuari. A l'element `body`, en canvi, hi trobarem tot el contingut visible del document i la seva estructura interna.

Els principals elements de `head` són `title`, `meta`, `link` o `script`. El primer permet informar al client del títol amb el qual es coneix el document. No ha de coincidir amb el títol visible que podrà encapçalar el format visual, sinó que identifica el nom amb el qual es vol donar a conèixer el document. L'element `meta` conté informació diversa pel client. Es tracta d'informació que no es mostrarà a l'usuari però pot ser útil pel automatitzar alguns processos del navegador. L'element `link` permet vincular informació continguda en fitxers externs. Els principals vincles que l'element referencia solen ser fitxers d'estil CSS, però n'admet també d'altres formats. Finalment amb l'element `script` podrem incloure algoritmes imperatius que s'executaran durant la càrrega del document. Javascript és el llenguatge més estès i reconegut per a tots els navegadors. Els algoritmes es podran incloure directament dins l'etiqueta o bé es podran referenciar com un fitxer extern que caldrà llegir i executar en el moment de la càrrega de l'HTML.

Algunes de les etiquetes que podem trobar el contenidor `body` s'anomenen *de bloc* en contraposició a les etiquetes *lineals*. Les primeres són contenidors que tendeixen a ocupar tota la pantalla en funció del seu contingut i entre elles acostuma a haver-hi una separació clara. Les lineals, en canvi, són contenidors pensats per disposar-se en una mateixa línia de text per mantenir alguna característica comú d'un conjunt de paraules o lletres que pertanyin a un mateix element de bloc. Per exemple l'element `p` (paràgraf), `form` (formulari) o `div` (divisió) pertanyen als elements de bloc, en canvi `em` (èmfasi), `code` (codi) o `span` (extensió) són elements lineals.

Alguns elements, amb independència de si són de bloc o lineals, tenen funcions molt precises com ara `article` que vol representar la part central del document, el contingut en el qual es troba l'article, la informació central; com ara `abbr` que

permet identificar una abreviació o `table`, `tr`, `td` ... adequades per descriure informació tabular. Altres elements, per contra, són molt genèrics i permeten fer-se servir per a una gran quantitat de casos ben diferents. L'exemple més extrem són `div` o `span`. De fet són una abstracció que permet marca qualsevol cosa. La versió de bloc és `div` i la versió lineal és `span`. Entre les més específiques i les més genèriques trobarem un gran ventall d'elements.

El llenguatge HTML destina també un gran esforç a descriure aspectes gràfics de la interfície de l'usuari i a poder interactuar amb ells. Són components funcionals com ara botons, camps de text, selectors de fitxers o ràdios usant l'etiqueta `input` o `button`, llistes de selecció amb l'etiqueta `select`, imatges gràfiques amb l'element `canvas` o un editor de text multilínia amb `textarea`.

No disposem de suficient espai per podem estudiar cada un dels elements HTML en detall. Tot i així, malgrat que no és imprescindible, és convenient que estigueu familiaritzats amb el llenguatge. A la xarxa disposeu de molts manuals i exemples que us poden ajudar amb aquesta familiarització.

Qualsevol comportament per defecte de les etiquetes HTML pot ser manipulat i canviat fent servir estils. CSS permet descriure estils parcials que es van complementant entre ells. La forma més bàsica és la que associa un conjunt d'estils a una etiqueta. Això farà que totes les etiquetes d'un document HTML que responguin a la mateixa denominació es presentin de la mateixa manera. Sabent que `h1` és l'element que conté el títol principal, que `h2` conté els apartats i `h3` els subtítols dins de cada apartat, podem definir un estil per cada un d'ells de manera que tot el document guardi un grau de coherència estilística que ajudi a identificar el que s'estigui llegint.

Per facilitar la descripció d'estils i mantenir la coherència esmentada, CSS propicia la definició mínima d'estils. És a dir, només caldrà definir un estil quan sigui estrictament necessari. Per això CSS desplega sempre l'estil en cascada cap a tots els components d'un element. Amb això volem dir que si un element es definís amb una mida de font petita, tots els elements continguts en el primer requeriran també una mida de font petita sempre que l'estil de la seva etiqueta no indiqui una mida de font diferent.

A la pràctica, la majoria de documents necessiten moltes excepcions a l'hora de definir els estils de les seves etiquetes, per això és comú poder matisar la selecció d'un estil a una etiqueta o conjunt especific (per exemple d'una zona). Aconseguirem identificar conjunts particulars d'etiquetes marcant-les amb un nom a l'atribut `class`. els noms identificadors dels atributs `class`, a la definició CSS s'identifiquen perquè porten un punt com a prefix i s'anomenen selectors de classe. De forma semblant, podem identificar elements concrets usant el seu atribut identificador (`id`). En aquest cas, els selectors d'identificadors es reconeixen a la definició CSS amb el prefix #.

Alguns elements HTML mantenen diversos estats durant la visualització. Per exemple l'etiqueta indicada reconeix quan un hipervincle ha estat ja visitat, quan el ratolí es troba sobre d'ell o quan es realitza el clic. Els diferents estats pels quals una etiqueta pot arribar a passar s'identifiquen usant un nom amb el prefix `:` i a CSS s'anomenen selectors de pseudoclasse.

Quan les definicions d'estil són complexes, sovint trobarem etiquetes sobredefinides per diversos estils CSS. En aquest cas prevaldrà sempre la definició que faci coincidir més noms d'elements contenidors, classes i identificadors si fos el cas. En cas d'empat, prevaldrà l'etiqueta definida en darrera instància.

Exemple de definició d'estils CSS

En aquest exemple podeu veure estils definits amb una única etiqueta i tots els seus components en els quals l'estil específic no es trobi sobreescrit, estils amb selectors de pseudoclasses (a `link`, per exemple), amb selectors de classes (vegeu `h2.ex1`) o d'identificadors (`#tagid1` n'és un exemple). Aquells estils marcats amb múltiples selectors (`div.class10 ul li`) s'aplicaran a les etiquetes contingudes en els elements indicats seguint una estructura jeràrquica. Per exemple, el selector `div.class10 ul li` s'aplicarà a les etiquetes `li` contingudes en etiquetes `ul` i en etiquetes `div` marcades per la classe `class10`. A la resta d'etiquetes `li` no se'ls aplicarà l'estil.

```
body {
      background: black;
      color: #80c0c0
}

a:link {
    color: #ff8080
}

a:visited {
    color: #ff0000
}
p:first-line {
    margin-left: 25px
}

div.foo:first-line {
    font-weight: bold;
    margin-left: 30px
}

div:first-letter {
     font-size: x-large;
     color: #ffffff
}

div.class10 ul li {
    font-size: x-large;
    font-decoration: italic
}

h2 em {
    font-weight: 900
}

h2.ex1 {
    color: green;
    line-height: 50px;
    font-size: 40px
}

.funkyclass {
     font: 36pt/40pt courier;
     font-variant: small-caps;
     border: thick dashed blue
}
#tagid1 {
    padding: 20px;
    border: 20px groove #ffffff
}
```

```
51
52 .class1, #tagid2 {
53     font-family: 'comic sans ms', fantasy;
54     color: rgb(100%,100%,0%)
55 }
56 .part1 {
57     font-size: xx-large;
58     color: #808000
59 }
```

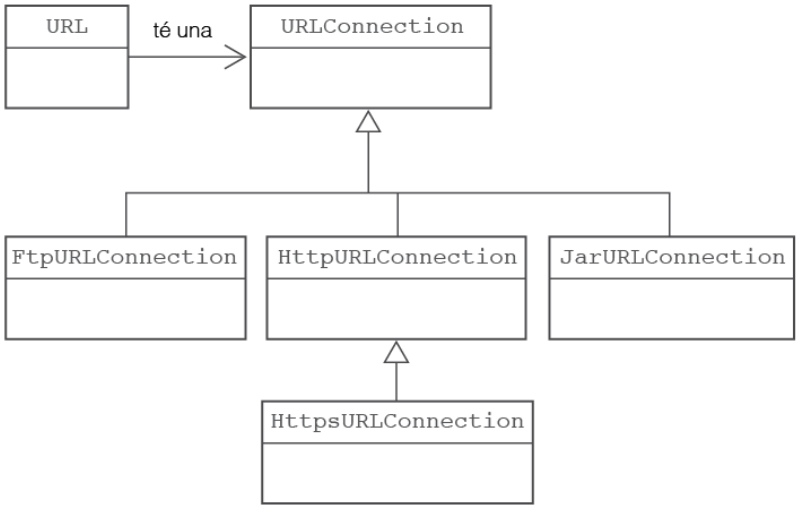
2.4.3 Implementació de clients web

Parlarem de clients de serveis web per referir-nos a aplicacions capaces de generar peticions HTTP, recollir la resposta que arribi del servidor, identificar l'estatus de la resposta i en cas d'èxit fer una tractament amb el contingut HTML o bé en cas d'error, fer una nova petició o presentar el problema a l'usuari. En general, doncs, la implementació dels clients requerirà dos processos força independents: l'encarregat de gestionar la connexió HTTP i l'encarregat de gestionar la informació HTML de la resposta.

Els navegadors són aplicacions clients que realitzen aquestes tasques per aconseguir mostrar una representació gràfica de la informació arribada des del servidor. El llenguatge HTML és tan versàtil que sovint els navegadors seran les úniques aplicacions necessàries a la banda client. La gran qualitat dels navegadors actuals fa difícil pensar en implementar noves alternatives. Tot i així, cal preveure la necessitat de crear aplicacions complementàries que permetin oferir respostes alternatives ja sigui en forma d'aplicació independent o integrades en un navegador concret.

Podem desenvolupar un gestor HTTP fent servir sòcols, però Java disposa d'una classe ja implementada que fa aquesta funció. Ens referim a `URLConnection`. Es tracta d'una classe abstracta que permet representar una connexió a un servei indeterminat com ara FTP o HTTP. La implementació del protocol HTTP es concreta amb la classe `HttpURLConnection` que serà la classe amb la qual realment treballarem sempre que demanem de fer connexions a servidors web. Vegeu l'esquema de la figura 2.9.

FIGURA 2.9. Jerarquia de classes URLConnection



De forma genèrica la classe `URLConnection` disposa del mètode `getHeaderField` per consultar qualsevol camp de la capçalera HTTP i `setRequestProperty` per assignar-hi valors. El primer rep com a paràmetre el nom del camp. El segon, a més del nom del camp, se li passa un segon paràmetre amb el valor que es desitja assignar. El valor retornat per `getHeaderField`, així com el paràmetre que assignarà el valor a `setRequestProperty` són de tipus *string*.

Exemple d'ús dels mètodes getHeaderField i serRequestProperty

Suposarem que `con` és una variable de tipus `URLConnection` i `charset` una variable de text que conté un joc de caracters vàlid, per exemple `UTF-8`.

```
con.setRequestProperty("Content-Type",
                            "application/x-www-form-urlencoded;
                                charset=" + charset)
...
if(weAccept(getHeaderField("Content-Type")){
    ...
}
```

El mètode de lectura `getHeaderField` accepta també com a paràmetre un número indicant la posició ocupada a la capçalera. D'aquesta manera és possible recórrer tots els camps sense necessitat de saber-ne el nom. En aquest cas el mètode `getHeaderFieldKey` ens retornarà el nom del camp.

```
for(int i=0; i<urlConnectionObject.getHeaderFields(); i++){
    System.out.print(urlConnectionObject.getHeaderFieldKey(i));
    System.out.print(": ");
    System.out.println(urlConnectionObject.getHeaderField(i));
}
```

La classe `URLConnection` disposa també de mètodes específics de consulta de camps de capçalera habituals que permeten treballar amb tipus de dades més adients. Podeu veure una relació d'aquest a la taula taula 2.4.

Taula 2.4. Taula en la qual es relacionen els mètodes de la classe URLConnection i els camps de la capçalera HTTP

Mètode per obtenir el valor	Camp HTTP	Tipus de dada usada	Comentaris
`getContentEncoding()`	`Content-Encoding`	String	
`getContentLength()`	`Content-Length`	`int`	`getContentLenth()` retornarà -1 si la capçalera no conté aquest camp
`getContentType()`	`Content-Type`	`String`	
`getDate()`	`Date`	`long`	Retorna el número de mil·lisegons transcorreguts des de l'1 de gener de 1970 o 0 si no hi ha data establerta a la capçalera. Es pot usar per construir un objecte `Date` o `Calendar`
`getExpiration()`	`Expires`	`long`	El format és el mateix que `getDate()`
`getLastModified()`	`Last-Modified`	`long`	El format és el mateix que `getDate()`

Si necessitem informació específica del protocol haurem de treballar amb la classe `HttpURLConnection`, però si la connexió l'hem obtinguda a través d'una instància d'URL, caldrà fer una conversió explícita.

```
URl url;
//conversió d'URLConnection a HttpURLConnection
HttpURLConnection connection = (HttpURLConnection) url.openConnection();
...
```

La classe `HttpURLConnection` disposa, per exemple, de mètodes com ara `getResponseCode()` per aconseguir el valor de l'estatus d'una resposta o el seu missatge amb `getResponseMessage()`. La classe permet també obtenir o assignar el mètode HTTP de la petició (`GET`, `POST`, `PUT`, `DELETE`...) així com altres funcionalitats específiques del protocol HTTP.

El funcionament d'aquesta classe és força senzill. Abans d'enviar la petició podrem manipular-la fent servir els mètodes d'assignació com ara `setRequestMethod(nomDelMetodePeticioHttp)` o `setRequestProperty(nomCampPeticio, valorCampPeticio)`. Un cop enviada la petició no serà possible assignar-hi dades a la capçalera i si ho intentem ens respondrà amb una excepció. Usant aquesta classe, no és necessari fer un enviament explícit de la petició HTTP, perquè aquest es realitza de forma automatitzada en realitzar qualsevol consulta referida a la resposta, com per exemple `getResponseCodeo` `getHeaderField`.

Quan el mètode utilitzat per realitzar la consulta sigui `POST`, podrem escriure el cos del missatge fent servir el mètode `getOutputStream` que ens retornarà el flux de sortida en el qual podrem anar escrivint tot el que sigui necessari de fer arribar en el cos. En aquest cas tampoc serà possible escriure dades un cop la petició hagi estat enviada.

Podem forçar l'enviament de la petició de forma explicita usant els mètodes del flux de sortida que forcin l'enviament com ara `flush` o `close`. Un cop enviada la petició és important fer el tancament del flux per alliberar els recursos de la xarxa. A continuació se us mostra un codi que escriu el contingut d'un fitxer al cos del missatge usant el tipus `multipart` de MIME.

```
URLConnection com=...;
String fieldName=...;
String fileName=...;
String boundary=...;
String contentType=...; //Tipus de contingut específic del fitxer a enviar

...

//Indiquem el tipus multipart a la capçalera
con.setRequestProperty("Content-Type", "multipart/form-data; boundary="+
    boundary);

...

//Escrivim el contingut del fitxer
byte[] buf = new byte[1024];
int len=0;
con.getOutputStream().write("--".getBytes());
con.getOutputStream().write(boundary.getBytes());
con.getOutputStream().write("\r\n".getBytes());
con.getOutputStream().write(("Content-Disposition: form-data; name=\""
                            +fieldName+"\"; filename=\""
                            +fileName+"\"").getBytes(charset));
con.getOutputStream().write("\r\n".getBytes());
con.getOutputStream().write("Content-Type: ".getBytes());
con.getOutputStream().write(contentType.getBytes());
con.getOutputStream().write("\r\n".getBytes());
con.getOutputStream().write("\r\n".getBytes());
//Comencen les dades del fitxer
FileInputStream stream = new FileInputStream(fileName);
while((len=stream.read(buf))!=-1){
    con.getOutputStream().write(buf, 0, len);
}
stream.close();
con.getOutputStream().write("\r\n".getBytes());

...
```

Un cop completades totes les parts, podrem forçar l'enviament de la petició fent:

```
//Tanquem la darrera frontera
con.getOutputStream().write("--".getBytes());
con.getOutputStream().write(boundary.getBytes());
con.getOutputStream().write("--".getBytes());
//Enviem la petició i tanquem el flux
con.getOutputStream().flush();
con.getOutputStream().close();

...
```

Recordeu que la consulta de qualsevol element de la resposta, tant les dades de la capçalera com el valor de l'estatus, comporta l'enviament immediat de la petició. Per tant cal anar en compte de realitzar les consultes sempre en l'ordre adequat.

La classe `HttpURLConnection`, disposa també de l'opció de demanar que es realitzi una nova petició quan el codi de l'estatus sigui de la forma 3XX, que indica sempre un valor de redireccionament causat per un trasllat (temporal o

definitiu) del recurs demanat. El mètode que permet realitzar aquesta configuració és `setInstanceFollowRedirects(certOFals)`. El mètode rebrà un valor lògic cert si es desitja fer efectius els redireccionaments o fals en cas contrari.

Per recollir el valor de cos de la resposta, usarem el mètode `getInputStream` que retornarà el flux d'entrada de dades provinents del servidor. L'obtenció de les dades seguirà la dinàmica habitual dels fluxos de dades. D'entrada el flux recollit és de tipus byte, però si sabem que el contingut és de tipus text, podem convertir-lo a un flux de caràcters per facilitar-ne el tractament.

Generalment el client rebrà un document HTML que haurà d'interpretar, ja sigui per mostrar a l'usuari o per realitzar qualsevol altra tasca. El tractament dels documents HTML solen fer-se utilitzant un tipus d'aplicacions anomenades analitzadors sintàctics. Els analitzadors sintàctics serveixen per aconseguir interpretar qualsevol producció que segueixi la sintaxi d'un llenguatge formal. Els llenguatges de programació com Java són llenguatges formals i el seu analitzador sintàctic és el compilador o l'intèrpret segons treballin de forma interpretada o compilada. Els llenguatges de marques derivats d'SGML o XML són també llenguatges formals i poden analitzar-se de forma semblant. La majoria de llenguatges de programació disposen de biblioteques amb analitzadors genèrics per facilitar el tractament dels documents que segueixin les especificacions d'aquests llenguatges.

Distingim dos tipus d'analitzadors. Aquells que llegeixen seqüencialment el document i gestionen el tractament que s'ha de realitzar fent invocacions específiques, a mode d'esdeveniments, cada cop que detectin el començament o final d'una etiqueta, o un cop llegit el contingut textual d'un element. Les dades llegides durant la seqüenciació s'aniran passant per paràmetres als mètodes invocats durant l'anàlisi. Així, per exemple, el mètode invocat en detectar l'obertura d'una etiqueta rebrà probablement el nom de l'etiqueta, la col·lecció d'atributs associats amb els seus valors i d'altra informació complementària. Sabent la informació traspassada podrem codificar les accions a realitzar amb aquestes dades.

Exemple de tractament d'un esdeveniment d'un analitzador sintàctic de marques

Suposem que el mètode invocat sigui `startTag` i que per paràmetre rebi una cadena amb el nom i un `HashMap` amb els atributs i valors. Un possible esquema seria:

```
1  public void starTag(String tagName, HasMap<String, String>
       attributes){
2
3      if(tagName.equalsIgnoreCase("html")){
4
5          processStartHtmlTag(attributes);
6
7      }else if(tagName.equalsIgnoreCase("body")){
8
9          processStartBodyTag(attributes);
10
11     }else if(tagName.equalsIgnoreCase("meta")){
12
13         processStartMetaTag(attributes);
14
15     }else if(...){
16
17         ...
```

```
18     }
19
20 }
```

D'aquesta manera podem crear un codi ben diferenciat i fàcil de mantenir per a cada etiqueta. Imaginem que volem construir un navegador per visualitzar documents HTML. Davant l'esdeveniment de l'inici d'una etiqueta de tipus `link` s'ha de detectar si es tracta d'un full d'estils, recuperar les dades del fitxer vinculat i carregar-les en memòria per tal que es puguin recuperar quan sigui necessari aplicar-les usant la informació relativa a cada etiqueta. Si en canvi l'etiqueta fos de tipus `img` el que cal fer és recuperar els bytes corresponents a la imatge, crear un instància de la classe `Image` amb les dades obtingudes, adaptar-la a la mida que ens indiquin els atributs, obtenir la posició del bloc veí per calcular la posició en la qual situarem la imatge i finalment plasmar mostrar la imatge per la pantalla. Cada element necessitarà un tractament específic. Caldrà també mantenir les dades comuns en memòria per tal de poder realitzar adequadament els càlculs sobre posicions, mides i estils a aplicar.

Exemple de processament d'una etiqueta link

En aquest exemple suposem que disposem d'un analitzador del format CSS, que anomenem `cssParser`. Quelcom semblant a l'analitzador que intentem construir però adaptat al format CSS i que en lloc de detectar etiquetes, detecta selectors i extreu el conjunt d'estils associats. Imaginem que disposem també d'una classe que emmagatzema els estils i permet ser interrogada a partir del nom, atributs i jerarquia a la qual pertanyi una etiqueta sobre la qual calcular l'estil a aplicar. Suposem que disposa d'un mètode anomenat `add` per afegir el conjunt d'estils analitzats per l'objecte `cssParser`.

```
1  public void processStartLinkTag(Map<String, String> attributes){
2      if(attributes.containsKey("rel")){
3          if(attributes.get("rel").equalsIgnoreCase("stylesheet")){
4              //Per defecte ho considerem CSS
5              String type= "text/css";
6              if(attributes.containsKey("type")){
7                  type = attributes.get("type");
8              }
9              if(type.equalsIgnoreCase("text/css")){
10                 try {
11                     URL url;
12                     String strUrl = attributes.get("href");
13                     if(strUrl.matches("\\b(https?|ftp|file)
                           ://.*\\b")){
14                         url = new URL(strUrl);
15                     }else if(strUrl.matches("//.*")){
16                         url = new URL(getProtocolBase()+strUrl);
17                     }else{
18                         url = new URL(getProtocolBase(),
                               getUrlBase(), strUrl);
19                     }
20                     cssManager.addCss(url.openStream());
21
22                 } catch (MalformedURLException ex) {
23                     //Tractament de l'error
24                 } catch (IOException ex) {
25                     //Tractament de l'error                }
26             }else{
27             //Un altre tractament...
28             }
29         }else{
30             //Un altre tractament...
31         }
32     }
33 }
```

Noteu la complexitat que implica la implementació de tot un navegador.

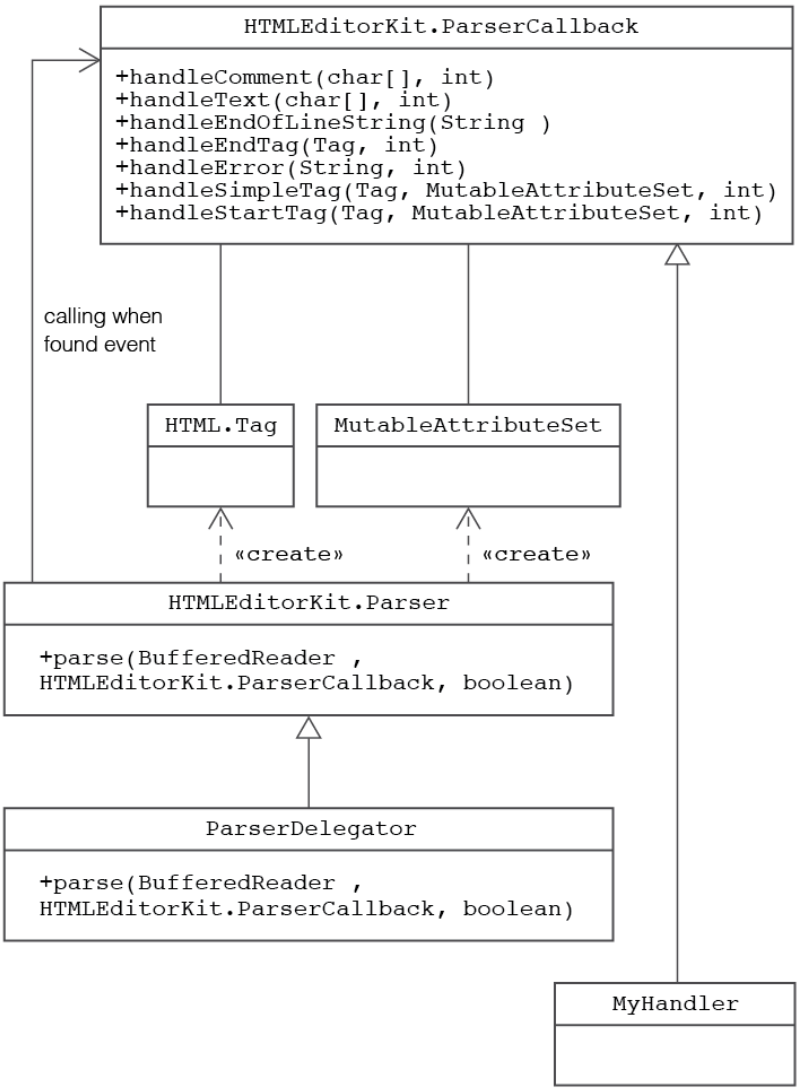
Java disposa de la biblioteca `SAX` per realitzar l'anàlisi de documents XML. El principal problema d'aquesta biblioteca són les restriccions pròpies de l'XML. Els documents mal formats provoquen errors durant l'anàlisi que acaben avortant el procés. De fet usant SAX només tindríem garantia d'èxit llegint documents XHTML perquè segueixen la sintaxi HTML respectant totes les restriccions dels XML. Malauradament a la xarxa hi ha una gran quantitat de documents HTML mal formats i per tant SAX no és el millor candidat.

La biblioteca `swing` del JDK, en canvi, disposa d'un conjunt de classes menys restrictives capaces també de fer una anàlisi sintàctic amb el corresponent llançament d'esdeveniments. La classe `ParserDelegator` s'encarrega de llegir el document i detectar el moment en què caldrà fer el llançament d'esdeveniments que concretament es limiten a 7 casos:

1. Quan es troba un comentari,
2. Quan es troba una etiqueta sense contingut
3. Quan es troba l'inici d'una etiqueta
4. Quan es troba el final
5. Quan es troba contingut textual
6. Quan es troba un salt de final de línia
7. Quan es detecta un error.

Per cada un d'aquests esdeveniments `ParserDelegator` invocarà un dels mètodes de la classe `ParserCallback` o alguna classe derivada. De fet, el normal és que s'utilitzin classes derivades per poder personalitzar el tractament a donar per cada esdeveniment. A la figura 2.10 se simbolitza això amb la classe `MyHandler` que representa la classe que personalitza el tractament. De fet, `ParserCallback` és una classe interna d' `HTMLEditorKit` i només existeix per poder ser invocada durant l'anàlisi, ja que tots els seus mètodes estan buits, no realitzen cap tasca. La idea de separar l'analitzador (`ParserDelegator`) del tractament (`MyHandler`) és molt potent perquè en essència dos analitzadors que haguessin de fer coses diferents realitzarien sempre la lectura i la detecció d'etiquetes de la mateixa manera i l'únic que variaria seria el tractament. Per realitzar una còpia en el disc dur del codi HTML i de tota la resta de fitxers vinculats com ara imatges, fitxers script, fulls d'estil, etc. caldrà crear només un classe derivada de ParseCallback. Per contra si decidim implementar un navegador que visualitzi el document. Podem aprofitar tot el codi realitat a excepció de la classe derivada de `ParserCallback` que haurà de fer un tractament diferent (vegeu la figura 2.10).

FIGURA 2.10. Jerarquia de les classes necessàries per a la implementació d'un parser o analitzador d'un document HTML



El mètode `handleStartTag(Tag t, MutableAttributeSet a, int pos)` rep per paràmetre l'etiqueta en forma de classe `Tag`. La classe `Tag` és un classe que guarda el nom de l'etiqueta, però que a més manté informació addicional sobre si l'etiqueta és de tipus bloc o lineal, o sobre si el seu contingut és de tipus preformatat o bé cal aplicar-li les regles d'estil corresponents. Generalment, només usarem aquest paràmetre per identificar el nom de l'etiqueta. La classe `Tag` conté també la majoria d'etiquetes HTML ja instanciades en forma de constant. El segon paràmetre rebut és una instància de `MutableAttributeSet`. Aquesta classe representa el conjunt d'atributs i disposa de mètodes per recórrer tots els seus elements i obtenir-ne la informació detallada (nom i valor). Finalment el tercer paràmetre informa de la darrera posició llegida just abans de fer la crida de l'esdeveniment.

El mètode `handleSimpleTag(Tag t, MutableAttributeSet a, int pos)` és molt similar a `handleStartTag`. Rep els mateixos paràmetres, però aquesta crida només es produeix quan es detecta un etiqueta simple, de les que es tanquen en la mateixa marca d'obertura, com per exemple `<br/>`.

El mètode `handleEndTag(tag : HTML.Tag, pos : int)`, s'invoca en detectar una marca de tancament i rep per paràmetre l'etiqueta que s'està tancant i la darrera posició llegida.

El mètode `handleText(data : char[], pos : int)` permet obtenir el contingut textual contingut a la darrera etiqueta oberta i encara no tancada. El contingut arriba en forma d'*array* de caràcters. El segon paràmetre fa referència també a la darrera posició llegida.

Semblant a l'anterior `handleComment(data : char[], pos : int)` rebrà les dades contingudes dins d'un comentari situat en qualsevol posició dins l'HTML. Normalment els comentaris s'insereixen entre el contingut textual. Per això la invocació d'aquest esdeveniment acostumarà a donar-se entre dues crides a `handleText`.

Finalment `handleError(errorMsg : String, pos : int)` s'invoca en detectar algun tipus d'errada, ja sigui sintàctica o d'altre tipus i s'utilitza per poder obtenir informació del tipus d'errada i poder avisar l'usuari.

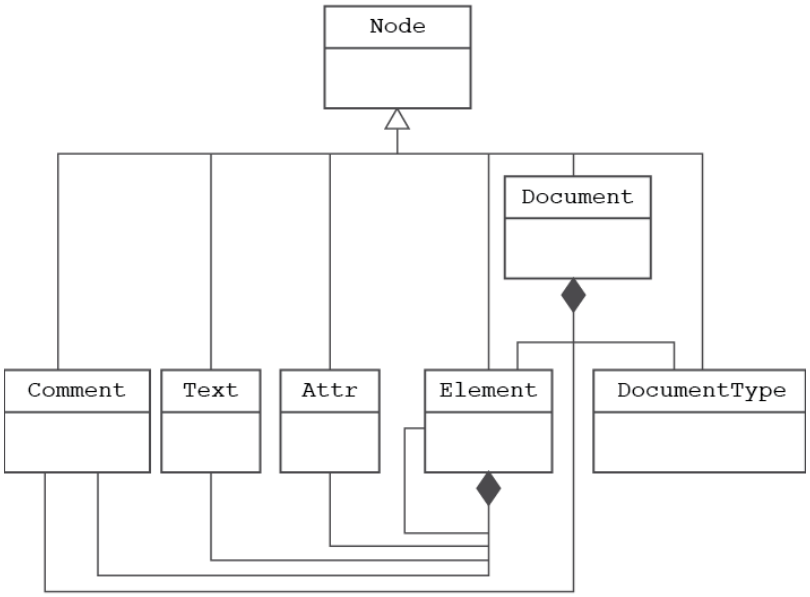
Un exemple simple que mostra per pantalla les URL de les etiquetes `link`, `script` o `img` en ajudarà a veure com funciona. La classe `HtmlEditorKitParser` ens permet configurar un objecte `PerserDelegator` amb una instància derivada de `ParserCallback` que passarem per paràmetre del mètode `init`. El mètode `parse` fa la crida pertinent per iniciar l'anàlisi usant la classe `ParserCallback` disponible. La classe `HtmlPrintExternalPathsHandler` és l'encarregada de detectar si l'etiqueta és del tipus que esperem, recollir l'URL i mostrar-la per pantalla. Com que són etiquetes que poden contenir text, però que també poden agafar la forma simple, recollirem ambdós esdeveniments en el procés anomenat `processTag`.

```
1  public class HtmlEditorKitParser{
2      HTMLEditorKit.ParserCallback handler;
3      ParserDelegator parser;
4      boolean ignoreCharset;
5
6      public void init(ParserCallback handler, boolean ignoreCharset) {
7          this.init(handler);
8          this.ignoreCharset=ignoreCharset;
9      }
10
11     public void parse(InputStream in) throws IOException {
12         parser.parse(new BufferedReader(new InputStreamReader(in)));
13     }
14 }
15
16
17 public class HtmlPrintExternalPathsHandler extends HTMLEditorKit.ParserCallback
       {
18     @Override
19     public void handleStartTag(Tag t, MutableAttributeSet a, int pos) {
20         processTag(t, a);
21     }
22
23     @Override
24     public void handleSimpleTag(Tag t, MutableAttributeSet a, int pos) {
25         processTag(t, a);
26     }
27
28     public void processTag(Tag t, MutableAttributeSet a){
```

```
29          String path;
30          if(t.equals(HTML.Tag.LINK)){
31              String value = (String) a.getAttribute(HTML.Attribute.HREF);
32              if(value!=null){
33                  processPath(value);
34              }
35          }else if(t.equals(HTML.Tag.SCRIPT)){
36              String value = (String) a.getAttribute(HTML.Attribute.SRC);
37              if(value!=null){
38                  processPath(value);
39              }
40
41          }else if(t.equals(HTML.Tag.IMG)){
42              String value = (String) a.getAttribute(HTML.Attribute.SRC);
43              if(value!=null){
44                  processPath(value);
45              }
46          }
47      }
48
49      private void processPath(String path){
50          System.out.println("Path: " + path);
51      }
52  }
```

Abans de començar a parlar de la implementació de la part del servidor, cal tenir en compte que el W3C ha creat una jerarquia d'interfícies per poder emmagetzemar una estructura XML/HTML qualsevol i facilitar la cerca i l'accés a qualsevol dels seus elements. Es coneix amb el nom de DOM (*Document Object Model*) i pot arribar a ser molt útil en aplicacions que hagin d'accedir contínuament als seus elements (vageu la figura 2.11). El tractament d'esdeveniments com el que acabem de veure només resulta pràctic quan s'han de realitzar un sol cop, però els navegadors actuals fan una tractament dinàmic de la informació i han de recuperar o modificar dades parcials de qualsevol part del document. En aquest cas es fa imprescindible usar DOM.

Figura 2.11. Jerarquia de classes Java per implementar DOM



Aquí es representen les classes principals ja que la jerarquia és molt més complexa i extensa

Java disposa de la classe `DocumentBuilder` per generar, a partir d'un `InputStream` que contingui un document XML, un objecte `Document` que representi l'estructura interna de l'XML llegit. Malauradament presenta la mateixa problemàtica que `SAX`. Com a alternativa podem construir un objecte DOM fent servir una classe `ParserCallback` ja que no resulta massa difícil d'implementar.

2.4.4 Implementació servidors web

Malgrat que la funció original dels servidors web era la de rebre peticions, cercar el document demanat entre els seus fitxers i enviar la resposta juntament amb el document si es trobava, actualment la majoria de llocs web generen les pàgines demanades dinàmicament. És a dir, els servidors han passat de ser un repositori de documents HTML a convertir-se en complexes aplicacions que fusionen fragments d'HTML amb dades provinents d'altres fonts com ara SGBD o fins i tot d'altres serveis externs.

El principal repte que se'ns planteja a l'hora de fer una implementació dels servidors web com a repositoris de documents té a veure amb la rapidesa a l'hora de servir les pàgines HTML i el suport de gran quantitat de peticions concurrents, que bàsicament se soluciona incrementant la potència del maquinari i afinant tant com sigui possible l'ús de la memòria i la gestió dels fils per a possibilitar la concurrència. Avui dia existeix una gran quantitat d'aplicacions molt eficients i seria absurd plantejar-nos implementacions alternatives.

En canvi, la generació dinàmica de les pàgines web implica una major complexitat i la intervenció necessària dels programadors en qualsevol projecte nou. Actualment existeix una gran quantitat d'aplicacions capaces de gestionar les peticions

HTTP i generar un conjunt d'utilitat que permetin als programadors agilitzar la creació de les seves aplicacions. De forma genèrica s'anomenen servidors d'aplicacions web perquè és possible desenvolupar-hi aplicacions complexes basades en HTTP i HTML. Són aplicacions molt dependents del llenguatge de programació utilitzat ja que necessiten integrar el que els programadors desenvolupin amb les biblioteques internes que disposin, per això cada servidor d'aplicacions web sol servir només per un únic llenguatge de programació.

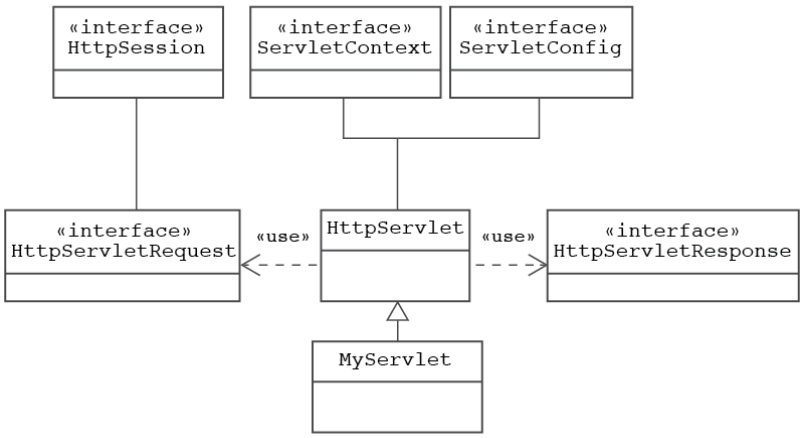
Mireu l'annex anomenat `Configurar NetBeans per treballar amb un servidor d'aplicacions web (contenidor EE) integrat a l'IDE` per saber com treballar amb un servidor d'aplicacions web des del NetBeans.

Java disposa d'una gran varietat de servidors d'aplicacions, malgrat que els més coneguts són Tomcat i Glassfish. El primer és un projecte d'Apache i el segon del propi Oracle. El desenvolupament original es remunta a Sun Microsystems, que Oracle va comprar juntament amb el desenvolupament de Java.

Tots els servidors destinats al llenguatge Java hauran de seguir l'estàndard especificat a la plataforma JEE (Java Enterprise Edition). JEE és un extensió del JDK que inclou l'especificació per crear aplicacions remotes. L'especificació JEE és realment molt extensa i s'escapa totalment dels objectius d'aquest mòdul. Tot i així caldrà donar una ullada a les biblioteques bàsiques que suporten l'extensió. Precisament aquestes enllacen directament amb el protocol HTTP, el que comunament es coneix amb el nom genèric de *servlets*.

Anomenem genèricament *servlets* a la jerarquia de classes Java que modelitza el protocol HTTP de la part del servidor. Malgrat que hi ha moltes més classes, a la figura 2.12 trobareu les classes bàsiques que us permetran realitzar qualsevol aplicació web.

Figura 2.12. Model de classes que representa el protocol HTTP



Aquestes classes permeten escriure noves aplicacions web creant classes que heretin d'HttpServlet i que en aquest esquema simbolitzem amb el nom MyServlet.

La classe principal del model és `HttpServlet`. No és una classe funcional perquè tots els seu mètodes es troben buits. De fet, bàsicament serveix per poder crear classes derivades que seran les que realment implementaran la funcionalitat als seus mètodes i que en l'argot Java es coneixen habitualment com a *servlets*.

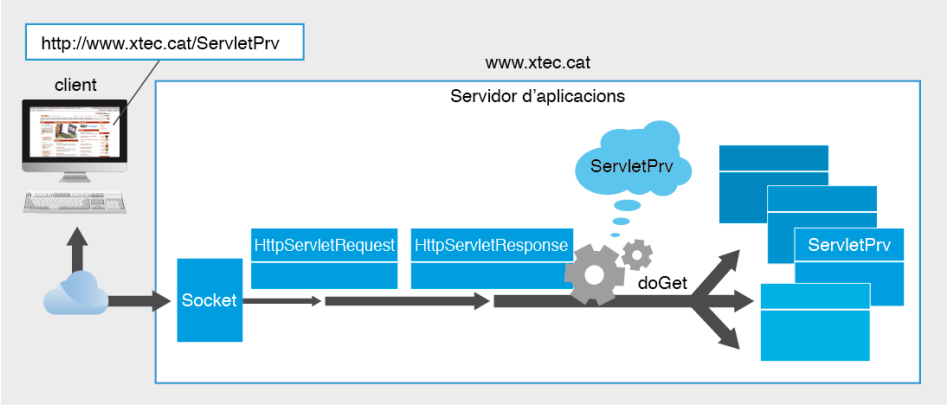
`HttpServlet` i les seves derivades disposen de mètodes que seran cridats cada cop que el servidor d'aplicacions rebi una petició HTTP d'algun client. Els mètodes

s'anomenen `doGet`, `doHead`, `doPost`, `duPut`, `doDelete`, `doTrace` i `doOptions` i es troben totalment relacionats amb el mètode HTTP especificat a la capçalera de la petició, de manera que la invocació es realitzarà en funció del mètode HTTP. Així un petició HTTP amb el mètode `GET` provocarà una execució del mètode `goGet` de la instància d' `HttpServlet` especificada a la URL de la petició. El mètode `doPost` s'executarà quan a la petició hi figuri el mètode `POST` i així successivament.

Habitualment els servidors d'aplicacions que treballen amb Java s'anomenen també contenidors de *servlets* o simplement contenidors.

Un servidor d'aplicacions pot contenir molts *servlets*, d'aquí que habitualment se'l coneix com a contenidor. Quan arribi una petició d'un client, el contenidor la llegirà i guardarà les seves dades en un objecte de tipus `HttpServletRequest`. També crearà una objecte de tipus `HttServletResponse` preparat per gestionar l'enviament de la resposta al client. Finalment analitzarà la URL per reconèixer el nom del *servlet* al qual caldrà invocar (vegeu la figura 2.13). Si ja existeix en memòria, una instància del *servlet* demanat (degut a peticions anteriors) executarà el mètode adequat d'aquesta instància. Si a la memòria no hi hagués encara cap instància del *servlet*, el servidor d'aplicacions en crearà una i seguidament executarà el mètode. Per agilitzar el procés de crida, els *servlets* invocats solen deixar-se en memòria mentre hi hagi prou espai i el rendiment sigui òptim. En situacions crítiques, el servidor d'aplicacions pot decidir eliminar les instàncies d'aquells *servlets* menys populars o dels que faci més temps que no es rebi una petició.

Figura 2.13. Esquema que il·lustra el funcionament d'un servidor d'aplicacions



La petició dels clients inclouen el nom dels servlets a executar. El servidor crea un objecte HttpServletRequest amb les dades rebudes i un HttpServletResponse per poder retornar la resposta, escull el servlet i invoca el mètode corresponent.

En carregar una instància d'un *servlet* a la memòria, es pot incialitzar amb un objecte `ServletConfig` que contindrà els paràmetres inicials per aconseguir que el *servlet funcioni*. Per exemple podríem carregar les dades de connexió a un SGBD si el *servlet* necessités treballar amb les dades d'una base de dades.

Els mètodes dels *servlets* candidats a ser executats en una petició (`doGet`, `doPost`, etc.) rebran sempre dos paràmetres, un amb la instància `HttpServletRequest` i l'altre amb la instància `HttServletResponse`. La instància `HttpServletRequest` contindrà totes les dades rebudes a la petició, com ara els paràmetres arribats en el cos o a través de l'URL, els camps de capçalera de la petició, les galetes (*cookies*) del client, les dades de connexió (adreça IP, port, etc.) o la URL utilitzada per aquesta petició. A més, el paràmetre `HttpServletRequest` inclou molta informació provinent del contenidor com

ara si l'usuari està o no autenticat, la sessió que el contenidor manté oberta per aquesta connexió i permet, si cal, iniciar un procés d'autenticació. El següent exemple ens mostra l'ús d'aquests mètodes.

Ús dels mètodes del paràmetre HttpServletRequest per obtenir la informació que conté

Suposarem que la URL demanada era /NomServlet?joc=parxis&categoria=principiant.

```
1
2  protected void doGet(HttpServletRequest request,
3                                  HttpServletResponse response)
4                                          throws ServletException,
                                               IOException {
5          boolean dadesErronies=false;
6          boolean html=true;
7          //Recollida dels paràmetres
8          String joc=request.getParameter("joc");
9          String categoria = request.getParameter("categoria");
10         String marca = request.getParameter("marca");
11
12         dadesErronies= joc!=null &&  categoria!=null && marca!=
               null;
13         String login = request.getRemoteUser();
14
15         String returnMessage;
16
17         if(dadesErronies){
18             //Error paràmetres erronis
19             returnMessage = "Falten paràmetres";
20         }else{
21             registrarMarca(login, joc, categoria, marca);
22             returnMessage = "Marca registrada correctament";
23         }
24
25         //Dades de la capçalera
26         String acceptType = request.getHeader("Accept");
27         if(acceptType!=null && !acceptType.matches(".*text/html.*
               ")){
28             //Escriurem text pla perquè el client no accepta HTML
29             html=false;
30         }
31
32         ...
33 }
```

El paràmetre de tipus `HttpServletResponse`, permetrà gestionar la resposta que cal donar a l'usuari. És a dir permetrà assignar l'estatus, afegir camps de capçalera, assignar el valor de les galetes i escriure el contingut a retornar que habitualment estarà formatat en HTML en un flux de tipus *writer* que la instància posarà a la nostra disposició. De nou un exemple ens ajudarà a entendre-ho.

Ús dels mètodes del paràmetre HttpServletResponse per enviar informació al client

A l'exemple, l'HTML generat s'escriu en el propi *servlet*. Això passa molt poques vegades. Només en els casos que la resposta a generar sigui molt simple. Normalment es treballa amb altres tecnologies més avançades com ara Java Server Page per facilitar la creació de respostes complexes, però la seva explicació cau fora de l'objectiu d'aquest mòdul.

```
1
2  protected void doGet(HttpServletRequest request,
3                                  HttpServletResponse response)
4                                          throws ServletException,
                                               IOException {
```

```
5       boolean dadesErronies=false;
6
7           ...
8
9       //Valor Content-Type de la capçalera
10      response.setContentType("text/html;charset=UTF-8");
11      //Per defecte l'estatus és 200
12      if(dadesErronies){
13          //Assigna l'estatus i el missatge
14          response.sendError(404, "Dades incorrectes a la petició")
                ;
15      }else{
16          response.addCookie(new Cookie("lastAction", "setScore"));
17      }
18      //Obtenim el flux en el qual escriure el contingut
19      PrintWriter out = response.getWriter();
20      try {
21          out.println("<html>");
22          out.println("<head>");
23          out.println("<title>Servlet Prv</title>");
24          out.println("</head>");
25          out.println("<body>");
26          out.println("<h1> Enregistrament de la puntuació</h1>");
27          out.println("<p>");
28          out.println(returnMessage);
29          out.println("</p>");
30          out.println("</body>");
31          out.println("</html>");
32      } finally {
33          out.close();
34      }
35  }
```

2.5 Altres serveis

A Internet podem trobar molts més serveis funcionant. Alguns, com els serveis DNS, es van originar conjuntament amb els primers serveis i han anat mantenint-se perquè encara donen un servei útil i en alguns casos com el citat, imprescindible. També hi ha hagut serveis que han anat sucumbint pel camí o es troben en fase d'extinció perquè d'altres serveis s'han imposat per eficiència o simplement per popularitat.

L'aparició de nous serveis és un fet força habitual. Alguns poden aparèixer com a resposta a noves necessitats que l'ús de la xarxa genera. Per exemple davant de la impossibilitat d'aconseguir adreces IP fixes, apareix la iniciativa DynDNS amb un èxit més que notable. Aquest servei permet associar un nom de domini a adreces IP dinàmiques. D'altres sorgeixen totalment vinculats a l'aparició de noves tecnologies com per exemple la quantitat creixent de serveis per a mòbils.

Actualment ens trobem en un moment de gestació de la futura web 3.0 en el qual els serveis tindran una paper fonamental. La qualitat aconseguida en el camp de la cerca constitueix l'embrió d'una gran quantitat d'idees en les quals la gestió del coneixement, la cerca semàntica, les relacions socials, l'oci o els serveis d'alertes personals i geolocalitzats es troben al centre de les investigacions en combinació amb l'aparició de nous i sofisticats dispositius. El desenvolupament de serveis tot just acaba de començar.

2.6 Escalabilitat dels serveis

El ritme de creixement de la xarxa ens obliga a posar sobre la taula de qualsevol desenvolupament el concepte d'escalabilitat. Entenem per escalabilitat la capacitat de fer créixer una aplicació o servei a mesura que sigui necessari sense perdre eficiència en el temps de resposta.

Els principals factors de creixement que poden fer minvar l'eficiència d'un servei són: l'increment del nombre de processos, l'increment de dades a processar i l'increment del nombre d'usuaris que en facin ús. Els dos primers els agruparem sota el nom d'increment de contingut. El darrer s'acostuma a conèixer com a increment de la concurrència, ja que la pèrdua d'eficiència es deu a la coincidència de peticions simultànies.

La capacitat del maquinari esdevé sempre decisiva en aquests casos, però de res servirà si menystenim aspectes del desenvolupament de programari. Un bon disseny, l'ús d'emmagatzematge temporal o una implementació acurada que controli la despesa de recursos i maximitzi el temps del processador permetran aconseguir sens dubte optimitzacions significatives.

Les tècniques de programació multifil ens poden ajudar a incrementar la capacitat de processament i també ens poden permetre donar servei a diversos usuaris a la vegada. Caldrà anar alerta, però, amb el nombre de fils a crear perquè cada fil gasta recursos de memòria i de processador i el sistema operatiu, a més de gestionar l'execució de l'aplicació, haurà de gestionar també la de cada fil, cosa que pot portar al col·lapse del sistema quan els recursos disponibles vagin acabant-se. Limitar el nombre màxim de fils pot resultar una solució acceptable, però si el nombre de clients simultanis supera massa el nombre de fils, deixarem clients sense atendre que rebran errors de timeout o de servei no disponible.

La distribució de la nostra aplicació pot també ser una solució quan cal mantenir un ús alt dels recursos i el nombre d'usuaris creix fins a desbordar la capacitat del servidor. Tot i així el seu desenvolupament incrementarà la complexitat i com a conseqüència el cost, tant de desenvolupament inicial com de manteniment posterior.

És important crear les aplicacions de la mida real que es necessitin, però dissenyar-les de tal manera que resulti possible escalar-les. Per aconseguir aquest objectiu serà necessari dividir l'aplicació en components independents susceptibles de ser modificats per incloure parcialment alguna de les millores esmentades i poder així fer créixer l'aplicació sense haver de canviar-ho tot.

2.6.1 Verificació del servei i monitorització dels components

Com podem detectar, però, la disponibilitat dels nostres serveis? Com podem saber quin component caldrà millorar? Aquesta no és una tasca fàcil perquè en una aplicació hi pot haver una gran quantitat de components i perquè la disponibilitat podria fallar només sota determinades circumstàncies de volum de treball o de nombre de connexions concurrents o d'altres qüestions similars que podrien enganyar-nos si al fer les proves de verificació del codi no es complissin les mateixes condicions.

Cal tenir en compte que a vegades pot donar-se el cas que un servei no es trobi disponible però no hagi caigut, sinó que aparentment tot es trobi en ordre. Necessitaríem una eina que ens pogués observar l'estat intern de la nostra aplicació; una utilitat que controli les errades que es puguin produir, que monitoritzi les condicions d'execució i que ens detecti possibles caigudes i pèrdues de servei.

Existeixen eines externes que poden informar-nos de l'estat de la màquina, ús de la memòria i del processador, nombre de fils concurrents, nombre de ports atesos, etc. Aquestes eines, a més, disposen de capacitat de fer saltar alarmes, activar processos o enviar correus d'avís, quan un servei deixa de funcionar o quan el sistema pren unes determinades condicions. Són eines força útils perquè fins i tot algunes d'elles, com per exemple Nagios o AppaDynamics, disposen de la capacitat de monitoritzar directament la màquina virtual de Java, usant la biblioteca JMX de monitorització del JDK. Això permet detallar molt la monitorització perquè podem decidir què s'ha d'observar exactament.

L'obtenció específica d'informació de la nostra aplicació caldrà programar-la i, per tant, preveure-ho durant el desenvolupament. Caldrà incloure sistemes de monitoratge i recollida de la informació allà on es consideri necessari. Citarem dos sistemes que ens poden resultar molt útils. El primer d'ells, els registres, que habitualment es coneixen com a *logs* i el segon, l'ús de classes de la bibliotex JMX que Java disposa per monitorar objectes concrets i obtenir en temps d'execució, les condicions en les quals són invocats.

Podeu trobar informació sobre el sistema Logger a http://docs.oracle.com/javase/7/docs/api/java/util/logging/Logger.html

El sistema de *logs* permet intercalar entre el codi instruccions que enviaran informació en un o més sistemes de sortida (habitualment solen ser fitxers i la pantalla) en funció d'un valor numèric que anomenem nivell. Java defineix al menys 7 rangs que tipifica com a `FINEST`, `FINER`, `FINE`, `CONFIG`, `INFO`, `WARNING` i `SEVER`. En executar una aplicació és possible assignar un valor inicial al nivell de manera que s'executin totes les instruccions *log* que tinguin associat un nivell igual o major. Malgrat que no s'acostuma a fer, es pot modificar el nivell inicial dinàmicament en temps d'execució de manera que en el nostre codi podem situar instruccions condicionals que alterin el comportament habitual dels registres. Per defecte el valor inicial de qualsevol execució és `SEVER`. Això permet fer servir el sistema com un registre d'errors, però tenim l'aplicació preparada per poder realitzar altres registres rebaixant el valor del nivell inicial.

Exemple que mostra una aplicació de monitoratge basada en logs

```
public void proccesClientRequest(Socket clientSocket){
        boolean farewellMessage=false;
        String clientMessage="";
        BufferedReader in=null;
        PrintStream out=null;

        Logger.getLogger("Nom de la classe").log(Level.INFO,
                                    "Entrada al mètode '
                                        proccesClientRequest'");

        try {
            in = new BufferedReader(new InputStreamReader(
                                                clientSocket.
                                                    getInputStream
                                                    ()));
            out= new PrintStream(clientSocket.getOutputStream());
            do{
                String dataToSend = "Sending ...";
                out.println(dataToSend);
                out.flush();
                clientMessage=in.readLine();
                farewellMessage = clientMessage!=null;
            }while((clientMessage)!=null && !farewellMessage);
        } catch (IOException ex) {
            Logger.getLogger("Nom de la classe").log(Level.SEVERE,

                                                "S'ha produit un
                                                    error.", ex);
        }

        Logger.getLogger("Nom de referència").log(Level.INFO,
                                    "Sortida del mètode '
                                        proccesClientRequest'");
}
```

La biblioteca destinada al monitoratge de Java (JMX) contempla diverses utilitats. Podem, per exemple, recollir els valors de l'estat de la màquina virtual i conèixer la quantitat de memòria gastada per la nostra aplicació, el temps que fa que està en execució, quan ha passat per darrer cop el `garbage collection`, quants fils s'han iniciat, quants queden encara actius, quants es troben en estat zombi, etc. Podem usar aquestes dades en combinació amb el sistema de *logs* per enregistrar aquesta informació quan es produeixi una errada, per exemple.

Exemple que mostra com fer servir la informació del sistema usant JMX

```
    ...
    } catch (IOException ex) {
        ThreadMXBean theradInfo = ManagementFactory.
            getThreadMXBean();
        MemoryMXBean memoInfo = ManagementFactory.getMemoryMXBean
            ();
        Logger.getLogger("ClassName").log(Level.SEVERE, "Meòria
            maxima:",
                                    memoInfo.getHeapMemoryUsage().
                                        getMax());
        Logger.getLogger("ClassName").log(Level.SEVERE, "Memòria
            usada:",
                                    memoInfo.getHeapMemoryUsage().
                                        getUsed());
        Logger.getLogger("ClassName").log(Level.SEVERE, "Nombre
            de fils:",
                                    theradInfo.getThreadCount());

```

```
12          Logger.getLogger("ClassName").log(Level.SEVERE,
13                                                "Error: "+ex.
                                                      getMessage(), ex)
                                                      ;
14      }
15      ...
```

Usant JMX també és possible monitoritzar un objecte específic per tal d'obtenir-ne les dades internes. Per fer-ho cal construir una classe de tipus `MXBean` o `MBean` en la qual codificarem el tractament que volem donar-li a la monitorització. L'avantatge que presenta aquesta darrera modalitat és que es poden connectar de forma remota una consola específica per monitorar a través de JMX, anomenada JConsole, en la qual podrem visualitzar les dades codificades. Per contra, direm que la codificació presenta certa complexitat amb la qual cal familiaritzar-se. Tot i així, és un sistema molt potent perquè permet la connexió d'eines externes com ara Nagios o AppaDynamics de forma que puguin actuar com a consola avançada amb totes les utilitats que aquestes eines posen a la nostra disposició.

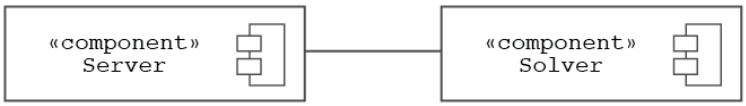
2.6.2 Disseny basat en components

Del disseny de components només donarem una visió molt general per fer-vos conscients de la importància que això suposa. Donarem també alguns consells que us ajudin en el moment de realitzar la planificació i el disseny. En el disseny de components, l'abstracció és fonamental. Trobem tres aspectes clau quan parlem de components: es tracta d'una **unitat** i per tant cal que presenti certa independència respecte els altres components; ha de ser reutilitzable en la mesura que sigui possible; i finalment hi ha components que formaran part d'altres components més grans.

Si analitzem en la seva globalitat la funcionalitat d'un servei distribuït, ens adonarem que a banda de les tasques específiques pròpies del problema que resol, el que habitualment es coneix com a lògica de negoci, trobarem moltes altres tasques comunes que es repeteixen sovint en gairebé tots els serveis. No sempre seran tasques implementades de forma idèntica, però sí que presentaran trets comuns.

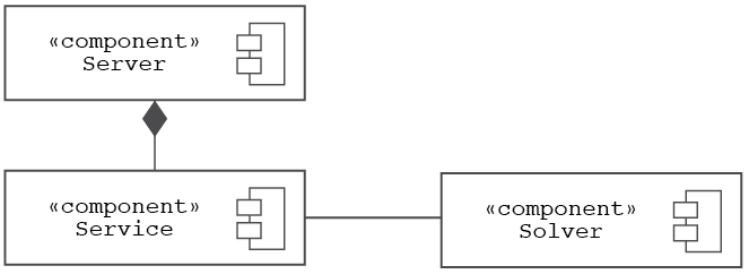
Una mica de disseny descendent ens ajudarà a descobrir els components i la seva funcionalitat. Partim dels components bàsics. Considerarem que en general un servei ha de tenir dos components: el component més específic encarregat de la "lògica de negoci" i el component que trobarem a qualsevol servidor encarregat d'escoltar el port assignat, gestionar les peticions que li arribin i derivar-les al component específic per tal d'obtenir una resposta i enviar-la al client (figura 2.14).

Figura 2.14. Components a la part servidor d'un servei I



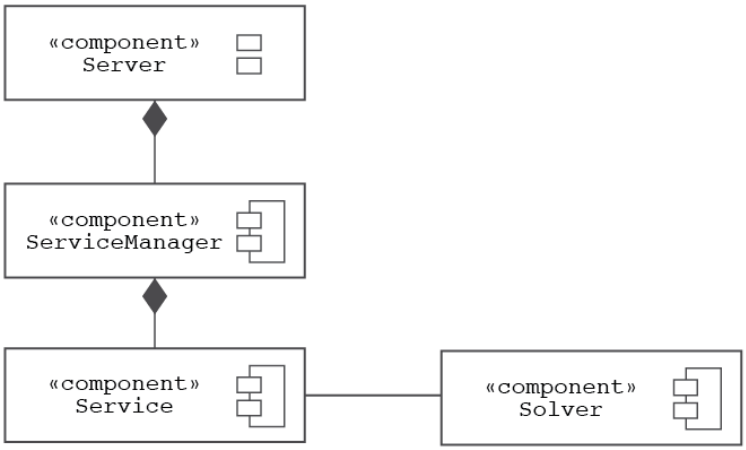
Aviat podem veure que el component que hem anomenat *Servidor* podria dividir-se en dos. L'un contenint l'altre. El servidor genèric atén el port a l'espera de qualsevol petició i un cop li arriba una haurà de resoldre-la. Podem entendre la necessitat de diferenciar entre l'escolta genèrica i l'atenció d'una petició concreta. Distingirem, doncs, un component servidor que contindrà un component específic per atendre cada petició que els clients enviïn, transformant els *inputs* dels clients en els paràmetres d'un mètode del component *Solver* que caldrà invocar per generar la resposta (vegeu la figura 2.15).

Figura 2.15. Components a la part servidor d'un servei II



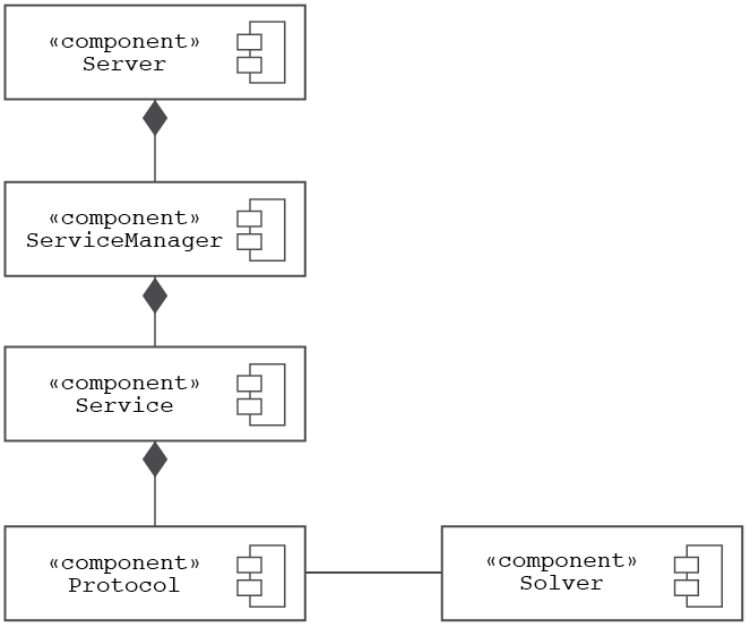
Davant la perspectiva d'haver de treballar amb diversos serveis a la vegada, podem intuir la necessitat de disposar d'una gestor de serveis sense necessitat de decantar-nos per cap solució a priori. Reconeixem la necessitat d'un gestor que acabarem implementant de maneres diverses segons la quantitat de peticions concurrents que calgui gestionar (figura 2.16).

Figura 2.16. Components a la part servidor d'un servei III



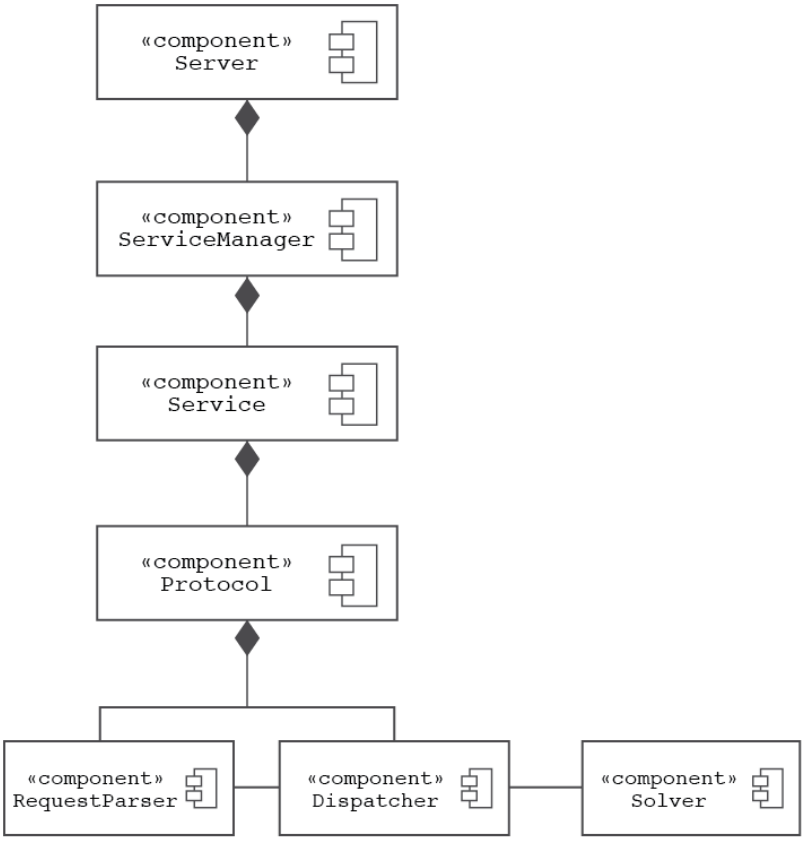
La complexitat de les peticions també ens pot suggerir que en ocasions haurem de definit un protocol i per tant poden suposar la necessitat de disposar d'un component que ens el gestioni (figura 2.17).

Figura 2.17. Components a la part servidor d'un servei IV



A més, la complexitat pot aconsellar d'afegir nous elements que ajudin a tractar de manera eficient la petició del client (component analitzador o `RequestParsers`) i la selecció del procés del *Solver* que caldrà invocar (component *Dispatcher*). Vegeu la figura 2.18.

Figura 2.18. Components a la part servidor d'un servei V



Què fem, però, amb tants components a l'hora d'implementar el servei? Caldrà implementar cada component en una classe independent? Podeu intuir que la resposta dependrà d'una implementació a l'altra. Hi haurà serveis en els quals es prevegi un ús escàs i amb unes peticions molt simples, però d'altres marcats per una gran concurrència de peticions, un temps de resposta gran, un joc de peticions complex o tot a la vegada. Sigui com sigui, però, cal entendre que tots ells, tinguin la mida que tinguin, realitzaran des d'una perspectiva abstracta, les mateixes funcions; no tots de la mateixa manera, és clar, en alguns casos la funcionalitat assignada a un dels components es podrà resoldre amb un únic mètode, en altres casos caldrà implementar tota una classe i a vegades serà necessari fer servir més d'una classe. A vegades caldrà gestionar fils i a vegades distribuir els processos, però en tots els serveis trobarem uns trets comuns que usarem per encaminar un disseny escalable.

Les interfícies ens ajuden a aconseguir el grau d'abstracció que necessitem permetent implementar amb una mateixa interfície solucions diferents. Darrere una interfície podem trobar fragments d'una classe, una classe sencera o fins i tot una biblioteca de classes que acabaran oferint la funcionalitat requerida amb la concreció necessària. Part de l'èxit pot passar, doncs, per crear interfícies prou versàtils per suportar diversos graus d'escalabilitat.

L'exemple que desenvolupem aquí és una simplificació d'un cas real en pro de la comprensió dels elements a tenir en compte per obtenir un disseny escalable. Es proposa una interfície que doni accés a cada component a excepció del component

Solver perquè no es poden conèixer a priori les seves necessitat ja que depèn de cada cas concret. Suposarem també que la comunicació està basada en caràcters.

```
// Fitxer "Server.java" especificant la funcionalitat del component Server.

/**
 * Aquesta interfície soporta la funcionalitat bàsica d'un servidor de
 * qualsevol servei.
 * @author Josep Cañellas <jcanell4@ioc.cat>
 */
public interface Server {

    /**
     * Aquets mètode preveu una manera de configurar el servidor. S'iniciarà
     * amb dos ports. El primer indica el port atès pel servei. El segon
     * informa del port utilitzat per fer el tancament del servidor. No
     * s'especifica com cal usar aquest port ja que dependrà de cada
     * implementació.
     * @param port és el port atès pel servei.
     * @param closePort és el port destinat al tancament del servidor
     */
    void init(int port, int closePort);

    /**
     * Comprova si la connexió acceptada és una petició per demanar el
     * tancament del servidor.
     * @param sòcol amb una connexió
     * @return cert només si el client demanava el tancament del servidor.
     */
    boolean isClosingSignal(Socket clieSocket);

    /**
     * Manté l'escolta del port PORT fins que rebi la instrucció
     * STOP. Cada cop que detecti la petició d'un client, generarà
     * un sòcol específic per a la comunicació amb aquest i l'iniciarà per tal
     * que la comunicació es faci factible.
     */
    void listen();

    /**
     * Tanca tots els serveis oberts. Tanca també el sòcol del client que ha
     * fet la petició de tancament del servidor i també el ServerSocket
     * principal.
     * @param closerSocket és el sòcol amb el qual ha arribat la petició de
     * tancament del servidor.
     */
    void close(Socket closerSocket);
}


// Fitxer "ServiceManager.java" especificant la funcionalitat del component
// ServiceManager.

/**
 * Es tracta d'una interfície que especifica la funcionalitat necessària per
 * crear i gestionar un conjunt de serveis concurrents.
 * @author Josep Cañellas <jcanell4@ioc.cat>
 */
public interface ServiceManager {
    /**
     * Permet crear un servei nou cada vegada que s'invoqui aquest mètode. En
     * cas que el servidor suporti concurrència mantindrà els serveis
         accessibles
     * per tal de poder-los gestionar.
     * @param service és el sòcol amb el servei a crear
     */
    public void generateService(Socket clientSocket);

    /**
     * Tanca tots els serveis actius
```

```
67      */
68     void closeServices();
69
70     /**
71      * Actualitza la llista de serveis actius, eliminant els que ja hagin
72      * finalitzat.
73      */
74     void updateServices();
75 }
76
77 // Fitxer "Service.java" especificant la funcionalitat del component Service.
78
79 /**
80  * La interfície representa un servei qualsevol. El servei contindrà la
81  * connexió (sòcol) específica amb un client i per tant disposa de mètodes per
82  * enviar i recollir missatges del client. També disposa de mètodes per
83  * realitzar el tancament de la connexió. El servei s'iniciarà amb el mètode
84  * processProtocol, que aplicarà el protocol específic que aquest servei
85  * tingui associat.
86  * @author Josep Cañellas <jcanell4@ioc.cat>
87  */
88 public interface Service {
89
90     /**
91      * Fa el tancament del sòcol específic del client. Tingueu en compte que
92      * per tal de poder atendre el tancament, caldrà invocar el shutdownInput
93      * del sòcol.
94      */
95     void close();
96
97     /**
98      * Informa de si el sòcol del client es troba obert o tancat retornant
99      * cert en cas que estigui tancat i fals en cas contrari.
100     * @return cert en cas que estigui tancat i fals en cas contrari
101     */
102    boolean isClosed();
103
104    /**
105     * Envia un missatge de tipus caràcter al servidor
106     * @param m és el missatge a enviar.
107     * @throws ServicesException en cas que es produeixi un error durant
108     * l'enviament
109     */
110    public void send(String m) throws ServicesException;
111
112    /**
113     * Rep un missatge acabat en un final de línia des del client. El mètode
114     * es bloqueja fins a obtenir el missatge del client o fins que el client
            es
115     * desconnecti. En cas de desconnexió del client es retornarà el valor null
            .
116     * @return el missatge rebut
117     * @throws ServicesException en cas que es produeixi un error durant
118     * la recepció.
119     */
120    public String readLine() throws ServicesException;
121
122 }
123
124 // Fitxer "Protocol.java" especificant la funcionalitat del component Protocol.
125
126 /**
127  * Aquesta interfíce representa el protocol a aplicar en qualsevol connexió
128  * client servidor. Diposa només d'un únic mètode per engegar el protocol.
129  * No hi ha més detall perquè cada protocol pot ser diferent i caldrà
130  * resoldre-ho durant la implementació.
131  * @author Josep Cañellas <jcanell4@ioc.cat>
132  */
133 public interface Protocol {
134    /**
```

```
135      * Inicia el protocol d'un servei.
136      * @throws ServicesException. En cas de qualsevol error durant l'execució
137      * del protocol es llançarà una excepció de tipus ServicesException
138      */
139     void processProtocol() throws ServicesException;
140 }
141
142 ...
```

Un tema també força important és el tractament de les excepcions. Seguint la màxima de no capturar cap excepció amb la qual no sapiguem què cal fer-hi, ens podem trobar quan fem abstraccions d'aquestes característiques, que cada implementació llanci excepcions diferents i que sovint calgui llançar excepcions de molt diverses jerarquies amb el conseqüent increment de complexitat quan sigui necessari fer el tractament.

Una possible solució passa per crear un excepció o una jerarquia comuna per a totes les implementacions, que permeti incloure l'excepció real generada per l'error en el moment de fer la instanciació. Això implica capturar sempre totes les possibles excepcions que es puguin produir, fer-ne el tractament que calgui en cas que sigui necessari i/o encapsular-la en una excepció nova específica de la jerarquia creada que llençarem tot seguit.

Exemple de captura d'excepcions per encapsular-les dins una nova excepció

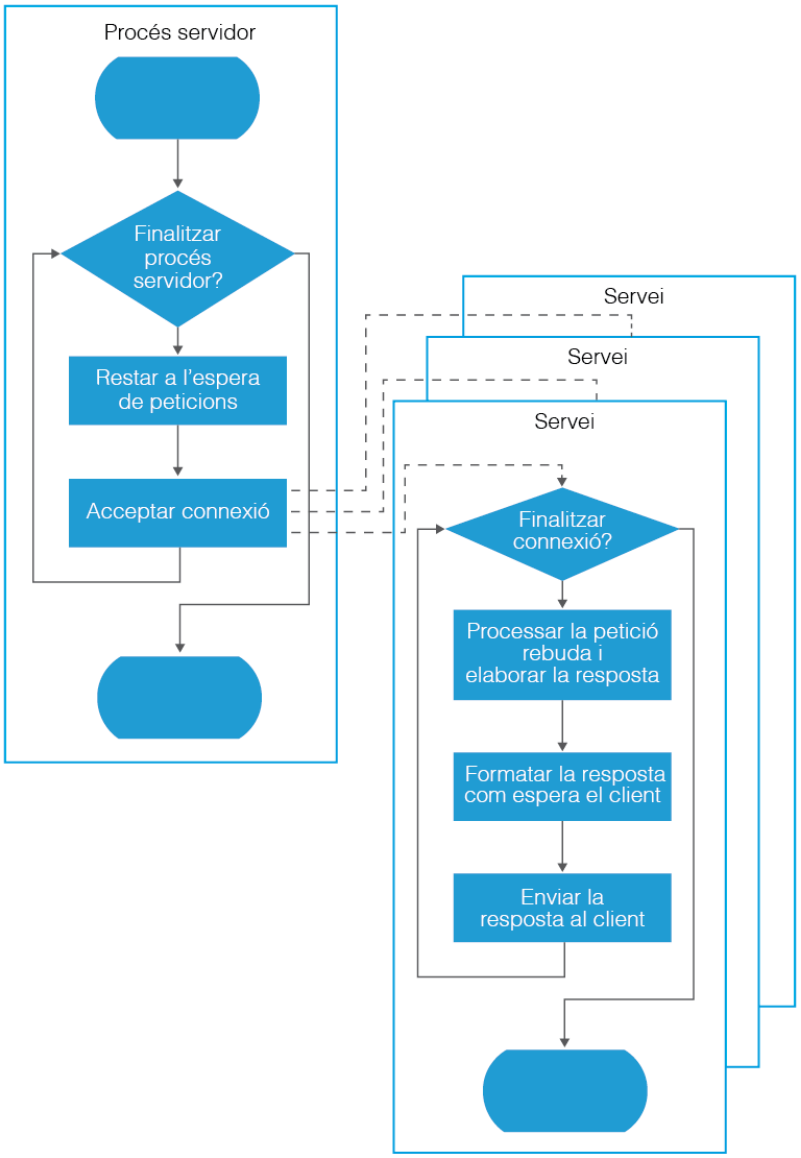
Imaginem que hem creat una excepció anomenada `ServicesException` i que tenim un mètode amb codi susceptible de llançar `IOException`. Una possible implementació pot ser:

```
1 public  void send(String missatge) throws ServicesException{
2     try {
3         out.write(missatge);
4         out.newLine();
5         out.flush();
6     } catch (IOException ex) {
7         throw new ServicesException(ex);
8     }
9 }
```

Suport de la concurrència de clients

Una de les solucions per aconseguir que un servidor pugui atendre diversos clients a la vegada consisteix en realitzar l'execució de cada client en un fil a part (vegeu la figura 2.19).

Figura 2.19. Esquema que il·lustra els processos durant l'execució d'un servidor concurrent



Es tracta d'un solució força senzilla si seguim els patrons dissenyats a les figures de figura 2.16 a figura 2.18. Prenent una implementació qualsevol d'un servidor TCP haurem d'afegir la funcionalitat específica del *ServiceManager* i també agrupar la corresponent al *Service* en una classe `Runnable` que es pugui executar en un fil apart.

La implementació de les interfícies no cal ubicar-la en classes independents. Cal valorar en cada cas l'esforç que representa cada alternativa i condicionar-hi la decisió. L'avantatge que representa l'implementació en classes diferents és que resultarà sempre molt mes fàcil fer-hi canvis. Tot i així, si prenem la precaució de seguir sempre les interfícies definides, fins i tot la divisió d'una classe en diverses pot resultat una tasca relativament senzilla i gens traumàtica.

La implementació del *ServiceManager* implica afegir una llista de serveis per poder gestionar-los quan calgui tancar-los com a atributs de la classe que implementi

la interfície. També caldrà resoldre els mètodes `addService`, `closeServices` i `updateServices`:

```
public class ... implements ServiceManager ... {
...
private List<Service> services = new ArrayList<Service>();

...

@Override
public void updateServices(){
    //Recorrem la llista i si hi ha serveis amb el sòcol tancat els
    //eliminem de la llista
    int i=services.size();
    --i;
    while(i>=0){
        if(services.get(i).isClosed()){
            services.remove(i);
        }
        --i;
    }
}

@Override
public void closeServices() {
    //Recorrem tots els serveis invocant el tancament
    for(int i=0; i<services.size(); i++){
        if(!services.get(i).isClosed()){
            services.get(i).close();
        }
    }
}

@Override
public void generateService(Socket clientSocket){
    //Es crea el servei i s'executa en un fil a part
    ServiceImpl service = new ServiceImpl(clientSocket);
    service.init();
    (new Thread(service)).start();
    addService(service);
}


private void addService(Service service) {
    //Afegim una nou servei a la llista i actualitzem la llista cada cop
    //que superem els 5 serveis suposadament actius
    services.add(service);
    if(services.size()>5){
        updateServices();
    }
}

...
```

Mantenint la funcionalitat del *ServiceManager* ben localitzada podem implementar noves solucions quan calgui. Per exemple si observem que el nombre de connexions concurrents augmenta fins a alentir el servidor, podem limitar el nombre de fils actius i emmagatzemar les peticions que desbordin la llista, en una cua de serveis pendents d'atendre. A mesura que vagin tancant els serveis es consultarà la cua i s'atendrà el primer servei que es trobi esperant. O potser si els temps d'espera fossin massa llargs caldria pensar en una sistema més distribuït en el qual l'atenció es pogués derivar a d'altres servidors especialitzats només en l'atenció dels serveis, etc.

El *Service*, si volem que suportar la concurrència, caldrà implementar-lo en una classe independent que implementi la interfície `Runnable`. Aquí suposarem que disposem d'una classe *Protocol* que gestionarà la comunicació client servidor.

```
public class ServiceImpl implements Runnable, Service{
    private BufferedReader in;
    private BufferedWriter out;
    private Socket clientSocket;
    private ProtocolImpl protocol = new ProtocolImpl();

    public ServiceImpl(Socket clientSocket) {
        this.clientSocket = clientSocket;
    }

    public void init(){
        protocol.init(this);
    }

    @Override
    public void run() {
        try {
            in = new BufferedReader(new InputStreamReader(clientSocket.
                getInputStream()));
            out= new BufferedWriter(new OutputStreamWriter(clientSocket.
                getOutputStream()));
            protocol.processProtocol();
        } catch (ServicesException ex) {
            Logger.getLogger("Service").log(Level.SEVERE, null, ex);
        }finally{
            if(!isClosed()){
                close();
            }
        }
    }

    @Override
    public boolean isClosed(){
        return clientSocket==null || clientSocket.isClosed();
    }

   @Override
   public void close(){
       synchronized(clientSocket){
           if(clientSocket!=null && !clientSocket.isClosed()){
               if(!clientSocket.isOutputShutdown()){
                   try {
                       clientSocket.shutdownOutput();
                   } catch (IOException ex) {
                       Logger.getLogger("Service").log(Level.SEVERE, null, ex);
                   }
               }
               if(!clientSocket.isInputShutdown()){
                   try {
                       clientSocket.shutdownInput();
                   } catch (IOException ex) {
                       Logger.getLogger("Service").log(Level.SEVERE, null, ex);
                   }
               }
               try {
                   clientSocket.close();
               } catch (IOException ex) {
                   Logger.getLogger("Service").log(Level.SEVERE, null, ex);
               }
           }
       }
    }

    @Override
    public  void send(String missatge) throws ServicesException{
```

```
64          try {
65              out.write(missatge);
66              out.newLine();
67              out.flush();
68          } catch (IOException ex) {
69              throw new ServicesException(ex);
70          }
71      }
72
73      @Override
74      public String readLine() throws ServicesException {
75          try {
76              return in.readLine();
77          } catch (IOException ex) {
78              throw new ServicesException(ex);
79          }
80      }
81  }
```

La resta d'implementacions, el *Protocol* i el *Solver* dependran força del tipus de servei que estem implementant, però si seguim uns criteris semblants als aplicats aquí, ens resultarà relativament fàcil poder fer modificacions a mesura que sigui necessari.

Seguretat i criptografia

Joan Arnedo Moreno

Programació de serveis i processos

Índex

Introducció **5**

Resultats d'aprenentatge **7**

1 Criptografia i Java **9**
1.1 Sistemes de xifrat simètric . . . 10
1.1.1 Generació de claus simètriques . . . 12
1.1.2 Claus simètriques basades en contrasenya . . . 13
1.1.3 Xifrat AES en mode ECB . . . 14
1.1.4 Xifrat AES en mode CBC . . . 17
1.2 Sistemes de xifrat asimètric . . . 20
1.2.1 Generació de claus privades i públiques . . . 21
1.2.2 Xifrat RSA directe . . . 22
1.2.3 Xifrat RSA amb clau embolcallada . . . 23
1.3 Firma digital . . . 25
1.3.1 Generació de firma digital RSA . . . 27
1.3.2 Validació de firma digital RSA . . . 27
1.3.3 Certificats digitals . . . 28
1.3.4 Emissió de certificats . . . 29
1.4 Gestió de claus . . . 31
1.4.1 L'eina Keytool . . . 32
1.4.2 Accés a un magatzem a través de codi Java . . . 33

2 Aplicacions Java segures a Internet **35**
2.1 Seguretat a la plataforma Java . . . 35
2.1.1 El gestor de seguretat . . . 37
2.1.2 Assignació de permisos . . . 38
2.1.3 Codi signat . . . 45
2.2 Connexions segures . . . 49
2.2.1 La negociació SSL . . . 50
2.2.2 Connexions SSL amb Java . . . 51
2.2.3 Gestió personalitzada de la informació criptogràfica . . . 57

Introducció

Les aplicacions informàtiques permeten gestionar i processar de manera molt eficient grans quantitats d'informació. Depenent del tipus d'aplicació, la informació a tractar pot tenir diferents propòsits i formats: potser són les propietats dels personatges d'un videojoc, d'un document de text enriquit o del conjunt de cel·les d'un full de càlcul. Hi ha tantes possibilitats com tipus d'aplicacions us poden venir al cap. Qualsevol tipus d'informació que puguem traduir a format digital és susceptible de ser tractada per un ordinador. Per acabar de posar la cirereta al pastís, l'auge d'Internet ha multiplicat enormement aquest avantatge, fent que qualsevol dada, no només sigui fàcil de processar, sinó també de ser intercanviada entre aplicacions, o accedida pels usuaris. Com a resultat, cada cop és més i més atractiu convertir tota la informació, ja sigui d'organitzacions o individus, al format digital, per facilitar el seu tractament i transport.

Ara bé, això obre una problemàtica molt important. Què succeeix quan algunes d'aquestes dades que tracten les aplicacions estan vinculades a aspectes sensibles, i no s'hauria de poder accedir a elles lliurement? Per exemple, les dades d'una declaració de la renda, la llista d'afiliats a un partit polític, o l'historial clínic d'un pacient. O, si anem a exemples menys extrems, simplement una fotografia una mica compromesa que heu pujat a una xarxa social, només per riure amb els vostres amics i ningú més. No pot ser que qualsevol persona dins l'organització que gestiona aquestes dades hi tingui lliure accés. Només les persones autoritzades haurien de poder accedir-hi. Idealment, ni tan sols un treballador amb accés físic a les màquines, com ara l'administrador de sistemes, hauria de poder veure-les si no està pertinentment autoritzat.

Com a futurs desenvolupadors d'aplicacions, aquesta és una problemàtica que haureu de tenir sempre molt present, ja que aquesta situació va més enllà d'una qüestió de confiança amb els vostres usuaris, sinó que fins i tot pot ser d'obligació per llei. Des de la seva implantació l'any 1999, la Llei Oficial de Protecció de Dades (LOPD) obliga a tractar amb un mínim grau de seguretat certs tipus d'informació i preveu multes de fins centenars de milers d'euros pels casos més graus en els quals no es faci satisfactòriament. Hi ha entorns on, simplement, no és possible implantar una aplicació sense garantir la seguretat de les dades tractades (com ara hospitals, ajuntaments, bancs, etc). Garantir la seguretat a les vostres aplicacions, avui en dia, més que un valor afegit és una necessitat. I una habilitat molt demandada. Per assolir-ho, en aquesta unitat s'estudia de quines eines disposeu quan programeu en Java per poder garantir la seguretat de les dades que tracteu.

A l'apartat "Seguretat i Criptografia" s'introdueix un concepte fonamental per poder garantir la seguretat de la informació dins d'una aplicació: l'ús de mecanismes criptogràfics. Si bé els seus principis són genèrics, i aplicables dins de qualsevol llenguatge de programació, la redacció se centra en com aplicar-la exclusivament

mitjançant Java. Concretament, estudia el funcionament de la biblioteca JCE (*Java Cryptography Extension*), que és la que ofereix el conjunt de classes que permeten aplicar transformacions criptogràfiques sobre unes dades.

Un cop establerts els coneixements bàsics sobre criptografia, aplicada a Java, l'apartat "Aplicacions Java segures a Internet" mostra com resoldre algunes de les problemàtiques més comunes a l'hora de crear programari que es distribueix o necessita connectar-se Internet. Per una banda, com garantir qui és el propietari d'una biblioteca que us heu descarregat i com garantir que no durà a terme tasques malicioses en el vostre equip. D'altra banda, veureu com establir connexions segures a servidors d'Internet.

Al llarg de la unitat didàctica se seguirà tot un conjunt d'exemples que permeten fer més fàcil la comprensió dels conceptes exposats. D'altra banda, per treballar-ne els continguts és convenient anar fent les activitats i els exercicis d'autoavaluació, a més de consultar la bibliografia bàsica proposada. Dins la disciplina de la programació, només la pràctica porta al domini dels conceptes exposats.

Resultats d'aprenentatge

En finalitzar aquesta unitat, l'alumne/a:

1. Protegeix les aplicacions i les dades definint i aplicant criteris de seguretat en l'accés, emmagatzematge i transmissió de la informació.

- Identifica i aplica principis i pràctiques de programació segura.
- Analitza les principals tècniques i pràctiques criptogràfiques.
- Defineix i implanta polítiques de seguretat per limitar i controlar l'accés dels usuaris a les aplicacions desenvolupades.
- Utilitza esquemes de seguretat basats en rols.
- Usa algorismes criptogràfics per protegir l'accés a la informació emmagatzemada.
- Identifica mètodes per assegurar la informació transmesa.
- Desenvolupa aplicacions que utilitzin sòcols segurs per a la transmissió d'informació.
- Depura i documenta les aplicacions desenvolupades.

1. Criptografia i Java

Avui en dia, les aplicacions informàtiques s'utilitzen per processar o, si més no, com a via d'accés, a una gran quantitat d'informació en format digital, cosa que fa que aquesta sigui cada cop més i més important. Penseu que sota el concepte *informació en format digital* es pot incloure des de la llista de clients i contactes d'una important multinacional fins el PIN, el número i la data de caducitat de la vostra tarja de crèdit.

Malauradament, això també vol dir que aquesta ha esdevingut un objectiu molt atractiu per entitats malicioses que volen treure'n profit a costa dels altres. Per aquest motiu, avui en dia és molt important garantir que les aplicacions gestionen les dades de manera segura. Afortunadament, poc a poc, els desenvolupadors de llenguatges de programació han cobrat consciència d'aquest fet i han anat aportant un seguit d'eines que us poden facilitar assolir aquest objectiu.

Estrictament parlant, la necessitat d'emmagatzemar o intercanviar informació de manera segura es remunta a temps immemorials. Des de sempre ha estat necessari intercanviar informació de manera que es garantís que cap altra persona implicada en el procés pogués interceptar-la i descobrir-la. Ja sigui un papir egipci amb una proposta d'aliança, una carta manuscrita entre dos amants a l'edat mitjana, les comunicacions de ràdio entre tropes militars en plena guerra mundial o un missatge de correu electrònic amb informació sensible per una multinacional en ple segle XXI. Des que els humans intercanviem missatges que aquesta problemàtica existeix. En alguns casos, cal garantir la privadesa de les dades.

La **privadesa de les dades** (*data privacy*, en anglès) és el servei de seguretat que garanteix que unes dades no puguin ser accedides lliurement. Només han de poder ser accedides per aquells individus que hi estiguin autoritzats.

D'entrada, el cas més fàcil de veure on caldria aplicar aquest servei és durant la transmissió d'un missatge. En el moment que el missatge deixa les mans de l'emissor i abans no arribin al receptor, aquest és susceptible de ser interceptat. En les comunicacions per Internet, el missatge es va desplaçant de manera que qualsevol part implicada en el seu recorregut pot fer-ne una còpia.

Al segle XVI, la reina d'Escòcia, Maria Estuard, va perdre el cap, literalment, al ser interceptades i desxifrades les seves cartes conspiratòries.

La privadesa també afecta a l'emmagatzematge de les dades. Per exemple, penseu en una aplicació d'emmagatzematge de fitxers al núvol. Les dades físiques, els fitxers en si, estan emmagatzemades en el disc dur d'un servidor en algun lloc del món. Per tant, qualsevol persona que pugui demanar l'accés a aquest servidor pot accedir també lliurement a les vostres dades. Tot el que deseu al núvol no és només vostre, sinó que també automàticament de l'empresa en la qual s'emmagatzema. Per exemple, això passaria a Dropbox.

El cas és que la privadesa de les dades s'hauria de poder garantir fins i tot en el cas que tercers tinguin a les seves mans el propi mitjà físic on estan escrites. Per ajudar-vos a resoldre aquest problema, teniu a la vostra disposició tota una branca de les matemàtiques: la criptografia.

La **criptografia** (escriptura secreta, del grec *kryptos*, amagat o secret, i *graphein*, escriptura) és la disciplina que s'encarrega d'estudiar com protegir la informació mitjançant la seva transformació, especialment mitjançant codis o xifrant.

El procés de **xifrat de dades** és el de transformar un conjunt de dades originals, les anomenades dades **en clar**, a un nou conjunt de dades totalment diferent, anomenades dades **xifrades**. Aquest resultat és totalment incomprensible i, idealment, té el mateix aspecte que si s'hagués generat totalment a l'atzar. Només el receptor disposa de les eines necessàries per recuperar les dades original. Aquest procés invers és el de **desxifrat**.

Per assolir aquesta fita, existeixen diferents **algorismes de xifrat**, o processos de transformació diferents. Cadascun té les seves característiques i es basa en l'aplicació de diferents algorismes de transformació sobre les dades originals. Java ofereix mecanismes per disposar alguns d'aquests algorismes, sense que els hàgiu d'implementar personalment.

El paquet **JCE (Java Cryptography Extension)** és una biblioteca que ofereix Java per poder executar diferents algorismes de xifrat als vostres programes.

L'objectiu de JCE és oferir una capa d'abstracció que permeti integrar diferents tipus d'algorismes criptogràfics, de manera que el codi sigui molt semblant o, si més no, només depengui d'uns pocs paràmetres al cridar alguns mètodes, independentment de l'algorisme triat. També està pensat perquè pugui ser estès per incloure nous algorismes que s'inventin en el futur mantenint la retrocompatibiltat de les aplicacions. Cal tenir en compte, però, que el JCE no disposa d'absolutament tots els algorismes existents a data d'avui.

1.1 Sistemes de xifrat simètric

El mecanisme de xifrat per antonomàsia, usat des de la invenció de la criptografia en temps pretèrits, és l'anomenat mecanisme de **clau simètrica**. En aquest, existeix un secret, la clau, que és compartida entre tots aquells que han de ser capaços de llegir les dades. Només qui en té possessió és capaç de xifrar-les i desxifrar-les correctament. Fent un símil planer, és com disposar d'un cofre indestructible, protegit amb un pany de tancament automàtic, on es pot desar un missatge. Només qui té una còpia de la clau del pany pot obrir el cofre i desar-hi

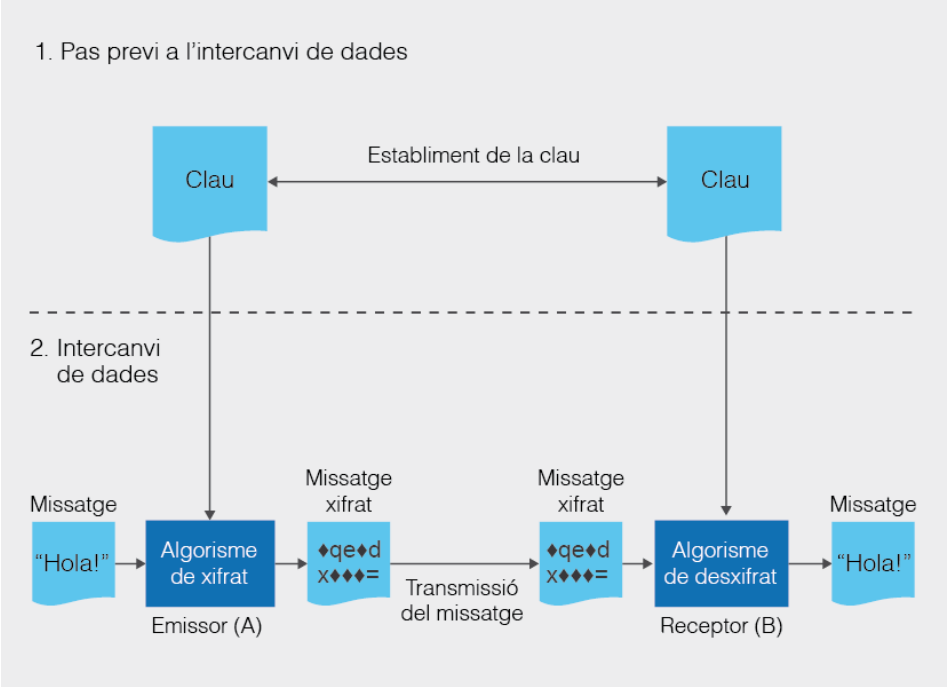
coses, o obrir-lo i recuperar el que hi ha a dins. Per tant, es donarà una còpia de la clau a qui es vulgui que pugui obrir el cofre. Encara que un intrús s'apoderi del cofre i se'l quedi, serà incapaç d'obtenir el seu contingut a menys que tingui la clau correcta.

En el cas d'un missatge de dades, és clar, no existeix tal cofre. La clau no és més que un conjunt de dades, també. En l'antiguitat, la clau era simplement una contrasenya en text, però en el cas d'un ordinador, pot ser qualsevol seqüència de bits. No cal ni que sigui representable com a text llegible. L'algorisme de xifrat combina les dades originals en clar amb la clau per generar com a resultat un conjunt de dades incomprensibles. Per recuperar el missatge original i desfer el procés d'ofuscació cal aplicar l'algorisme de desxifrat usant una còpia de la clau usada durant el procés de xifrat. Si no s'usa la clau correcta, el resultat seran també dades incomprensibles i, per tant, el missatge original romandrà protegit.

En el cas d'un intercanvi de missatges a dues bandes, abans de poder procedir a l'enviament de les dades, caldrà que les parts implicades acordin quina ha de ser la clau secreta. Normalment, la part interessada, ja sigui l'emissor o el receptor, és qui tria la clau i se la fa saber a l'altre. Ara bé, aquest pas és extremadament delicat, ja que cal tenir en compte que el procés d'enviament de la clau també és susceptible de ser interceptat. Per tant, cal fer-ho garantint que es duu a terme de manera segura. Això se sol fer en persona o mitjançant un missatger fiable.

Un esquema del mecanisme d'enviament de missatges xifrats usant clau simètrica es pot veure a la figura 1.1.

Figura 1.1. Esquema del procés de xifrat basat en clau simètrica



Al vostre dia a dia de ben segur que alguna vegada heu intercanviat de manera segura claus simètriques o contrasenyes compartides. Un cas típic és l'establiment de la contrasenya de la xarxa sense fils d'un encaminador ADSL, que s'envia en

un sobre tancat, en mà, per part d'un empleat de l'operador telefònic. Per resoldre aquesta problemàtica caldrà usar una mica d'imaginació.

Evidentment, en el cas d'usar aquest sistema per protegir les vostres dades d'ús totalment personal, vosaltres sou l'emissor i el receptor i, per tant, no cal amoïnar-se pel pas previ d'establiment de cap clau.

Abans de convertir-se en estàndard, l'algorisme AES es coneixia amb el nom de *Rijndael*.

Actualment, existeixen diferents algorismes de xifrat simètric en ús dins del context de les aplicacions informàtiques. Aquest apartat se centra en l'algorisme anomenat AES (*Advanced Encryption Standrad*), que és el que es considera més segur i l'estàndard en algorismes de xifrat des de l'any 2001. Aquest va ser creat per Joan Daemen i Vincent Rijmen i va guanyar el seu estatus d'estàndard després de participar en un concurs molt exigent a nivell internacional. Tot i així, cal tenir en compte que existeixen altres algorismes que encara estan en ús. Per exemple, l'anterior estàndard, el DES (*Data Encryption Standrard*) o el TripleDES. Les modificacions en el codi que cal fer per poder usar un algorisme o un altre mitjançant JCE són molt poques, fet pel qual centrar-se en només un és més que suficient per entendre com fer-ho amb qualsevol altre.

1.1.1 Generació de claus simètriques

Abans de plantejar-se l'execució d'un algorisme de xifrat, primer cal una condició indispensable: decidir quina serà la clau mitjançant la qual es protegirà la informació. Per tant, aquest és el primer pas a resoldre. Tothom qui tingui accés a aquesta clau serà capaç de generar dades xifrades i de desxifrar-les posteriorment.

Des del punt de vista d'un algorisme de xifrat que ha de ser executat per un ordinador, una clau simètrica, no és més que una seqüència aleatòria de bits d'una llargària determinada. Les llargàries vàlides vindran donades exclusivament pel tipus d'algorisme que es vol executar. A part d'això, no cal complir cap altra condició.

El JCE, però, no treballa normalment amb seqüències de bytes a l'hora de gestionar claus, sinó amb instàncies de la classe `javax.crypto.SecretKey`. Les biblioteques del JCE ofereixen una classe auxiliar capaç de generar per si mateixa claus segures per diferents tipus d'algorismes de xifrat. Es tracta de la classe `javax.crypto.KeyGenerator`. El seu ús es mostra en el següent bocí de codi:

L'algorisme AES només treballa amb claus de 128, 192 o 256 bits (a major mesura, més seguretat).

```
1  public SecretKey keygenKeyGeneration(int keySize) {
2      SecretKey sKey = null;
3      if ((keySize == 128)||(keySize == 192)||(keySize == 256)) {
4        try {
5          KeyGenerator kgen = KeyGenerator.getInstance("AES");
6          kgen.init(keySize);
7          sKey = kgen.generateKey();
8
9        } catch (NoSuchAlgorithmException ex) {
10         System.err.println("Generador no disponible.");
11       }
12     }
13     return sKey;
14   }
```

1.1.2 Claus simètriques basades en contrasenya

Una clau simètrica ha de poder ser intercanviada. Una possibilitat és desar-la en un fitxer o escriure els seus bytes, però això seria feixuc per les persones. Una manera de facilitar la gestió de claus criptogràfiques és generar-les a partir d'una contrasenya llegible. Que trieu una contrasenya fàcil o difícil d'endevinar ja és un altre tema.

Per fer-ho podeu aprofitar la particularitat que qualsevol seqüència de bytes de la mida correcta pot servir com una clau simètrica. Només cal, partint de la contrasenya, generar tants bytes com sigui necessari, però de manera que, donada una contrasenya, només aquesta generi una seqüència de bytes donada. I que mai sigui possible, o si més no extremadament improbable, que dues contrasenyes diferents arribin a generar la mateixa clau.

Un exemple de generació i compartició de clau amb contrasenya es troba a l'activació del xifrat als encaminadors ADSL domèstics.

Si hi penseu, aquesta tasca té certes dificultats d'entrada a menys que la contrasenya triada tingui exactament la mida de la clau desitjada. Si té més lletres que els bytes que calen, cal treure'n, i si és massa curta, cal afegir-ne. Haver d'usar una mida concreta i exacta pot ser una restricció una mica empipadora quan precisament el que es volia era facilitar la usabilitat (per una clau AES de 128 bits, 16 caràcters en alfabet llatí). Però si s'accepta qualsevol mida i després no es fan les coses amb traça, us trobareu que contrasenyes diferents poden arribar generar a la mateixa clau. Una manera correcta de resoldre aquest problema és usant un algorisme criptogràfic de *hash* sobre la contrasenya per generar la clau.

> Un **algorisme de *hash***, o resum (*digest*, en anglès), és una transformació criptogràfica que es pot aplicar a un conjunt de dades de manera que compleix sempre un seguit de condicions. Primer de tot, que la relació entre possibles entrades i sortides sigui bijectiva. O sigui, que donada una mateixa entrada, sempre resulti en exactament la mateixa sortida i que dues sortides diferents vinguin sempre d'entrades diferents. D'altra banda, no ha de ser reversible. A partir només d'un resultat, ha de ser pràcticament impossible endevinar quina ha estat l'entrada original que l'ha generat.

En l'actualitat, hi ha diversos algorismes de *hash*. Tot i així, el més utilitzat avui en dia, i el que es considera un estàndard, és l'algorisme SHA-1 (*Secure Hash Algoritm*, o Algorisme Segur de *Hash*). Aquest, a partir d'una entrada de dades de mida arbitrària, genera un resultat de 160 bits. Si es desitja una sortida de més longitud, existeix una versió millorada, més segura, tot i que no tan popular: el SHA-256. Tal com diu el seu nom, aquest genera 256 bits de sortida. De totes formes, des de l'any 2012, ja existeix el SHA-3, una nova versió encara millor i més segura.

Al JCE, la classe `MessageDigest` permet executar algorismes de *hash* sobre unes dades. Les instàncies d'aquesta classe s'obtenen invocant el mètode `getInstance`, que té com a paràmetre l'algorisme desitjat (per exemple `SHA-1`

o SHA-256). Caldrà triar-ne un que generi una sortida de com a mínim la mida de clau desitjada.

Un cop es disposa del resultat d'executar l'algorisme de *hash*, n'hi ha prou amb extreure'n tants bytes com es necessitin per generar la clau de la mateixa manera que si s'haguessin triat a l'atzar. El següent codi mostra com s'obtindria una clau simètrica AES a partir d'un text d'una contrasenya, amb l'ajut de l'algorisme SHA-256.

```
public SecretKey passwordKeyGeneration(String text, int keySize) {
    SecretKey sKey = null;
    if ((keySize == 128)||(keySize == 192)||(keySize == 256)) {
      try {
        byte[] data = text.getBytes("UTF-8");
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] hash = md.digest(data);
        byte[] key = Arrays.copyOf(hash, keySize/8);
        sKey = new SecretKeySpec(key, "AES");
      } catch (Exception ex) {
        System.err.println("Error generant la clau:" + ex);
      }
  }
    return sKey;
  }
```

1.1.3 Xifrat AES en mode ECB

En contraposició als algorismes de xifrat en bloc, existeixen els anomenats de xifrat en flux.

L'algorisme AES pertany a una família anomenada de **xifrat en bloc**. Això vol dir que l'algorisme no sap treballar directament amb dades de longitud arbitrària, sinó que només sap transformar bocins o blocs d'una mida molt concreta. Ni més gran ni més petit. Per tant, per poder transformar dades de qualsevol mida, el que es fa és dividir les dades d'entrada en blocs, de la mida adient, mantenint el seu ordre dins la seqüència. Llavors, l'algorisme processa cada bloc individualment, usant la clau completa, de manera que es van obtenint successius resultats parcials. El resultat final s'obté concatenant tots els resultats parcials en el mateix ordre que s'han anat processant per disposar de nou d'una única seqüència de bytes. Cada tipus d'algorisme treballa amb una mida de bloc diferent. Per exemple, l'algorisme AES té una mida de bloc de 16 bytes.

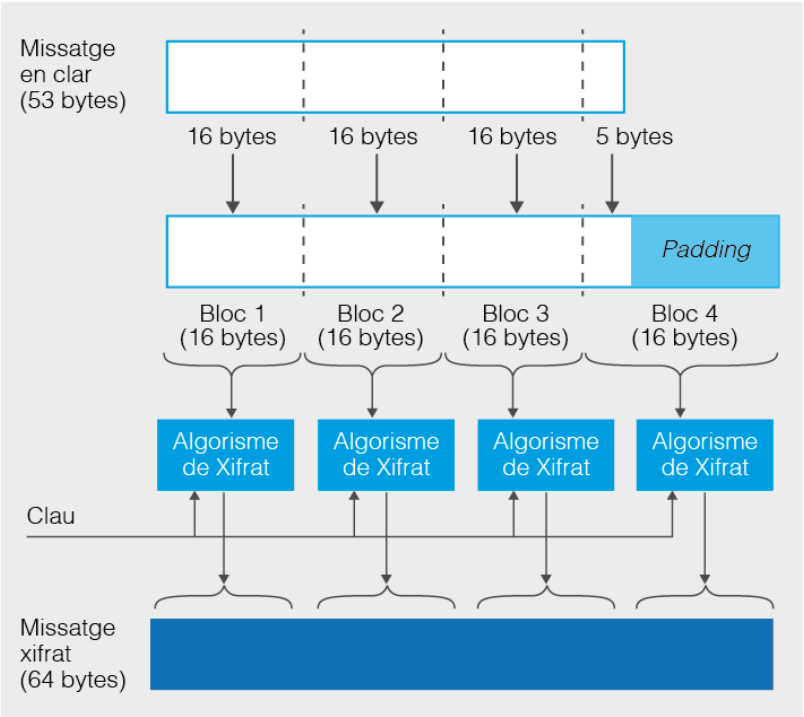
Al treballar d'aquesta manera, pot ser que la mida de les dades que es volen transformar no encaixi exactament amb un múltiple de la mida de bloc i, per tant, el darrer bocí no tingui suficients dades per arribar a la mida exacta d'un bloc. En el cas de l'AES, si l'entrada té 53 bytes, quedarà dividida en tres blocs de 16 bytes, i el darrer en tindrà només 5. Per arribar a la mida establerta, l'algorisme genera el que s'anomena un ***padding*** (un farcit) fins arribar a la mida de bloc. Això vol dir que s'afegeixen valors arbitraris, com poden ser zeros, a les dades que han quedat del darrer bloc. El *padding* simplement s'ignora en procés de desxifrat, però el seu format és un paràmetre que s'ha de subministrar a l'algorisme.

Pel cas de l'exemple d'una entrada amb 53 bytes, el darrer bloc tindria els 5 bytes finals de l'entrada, i tot seguit un *padding* d'11 bytes a 0, per assolir el

total necessari de 16 bytes. La figura 1.2 mostra un esquema gràfic d'aquest funcionament.

La mida total de les dades xifrades amb un algorisme de bloc sempre serà múltiple de la mida de bloc.

Figura 1.2. Esquema del procés de xifrat en mode ECB amb una mida de bloc de 16 bytes



Els algorismes de xifrat de bloc, a la vegada, poden funcionar principalment en dos modes d'operació: ECB (*Electronic Code Book*, Llibre de Codis Electrònic) i CBC (*Cyclic Block Chaining*, Encadenat de Blocs Cíclic). N'hi ha que en poden suportar més, però aquests són els més habituals.

El JCE ofereix la classe `Cipher`, que serveix tant per xifrar com per desxifrar dades. Com passa amb altres classes d'aquesta biblioteca, per obtenir una instància cal invocar el mètode estàtic `getInstance`. Aquest té com a paràmetre una cadena de text amb l'identificador de l'algorisme que es vol usar. Aquest identificador es composa sempre de tres valors separats per una barra (/):

- El nom de l'algorisme en majúscula. Per exemple, "AES".
- El mode d'operació, com ara "ECB" o "CBC".
- L'identificador del tipus de *padding*. N'hi ha de diferents, però se sol usar normalment "PKCS5PADDING".

Un exemple d'identificador d'algorisme podria ser "AES/ECB/PKCS5PADDING". A la documentació de la classe `Cipher` hi ha els tipus identificadors d'algorismes que és capaç d'acceptar.

Tant el procés de xifrat com el de desxifrat sempre segueix el mateix esquema:

1. Un cop instanciada la classe `Cipher`, s'inicia especificant si es vol usar per desxifrar o per xifrar i quina és la clau que cal usar. Això es fa amb el mètode `void init(int opmode, Key key)`. El primer paràmetre indica si es vol xifrar o desxifrar, triant una de les constants estàtiques `Cipher.ENCRYPT_MODE` o `Cipher.DECRYPT_MODE`. El segon paràmetre és la clau simètrica que voleu usar.

2. Si el conjunt de dades a processar no és gaire gran i es troben totes dins d'un *array* de bytes, es pot usar el mètode `byte[] doFinal(byte[] input)` per processar-les totes de cop. El resultat obtingut és el valor resultant. Tot el procés de tractament de blocs es fa de manera transparent dins de la instància de `Cipher`, vosaltres no us heu de preocupar de res.

3. També pot ser que no es disposi de totes les dades i que aquestes s'estiguin obtenint d'un flux d'entrada que requerirà successives lectures per captar tota la informació associada (per exemple, d'un `FileInputStream` si està tractant un fitxer). En aquest cas, s'ha d'usar un altre sistema. Caldrà fer crides successives al mètode `byte[] update(byte[] input)` amb els bocins de dades que es van llegint del flux, i anar desant els resultats parcials. Un cop s'hagin tractat totes les dades, cal finalitzar el procés amb la crida al mètode `byte[] doFinal()`, sense paràmetres, per obtenir el darrer bocí del resultat xifrat.

La instal·lació per defecte de l'entorn de desenvolupament de Java té un seguit de limitacions en la mida de les claus que es poden usar per xifrar dades (no hi ha cap limitació simplement per generar les claus). Això es deu a antigues polítiques restrictives d'exportació d'eines criptogràfiques fora dels Estats Units. Es consideraven armament militar. Si es volen usar claus AES de més de 128 bits, cal descarregar-se i instal·lar els fitxers de polítiques per criptografia forta: per la versió 7 de Java, és l'anomenat *Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files 7*. Aquests estan disponibles lliurement a la pàgina de descàrregues de Java. Abans de seguir, és molt recomanable que us l'instal·leu.

El següent bocí de codi mostra com usar la classe `Cipher` per xifrar dades desades dins un *array* de bytes usant l'algorisme AES. El tipus de *padding* triat és `PKCS5Padding`.

```
public byte[] encryptData(SecretKey sKey, byte[] data) {
    byte[] encryptedData = null;
    try {
      Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
      cipher.init(Cipher.ENCRYPT_MODE, sKey);
      encryptedData =  cipher.doFinal(data);
    } catch (Exception  ex) {
      System.err.println("Error xifrant les dades: " + ex);
    }
    return encryptedData;
  }
```

Per desxifrar les dades, el procés és exactament igual que el xifrat, amb la única diferència que el mode de l'objecte `Cipher` és `Cipher.DECRYPT_MODE`. En aquest cas, l'entrada seran les dades xifrades.

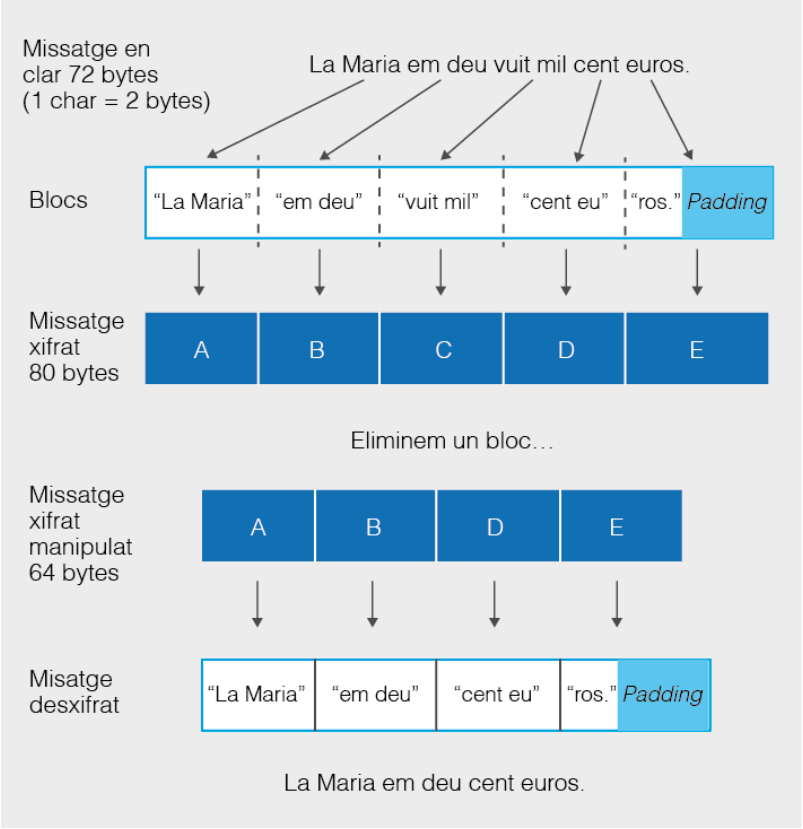
És important tenir present què passa quan s'intenten desxifrar unes dades amb una clau incorrecta. En aquest cas, els bytes resultants seran arbitraris i no tindran cap relació amb les dades originals en clar. A nivell d'un programa en Java, això pot dur a dues situacions diferents que cal controlar. El cas més habitual és que aquestes dades no tinguin un *padding* en el format adient i, per tant, el mètode `doFinal` falli, llançant una excepció del tipus `javax.crypto.BadPaddingException`. L'altra possibilitat, menys freqüent, és que, per pura casualitat, el resultat tingui un *padding* correcte. En aquest cas, el mètode s'executa correctament, però el resultat no té cap sentit. Per exemple, si s'havia xifrat un text llegible, el resultat són valors no representables textualment, o un text totalment incomprensible. És extremadament improbable, encara que no absolutament impossible, que unes dades desxifrades amb una clau incorrecta resultin en un text llegible, tot i que diferent de l'original.

1.1.4 Xifrat AES en mode CBC

El mode d'operació EBC és el més senzill d'usar per xifrar en bloc. Malauradament, la manera com processa les dades en clar per ser xifrades té un parell de problemes que el poden fer vulnerable a atacs si s'intercepten les dades protegides.

Per una banda, donat que cada bloc és totalment independent de l'altre i el resultat només és una concatenació de resultats parcials, l'atacant pot eliminar o reemplaçar blocs del text xifrat sense que sigui possible detectar, a priori, aquest fet. Per veure-ho millor, la figura 1.3 mostra un esquema d'aquest atac.

FIGURA 1.3. Esquema d'atac al mode ECB per modificar el significat d'un missatge



En aquest exemple se suposa una mida de bloc de 16 bytes.

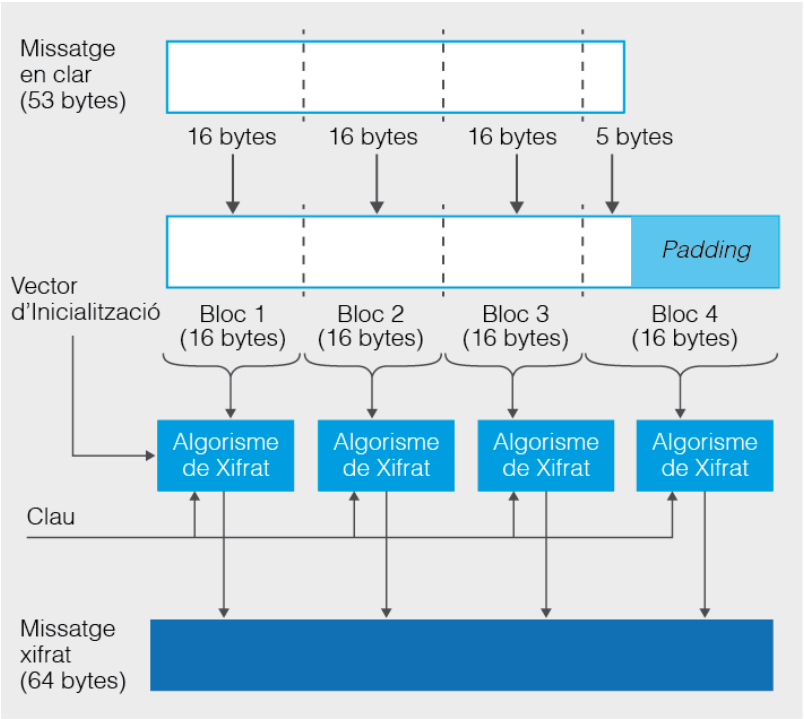
D'altra banda, i aquest és el problema més important, el mode ECB fa que, al dividir el text en clar en blocs, si apareixen blocs repetits a diversos llocs (que contenen exactament els mateixos bytes), el text xifrat també contindrà blocs repetits xifrats. Això pot semblar una minúcia sense importància, però, sense aprofundir més, cal fer palès que realment és per aquesta mena de detalls per on es comença a estudiar com trencar un missatge xifrat tot i no disposar de la clau.

La manera d'evitar aquest dos problemes és fer que els diferents blocs no es xifrin de manera totalment independent, sinó que el procés de xifrat depengui no només de la clau, sinó també del resultat de processar el bloc anterior. Aquest és el mode d'operació CBC.

A l'hora de programar en Java, l'única dificultat que cal superar per usar aquest mode d'operació és veure què fer amb el primer bloc, ja que no té cap bloc anterior. En aquest cas, el que se sol fer és generar un bloc especial anomenat **Vector d'Inicialització** (*Initialization Vector*, o IV per abreujar), que és el que cal aplicar pel primer bloc juntament amb la clau.

La Figura figura 1.4 resumeix el mode d'operació CBC a l'hora de xifrar els blocs i d'usar l'IV.

Figura 1.4. Esquema del procés de xifrat en mode CCB amb una mida de bloc de 16 bytes i un vector d'inicialització



Per veure com funciona un objecte `Cipher` quan es vol xifrar en mode CBC, el millor és un exemple. Si estudieu les instruccions Java del bocí de codi que es mostra tot seguit, podreu observar que el codi és molt semblant al mode ECB.

```
//Definició d'un IV estàtic. Per l'AES ha der ser de 16 bytes (un bloc)
  public static final byte[] IV_PARAM = {0x00, 0x01, 0x02, 0x03,
                                         0x04, 0x05, 0x06, 0x07,
                                         0x08, 0x09, 0x0A, 0x0B,
                                         0x0C, 0x0D, 0x0E, 0x0F};

  public byte[] encryptDataCBC(SecretKey sKey, byte[] data) {
    byte[] encryptedData = null;
    try {
      Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
      IvParameterSpec iv = new IvParameterSpec(IV_PARAM);
      cipher.init(Cipher.ENCRYPT_MODE, sKey, iv);
      encryptedData =  cipher.doFinal(data);
    } catch (Exception  ex) {
      System.err.println("Error xifrant les dades: " + ex);
    }
    return encryptedData;
  }
```

Al desxifrar caldrà garantir que el vector d'inicialització és exactament el mateix que es va usar per xifrar. En cas contrari, no obtindreu el text en clar original encara que la clau sigui correcta.

1.2 Sistemes de xifrat asimètric

El sistemes de xifrat simètric són molt eficients ja que, en el fons, no són més que l'aplicació directa als ordinadors dels principis de criptografia tradicional: compartir un secret entre les entitats que es comuniquen. Simplement aprofiten que els ordinadors poden realitzar moltes més operacions de càlcul en pocs instants. Per xifrar algunes dades amb l'AES usant només paper i llapis us hi estaríeu una bona estona!

Ara bé, els esquemes de xifrat simètric tenen un seguit de problemes. Per això, a finals del segle XX sorgeixen el sistemes de **clau asimètrica**. Aquests replantegen l'escenari del cofre protegit amb pany d'una manera certament innovadora. Imagineu-vos que aquest cofre ara té dos panys, cadascun depenent d'una clau totalment diferent. Una de les claus permet obrir una escletxa per on únicament es pot fer entrar el missatge, però no permet treure'l de dins. Aquesta serà l'anomenada **clau pública**. L'altre pany obre el cofre i permet treure els missatges desats a dins. Aquesta s'anomenarà la **clau privada**.

En aquest escenari, el receptor disposa d'una clau de cada tipus i s'encarrega de distribuir una còpia de la clau pública a tothom de qui vulgui rebre missatges secrets. L'avantatge fonamental és que si un espia intercepta una còpia de la clau pública, li servirà de ben poc. Només podrà desar missatges dins el cofre, però mai obrir el cofre, que és el que realment vol. Per tant, aquesta clau no cal que es transmeti de manera segura. De fet, com més fàcil sigui que el màxim de gent en tingui una còpia, millor i tot. Per això aquests sistemes també s'anomenen **de clau pública**.

Descrit a nivell general, en un sistema basat en clau pública cada usuari disposa d'un parell de claus: la seva clau privada i la seva clau pública. Cada usuari genera personalment el seu parell de claus d'acord a un procediment molt concret. Al contrari que el sistema de clau simètrica, aquestes dues claus han de complir unes condicions matemàtiques molt especials, que no s'entrarà a explicar. El resultat és que no val qualsevol seqüència de bits per generar-les, el procés és força més complex i una mica més lent. Llavors, cadascú s'encarrega de distribuir a tothom la seva clau pública, sense haver de preocupar-se de si algú intercepta la transmissió. Per exemple, la pot penjar a la seva plana web. La clau privada se la guarda de manera segura, de manera que ningú altre la sàpiga mai. Cada clau privada només la coneix el seu creador i ningú més.

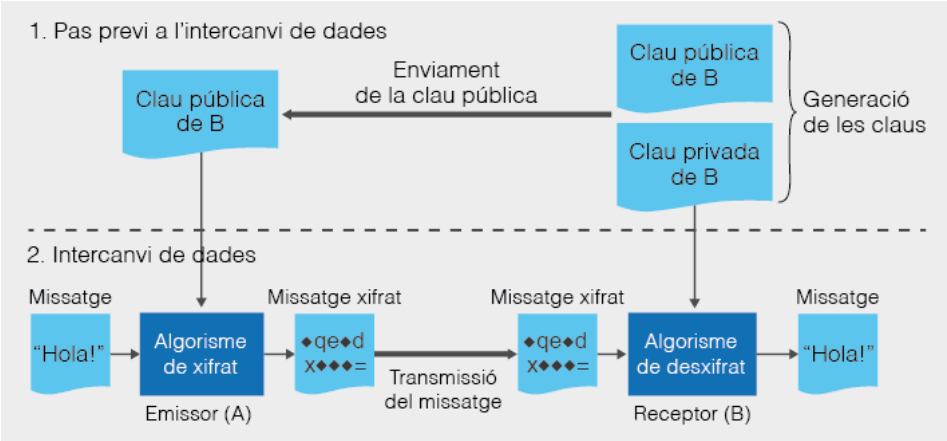
La figura 1.5 mostra l'esquema del mecanisme d'enviament de missatges xifrats usant clau pública.

Els creadors de l'RSA també són fundadors d'una empresa de seguretat molt important a nivell mundial amb el mateix nom.

Tal com passa amb els algorismes de clau simètrica, actualment existeixen diversos algorismes de clau pública. En aquest apartat s'usa com a fil argumental el que es pot anomenar actualment l'estàndard *de facto*, l'RSA. Aquest algorisme va ser publicat l'any 1977 i el seu nom prové de les inicials dels cognoms dels seus creadors, Ron Rivest, Adi Shamir i Leonard Adleman. A nivell d'implementacions pràctiques a gran escala, ara mateix és l´únic que s'utilitza.

De ben segur que vosaltres ja heu fet ús de sistemes basats en aquest algorisme algun cop, ja que és el que garanteix la seguretat a les pàgines web segures, com ara la banca en línia o les botigues virtuals.

Figura 1.5. Esquema del procés de xifrat basat en clau asimètrica o pública



1.2.1 Generació de claus privades i públiques

Al contrari que amb les claus simètriques, els parells de claus asimètriques no són un conjunt arbitrari de bytes que es poden generar de qualsevol manera. Donades les particularitats del xifrat asimètric, perquè el sistema funcioni, tant la clau pública com la clau privada han de tenir un conjunt de propietats matemàtiques molt concretes. Si no es compleixen aquestes propietats, o no funcionarà l'algorisme, o serà molt fàcil d'endevinar la clau privada a partir de la clau pública. Aquestes propietats varien depenent de l'algorisme.

Un nombre primer és aquell que només és divisible exactament, donant la resta de la divisió 0, entre ell mateix o 1.

Sense entrar en detall en els motius matemàtics, pel cas de l'RSA, la clau privada es genera a partir del producte de dos valors enters primers molt grans. La magnitud d'aquest *molt gran* depèn de la mida de la clau que es cerca. Pel cas d'una clau RSA de 2.048 bits, que és el mínim considerat segur avui en dia, això vol dir un valor que estigui entre 2^2.047 i 2^2.047, ni més gran ni més petit. O sigui, un enter codificat usant 256 **bytes** (no bits). Per donar certa perspectiva, el tipus primitiu enter més gran del Java, el `long`, ocupa 8 bytes.

Afortunadament, el JCE ja proporciona la classe `KeyPairGenerator` que permet generar automàticament parells de claus correctes, de manera segura. A menys que us interessi la base matemàtica de l'RSA, coneixent aquesta classe és més que suficient. Per usar-la, n'hi ha prou amb saber quin algorisme de xifrat es voldrà usar i la mida de la clau en bits.

Tot seguit es mostra un bocí de codi d'exemple sobre com usar aquesta classe.

```
public KeyPair randomGenerate(int len) {
    KeyPair keys = null;
    try {
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
      keyGen.initialize(len);
      keys = keyGen.genKeyPair();
    } catch (Exception ex) {
      System.err.println("Generador no disponible.");
    }
    return keys;
  }
```

Els parells de claus es representen al JCE amb la classe `KeyPair`. Aquesta classe proporciona els mètodes `getPrivate()` i `getPublic()` per obtenir la clau privada i pública continguda, respectivament. En aquest sentit, les claus privades es representen amb la classe `PrivateKey` i les públiques amb la classe `PublicKey`.

1.2.2 Xifrat RSA directe

El sistema per xifrar i desxifrar mitjançant algorismes de clau pública que ofereix JCE és idèntic a l'emprat per clau simètrica. També es basa en la mateixa classe `Cipher`. Només cal anar amb una mica de compte amb el fet que ara, per xifrar, cal usar la clau pública com a paràmetre i, al desxifrar, la clau privada. Si us confoneu i useu una enlloc de l'altra on no correspon, l'algorisme no funcionarà.

El següent bloc de codi mostra com xifrar dades usant l'algorisme RSA en mode ECB i amb un format de *padding* `PKCS1Padding`, que és el que s'usa en el criptosistema RSA.

```
public byte[] encryptData(byte[] data, PublicKey pub) {
    byte[] encryptedData = null;
    try {
      Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding","SunJCE");
      cipher.init(Cipher.ENCRYPT_MODE, pub);
      encryptedData =  cipher.doFinal(data);
    } catch (Exception  ex) {
      System.err.println("Error xifrant: " + ex);
    }
    return encryptedData;
  }
```

Cal dir que, tot i que cal declarar un mode d'operació i un *padding* als paràmetres d'inicialització de la classe `Cipher`, la manera com aquests afecten l'algorisme no té res a veure amb el cas de clau simètrica. Tenen un significat diferent.

Per desxifrar unes dades, el procés és pràcticament igual, però caldrà usar la clau privada al inicialitzar `Cipher` en mode `Cipher.DECRYPT_MODE`.

1.2.3 Xifrat RSA amb clau embolcallada

Malauradament, el sistema de xifrat RSA té un petit parany en el qual és fàcil caure si no es coneixen els principis matemàtics amb els quals opera. Sense entrar en detall, n'hi ha prou amb saber que la mida de les dades que es poden xifrar està fitada per la mida d'un dels components de la clau privada, anomenat el seu *mòdul*. La mida d'aquest mòdul és el nombre de bytes que cal per representar la mida de la clau. Per exemple, una clau de 1.024 bits té un mòdul de 128 bytes, mentre que una clau de 2.048 en té un de 256 bytes.

Els 11 bytes que es resten a la fórmula es corresponen als que calen per generar el *padding*.

Concretament, l'algorisme RSA només pot ser usat per xifrar dades de longitud **(mida clau RSA)/8 - 11**.

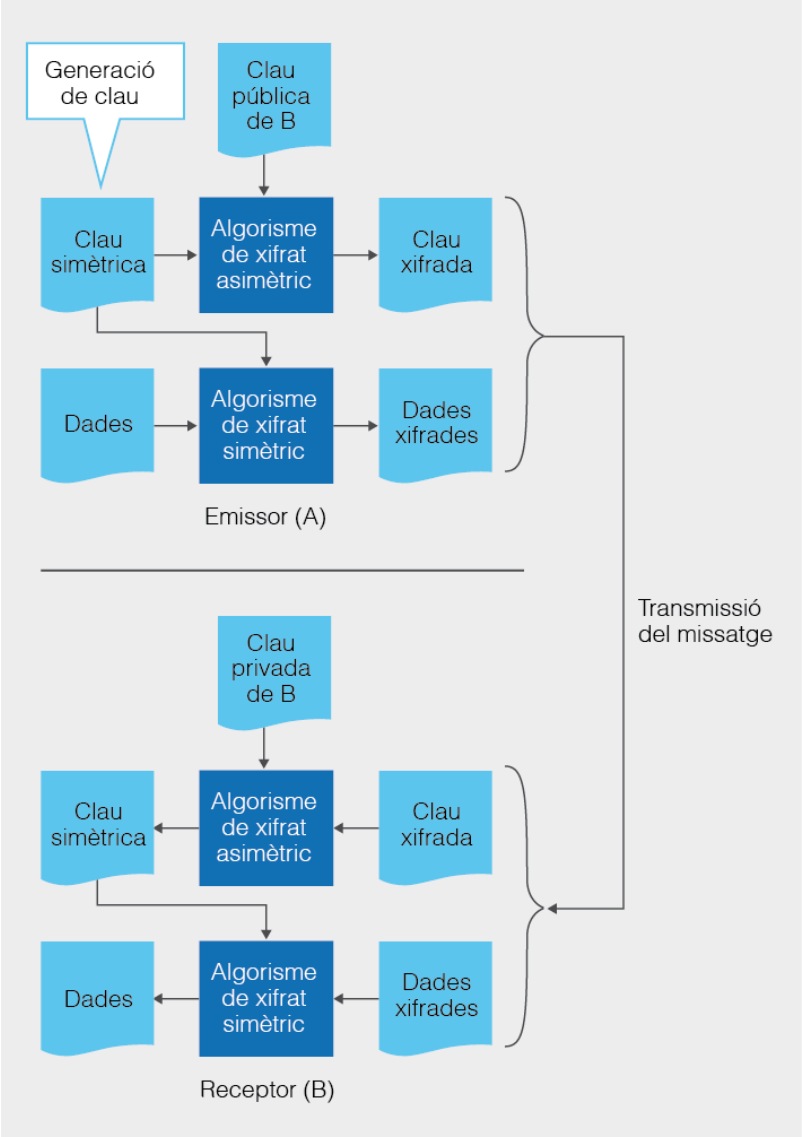
Per tant, una clau de 2.048 bits només pot ser usada per xifrar fins a 245 bytes. Si les dades tenen una mida superior, aquesta es trunca a 245 bytes i la resta es perden. Aquests bytes truncats mai es podran recuperar a l'aplicar l'algorisme de desxifrat.

Evidentment, aquesta és una limitació molt greu, ja que 245 bytes són molt pocs per la mena de dades que es processen avui en dia. Una solució podria ser usar simplement una clau més gran. Però per dades grans, la clau seria enorme, tan gran o més que les pròpies dades. A més a més, l'RSA és un algorisme lent, en comparació amb un sistema de clau simètrica, i com més gran és la clau, molt més lent es torna. Amb la tecnologia actual és inviable. La solució de compromís per resoldre aquest problema és una aproximació híbrida, que combina xifrat asimètric amb xifrat simètric.

En un sistema de **clau embolcallada** (*wrapped key* en anglès), les dades es xifren usant una clau simètrica d'un sol ús, generada a l'atzar. Aquesta clau llavors es xifra usant la clau pública del destinatari del missatge. Finalment, s'envia al destinatari el missatge i la clau xifrades, conjuntament.

A causa d'aquest sistema, el destinatari el que ha de fer per accedir a les dades és desxifrar la clau simètrica amb la seva clau privada i llavors, un cop la té, usar-la per desxifrar el missatge original. Per tant, cal fer dos processos de xifrat i desxifrat, un usant un algorisme de clau simètrica i un altre de clau pública. Tot el procés es mostra en l'esquema a la figura 1.6.

Figura 1.6. Enviament de missatges xifrats usant el sistema de clau embolcallada



El següent codi us mostra un exemple de com fer-ho mitjançant la biblioteca JCE. Com podeu veure, tot plegat és una fusió de la part de xifrat simètric i xifrat asimètric. L'única particularitat amb la qual cal anar amb compte és en la inicialització i ús de la classe `Cipher` a l'embolcallar o desembolcallar la clau simètrica:

- El mode d'inicialització passa a ser `WRAP_MODE` o `UNWRAP_MODE`, enlloc de `ENCRYPT_MODE` o `DECRYPT_MODE`.
- Els mètodes per xifrar i desxifrar la clau que cal usar són `wrap` i `unwrap`, enlloc de `doFinal` o `update`.

```
public byte[][] encryptWrappedData(byte[] data, PublicKey pub) {
    byte[][] encWrappedData = new byte[2][];
    try {
      KeyGenerator kgen = KeyGenerator.getInstance("AES");
      kgen.init(128);
      SecretKey sKey = kgen.generateKey();
      Cipher cipher = Cipher.getInstance("AES");
      cipher.init(Cipher.ENCRYPT_MODE, sKey);
```

```
9       byte[] encMsg = cipher.doFinal(data);
10      cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding");
11      cipher.init(Cipher.WRAP_MODE, pub);
12      byte[] encKey = cipher.wrap(sKey);
13      encWrappedData[0] = encMsg;
14      encWrappedData[1] = encKey;
15    } catch (Exception  ex) {
16      System.err.println("Ha succeït un error xifrant: " + ex);
17    }
18    return encWrappedData;
19  }
```

Per desxifrar només cal fer el procés invers.

1.3 Firma digital

Inicialment, l'ús dels termes *criptografia* i *xifrat* eren pràcticament intercanviables. L'aplicació d'aquesta disciplina tradicionalment ha estat per assolir la privadesa en l'intercanvi de missatges. Però amb la popularitat d'Internet i la proliferació en l'intercanvi telemàtic de missatges o documents, es fa patent la necessitat de disposar d'altres serveis de seguretat més enllà del de la privadesa. Principalment, els següents:

- **Integritat**: poder identificar si un document ha estat manipulat. Cal fer palès que aquest servei no evita la manipulació, només fa possible que sempre pugui ser detectada pel receptor.
- **Autenticació**: poder garantir quina és la identitat de l'autor del document, evitant que sigui suplantat.
- **No-repudi**: evitar que l'autor pugui negar que ell ha generat el document. El receptor pot demostrar a un tercer la identitat de qui ha emès realment el missatge.

Aquests serveis, si es contemplen tots tres junts, són els que us permeten dur a terme intercanvis fiables de dades, com ara realitzar transaccions o la venda de productes, fer la declaració de la renda telemàticament o connectar-vos a la banca en línia. També permeten garantir que si s'emmagatzema un document a un ordinador, ningú l'haurà modificat quan es torni a obrir sense que la manipulació no es faci patent.

Els grafòlegs són els encarregats de validar una firma manuscrita.

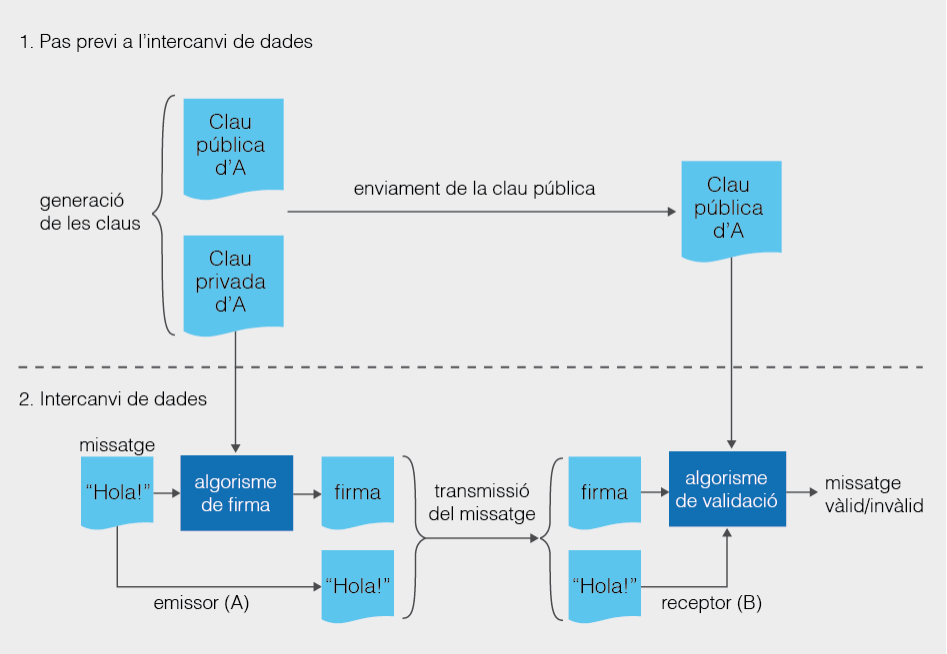
Quan les comunicacions a distància es feien majoritàriament per escrit, garantir aquests punts es considerava relativament simple mitjançant la firma manuscrita. Amb la digitalització de documents i la seva transmissió telemàtica, això ja es converteix en una tasca molt més complicada. Aquest cop no és possible traslladar directament el mecanisme que s'usa en paper. Els mecanismes de xifrat amb ordinadors tampoc no permeten obtenir exactament les mateixes propietats que una firma manuscrita. Caldrà un sistema diferent. La solució la teniu al vostre abast, de nou, gràcies a l'ajut d'una tècnica criptogràfica.

La **firma digital** és el mecanisme criptogràfic que trasllada totes les propietats de la firma manuscrita del món físic a la informació en format digital.

La firma digital imita la firma manuscrita en documents.

Els sistemes de firma digital normalment es basen en un esquema de claus asimètriques, d'acord al procés que es mostra a la figura 1.7. En aquest s'inverteixen els rols de clau pública i privada. L'algorisme criptogràfic de firma emprat en l'emissor es basa en la seva clau privada, que només té ell i ningú, mentre que el procés de validació, en canvi, es basa en la seva clau pública, que pot tenir tothom. També és important remarcar que, al contrari que en el procés de xifrat, el resultat de la firma no reemplaça les dades originals. L'emissor envia les dades i la firma conjuntament, igual que en un document manuscrit signat la firma no reemplaça el seu contingut, sinó que n'és una addenda.

Figura 1.7. Esquema de firma digital basat en clau asimètrica



Perquè la validació de la firma sigui correcta a l'extrem del receptor, s'han de complir les següents condicions, cadascuna associada a un dels seus paràmetres d'entrada. Si qualsevol d'aquest tres paràmetres no és el que correspon a l'extrem de l'emissor, l'algorisme de validació indica que la firma no és vàlida.

L'algorisme de validació només detecta si un document i la seva firma són vàlids o no (cert o fals). Si és invàlid, no pot indicar-ne el motiu exacte.

- Les dades rebudes han de ser exactament les mateixes que les que es van usar a l'executar l'algorisme de firma.
- La firma pròpiament tampoc ha estat modificada.
- La clau pública usada per la validació ha de ser exactament l'associada a la clau privada usada en la generació de la firma. Si es valida amb una altra clau pública, l'algorisme dirà que la firma no és vàlida. Per tant, una firma vàlida només es pot haver generat amb una clau privada concreta, cosa que identifica unívocament al seu propietari com el signant.

Actualment, existeixen diferents algorismes de firma digital. Aquest apartat se centra en el sistema de firma RSA, aprofitant que ja s'ha explicat el sistema de xifrat amb el mateix nom. Es tracta del sistema actualment més popular i el més fàcil d'usar dins de la JCE de Java.

1.3.1 Generació de firma digital RSA

El punt de partida de la firma RSA és idèntic al del xifrat. Existeixen dues claus, pública i privada, que són exactament les mateixes que es poden usar per xifrar les dades. En aquest sentit, no hi ha absolutament cap canvi. Unes claus generades per xifrar es poden usar també per signar, i viceversa. Per tant, les clau RSA poden usar-se per totes dues coses, cosa que resulta un gran avantatge.

La classe responsable de gestionar una firma digital és `Signature`. Si ja se sap com xifrar dades no és gaire difícil d'usar, ja que el procés de firma és molt semblant. El següent codi d'exemple mostra aquest procés.

```
public byte[] signData(byte[] data, PrivateKey priv) {
    byte[] signature = null;

    try {
      Signature signer = Signature.getInstance("SHA1withRSA");
      signer.initSign(priv);
      signer.update(data);
      signature = signer.sign();
    } catch (Exception ex) {
      System.err.println("Error signant les dades: " + ex);
    }
    return signature;
  }
```

Fixeu-vos que els noms dels algorismes de firma solen tenir el format "XXXwithYYY", de manera que són fàcils d'identificar a simple vista.

1.3.2 Validació de firma digital RSA

El procés de validació també es duu a terme amb la classe `Signature` i és molt semblant al de firma. Només cal recordar les seves particularitats: que ara s'usa la clau pública del signant per validar i que, a més a més de les dades originals, un dels paràmetres és la firma.

El següent exemple mostra com validar unes dades amb la seva firma. Fixeu-vos que ara cal cridar els mètodes `initVerify` per inicialitzar el procés amb la clau pública que cal usar i `verify` per executar la validació de la firma pròpiament.

```
public boolean validateSignature(byte[] data, byte[] signature, PublicKey pub)
     {
    boolean isValid = false;
    try {
      Signature signer = Signature.getInstance("SHA1withRSA");
```

```
5        signer.initVerify(pub);
6        signer.update(data);
7        isValid = signer.verify(signature);
8      } catch (Exception ex) {
9        System.err.println("Error validant les dades: " + ex);
10     }
11     return isValid;
12   }
```

1.3.3 Certificats digitals

Un aspecte fonamental de la criptografia asimètrica, fora de la implementació de l'algorisme, és la distribució i gestió de la clau pública. Per poder fer un intercanvi segur de missatges, cal que totes les parts implicades disposin de les claus públiques de la resta. Al contrari que en el cas de les claus simètriques, no és necessari dur a terme aquest intercanvi en secret, ja que disposar de la clau pública d'un altre no posa en perill les comunicacions.

Ara bé, a l'hora de gestionar claus públiques, caldrà establir algun sistema per distingir clarament, però de manera senzilla, la identitat del seu propietari. És per aquest motiu que les claus públiques normalment no se solen gestionar directament, agafant la seva seqüència de bytes tal com s'han generat. Aquestes s'inclouen en una estructura de dades que, a part de la clau, conté un conjunt d'informació addicional. El que s'anomena un certificat digital.

> Un **certificat digital** és un document electrònic que estableix la identitat del propietari d'una clau pública.

Si es vol fer un símil, en certa manera, un certificat digital és com un carnet d'identitat. Però en lloc d'associar un nom o identitat a una cara, el que fa és associar-lo a una clau pública, i la clau privada associada.

La definició completa de l'estàndard x.509 es pot trobar a http://www.ietf.org/rfc/rfc2459.txt.

Els certificats digitals més usats segueixen l'anomenat estàndard X.509 per estructurar tota la informació continguda. Aquest estàndard és complex i no val la pena explicar-lo fins al darrer detall. Els continguts més importants per entendre què és un certificat digital són els següents:

- La clau pública del seu propietari
- La identitat del propietari (anomenat també *Subject*)
- La identitat de l'emissor del certificat (anomenat també *Issuer*)
- La seva data de caducitat
- Una firma digital, generada per l'emissor, de tota la informació continguda

Les identitats que conté un certificat digital, tant del propietari com de l'emissor, es codifiquen mitjançant un **nom distingit** (o *Distinguished Name*, en anglès, també

abreujat com a DN). Aquest s'estructura a partir d'una seqüència molt concreta de components, d'acord a la taula 1.1. Aquests s'indiquen en el mateix ordre que a la taula.

Taula 1.1. Components d'un nom distingit (DN, Distinguised Name)

Acrònim	Nom llarg	Descripció
CN	*Common Name*	Nom i cognoms si és una persona. Nom DNS si és una màquina.
O	*Organization*	Nom de l'organització
OU	*Organizational Unit*	Nom de la unitat estructural
L	*Localty*	Localitat i ciutat
ST	*State*	Província o estat (dels EEUU)
CO	*Country*	Dues lletres inicials del país (en anglès)

Val la pena remarcar un dels elements més importants d'un certificat digital: la firma digital de tot el seu contingut. De fet, aquest element és el que dóna validesa al certificat com a tal, ja que, tractant-se d'un document electrònic, és susceptible de ser manipulat. En cas contrari, qualsevol persona podria agafar el certificat d'una altra persona i posar la seva clau, amb l'objectiu de suplantar-la.

La firma continguda dins d'un certificat digital es genera usant la clau privada de l'emissor.

1.3.4 Emissió de certificats

A l'hora de distribuir la nostra clau pública usant certificats digitals, una pregunta molt important és: qui me'l genera? O sigui, qui serà l'emissor? En aquest sentit, normalment, existeixen dues vies.

Una opció és que el mateix propietari de la clau sigui l'encarregat de generar-se ell mateix un certificat. Al contrari que un carnet físic, que pot tenir sistemes de protecció basades en el seu mitjà físic, de manera que sigui molt difícil l'autoedició (com passa amb un DNI), un document digital pot ser generat per qualsevol persona que sàpiga el seu format. Només cal disposar del programa adient. De fet, existeixen diversos programes, tant de lliure distribució com de pagament, que permeten generar certificats sense problemes.

Els certificats en els quals la identitat del propietari i l'emissor és la mateixa s'anomenen **autosignats**.

En aquest tipus de certificats, el nom distingit del propietari (*Subject*) i de l'emissor (*Issuer*) és exactament el mateix. Així mateix, el certificat està signat amb la clau privada de la pròpia clau pública continguda. Si bé aquest sistema és el més senzill, hi ha un problema força important. Si qualsevol persona pot generar-se un certificat autosignat amb el programa adient, això vol dir que també pot triar lliurement qualsevol de les dades que contingui. Aquestes poden ser reals o inventades. Res impedeix crear un certificat autosignat amb *Batman* al nom del propietari i *Ajuntament de Gotham City* al de l'emissor.

Els certificats autosignats resulten molt útils com certificats de prova per aplicacions en desenvolupament, ja que permeten la màxima flexibilitat en les dades que contenen.

Com podeu imaginar, aquest fet limita molt el seu ús. A nivell d'ús personal, només té sentit usar-los si, al fer l'intercanvi de certificats, les parts implicades poden comprovar, a allà i en aquell moment, que les dades que conté són certes. Per exemple, si es fa l'intercanvi en persona, o ja es coneix al propietari i el té penjat a una pàgina web personal que ja se sap a priori que és autèntica. En aquest sentit, ha d'existir un grau de confiança prèvia entre totes les parts implicades.

En la majoria de casos, un certificat no serà autosignat, ja que precisament a Internet sovint les parts implicades no es coneixen i sempre hi ha el perill que un atacant usi sistemes ben sofisticats per intentar fer-se passar per algú altre de manera expressament malintencionada.

La majoria de certificats digitals usats avui en dia en entorns reals han estat signats per una tercera entitat, una **autoritat de certificació**.

Una autoritat de certificació (o CA, *Certification Authority*) és una entitat que actua com a notari digital, de manera que tothom implicat en un intercanvi de claus sota el seu paraigües confia en què mai emetrà un certificat amb dades incorrectes. Qualsevol pot actuar com una CA, però és clar, que algú realment l'accepti com a tal ja dependrà del grau de confiança que inspiri. A petita escala, com una empresa, l'administrador de sistemes o el departament de recursos humans pot actuar com a CA de tots els treballadors. A gran escala, a nivell d'un país sencer, normalment es tracta de governs o organitzacions.

Quan algú vol generar un certificat, envia una petició amb la seva clau pública i la seva identitat a l'autoritat. Un cop la rep, l'autoritat comprova d'alguna manera que les dades del sol·licitant siguin reals. Per exemple, obligant-lo a presentar-se en persona a una de les seves oficines, amb algun document que acrediti la seva identitat (com un DNI). Només llavors genera el certificat, signant-lo amb la seva clau privada. Un usuari pot tenir diversos certificats, cadascun emès per una CA dins d'un àmbit diferent, igual que es pot tenir un DNI, un carnet de bibiloteca i un carnet d'estudiant, alhora.

La clau pública d'una CA es distribueix en un certificat digital autosignat.

Una de les responsabilitats d'una CA és disposar de tota una infraestructura per poder dur a terme aquest servei, així com posar a disposició del públic la seva clau pública de manera segura i poder donar de baixa certificats emesos.

S'anomena una **infraestructura de clau pública** (o PKI, *Public Key Infrastructure*) al marc per desplegar un mecanisme segur d'intercanvi de claus públiques.

Un cop es disposa d'un certificat generat per una CA, tothom que confiï en la veracitat de l'autoritat podrà confiar en què les dades que conté el certificat són correctes. Per les propietats de la firma digital, ningú pot haver-lo falsificat.

Per estar segur de la validesa d'un certificat, i veure si realment ha estat emès per l'autoritat que posa al seu camp de l'emissor, caldrà fer un seguit de comprovacions. El més important és veure si la firma digital del certificat és correcta, usant la clau pública de la CA. D'altra banda, també caldrà comprovar si el certificat és vigent en la data actual, tant per la seva data d'emissió i d'expiració, com mirant si està o no a la llista de certificats revocats, donats de baixa prematurament, que proporciona la mateixa CA.

Un parell d'exemples de CA propers a vosaltres els trobareu en l'Agència Catalana de Certificació (CATCert) o la Fábrica Nacional de Moneda i Timbre (FNMT), que generen certificats vàlids per fer la declaració de la renda per Internet, entre d'altres coses.

1.4 Gestió de claus

En un sistema amb N usuaris, cada usuari ha de gestionar N - 1 claus simètriques. En total hi ha en circulació (N *(N - 1))/2 claus simètriques.

Un dels principals problemes que cal resoldre per desplegar correctament una aplicació que usa sistemes criptogràfics és la gestió de les claus. Això sovint dóna més feina i és més susceptible a atacs que la implementació del sistema criptogràfic. Per una banda, hi ha la distribució de les claus de manera segura. D'altra banda, tampoc no s'ha d'oblidar el seu emmagatzematge un cop s'han rebut correctament. En tots dos casos és imprescindible que només les persones autoritzades puguin accedir a claus secretes (simètriques o privades), o si més no, garantir que fer-ho no sigui trivial. Cosa que no suceeix si simplement es desen els bytes de la clau a un fitxer o s'apunten a un paper. També cal veure com facilitar aquestes tasques quan una persona no només ha de treballar amb una clau, sinó amb moltes.

Per intentar pal·liar aquestes problemàtiques, les biblioteques criptogràfiques solen oferir la possibilitat de treballar amb magatzems de claus. Java no n'és una excepció.

Un **magatzem de claus** (o *keystore*, en anglès) és un repositori que permet emmagatzemar diverses claus criptogràfiques en un únic mitjà i que garanteix que el contigut es troba protegit, per exemple, mitjançant una contrasenya, en cas de pèrdua o robatori.

Un exemple de magatzem de claus associat a maquinari és el DNI electrònic.

Existeixen diferents tipus i formats de magatzems de claus. Els més habituals solen ser fitxers, però també hi ha magatzems associats a maquinari, com poden ser les targes intel·ligents (*smartcards*) o alguns llapis de memòria USB. El JCE treballa per defecte amb dos formats molt concrets, basats en fitxers: els anomenats JKS (*Java Keystore*) i els JCEKS (*JCE Keystore*). Ambdós són propietaris i exclusius de Java. Per tant, qualsevol clau emmagatzemada dins d'aquests tipus de magatzem només pot ser tractada des d'un programa fet amb Java.

Un magatzem de claus és com un clauer digital.

Un magatzem de claus Java pot desar una quantitat arbitrària d'informació criptogràfica en un únic fitxer, usualment amb extensió JKS. Aquest s'organitza internament en diferents entrades (*entries*, en anglès). Cada entrada s'identifica amb un **àlies**, una cadena de text triada pel propietari del magatzem, que la identifica de manera unívoca. Sempre que s'accedeix a la informació, cal dir quin àlies es vol accedir, comptant que mai n'hi poden haver de repetits.

Per garantir que la informació està protegida, un magaztem Java la pot xifrar amb contrasenya, si així es desitja. Les contasenyes es poden assignar a dos nivells. Per una banda, el magatzem es pot protegir a nivell general, de manera que simplement per accedir-hi i intentar explorar quin és el seu contingut faci falta una contrasenya. D'altra banda, cada entrada individual del magatzem es pot protegir amb una contrasenya diferent. Per tant, en el cas més restrictiu, un magaztem que contingui N entrades pot arribar a estar protegir amb 1 + N contrasenyes: una general al magatzem i una per cada entrada.

1.4.1 L'eina Keytool

Si bé el JCE ofereix la possibilitat de generar claus o de gestionar magatzems mitjançant codi Java, el cas més habitual és generar-les per línia de comandes, de manera que ja estiguin automàticament desades a un magatzem segur, llestes per ser usades als vostres programes.

Keytool és un petit executable que es proporciona amb totes les versions del Kit de Desenvolupament de Java (JDK), juntament amb el compilador i l'intèrpret. Permet gestionar magatzems de claus JKS i JCEKS desde línia de comandes.

El manual de referència de la Keytool el podeu trobar a http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html.

Aquesta eina té moltes opcions, però enumerem només les més típiques a la taula 1.2. Cal anar amb una mica de cura al triar les opcions i paràmetres amb les quals executar Keytool, ja que algunes d'elles tenen valors per defecte en cas que no s'especifiquin. El cas de les opcions "-storetype" i "-keystore" és especialment important.

Taula 1.2. Opcions típiques de la Keytool

Opció	Tipus de paràmetre	Valor per defecte
-storetype	Tipus de magatzem (JKS, JCEKS)	JKS
-keystore	Nom del fitxer a tractar	fitxer ".keystore" a la carpeta $HOME de l'usuari
-alias	Nom de l'alies de l'entrada a tractar	myalias
-keyalg	Algorisme criptogràfic	DES per clau simètrica, DSA per asimètrica
-keysize	Mida de la clau	Varia segons l'algorisme. Per RSA és 2.048
-storepass	Contrasenya del magatzem	(Si no es posa, preguntarà pel teclat)
-list	Mostra el contingut del magatzem	N/A
-v	Mode "verbose". Aporta informació extra	N/A

En qualsevol cas, podeu consultar-les totes fent *keytool -help*.

1.4.2 Accés a un magatzem a través de codi Java

Finalment, tot i que l'eina Keytool ofereix un mecanisme senzill per generar tota mena de claus i certificats directament a dins de magatzems criptogràfics, una tasca inevitable serà accedir-hi des del codi de les vostres aplicacions per poder executar algorismes criptogràfics.

La gestió de magatzem de claus mitjançant les biblioteques de JCE es porta a terme a través de la classe `java.security.KeyStore`. El procés per poder interactuar amb un magatzem de claus mitjançant una instància d'aquesta classe sempre és el mateix:

1. S'obté una instància del tipus de magatzem amb què es vol treballar usant el mètode estàtic `getInstance` que té com a únic paràmetre el tipus de magatzem.
2. S'inicialitza la instància mitjançant el mètode `load`, que té dos paràmetres: la ruta al fitxer i la contrasenya amb la qual es va protegir al ser creat.
3. Un cop inicialitzat, ja es poden cridar lliurement els seus mètodes per consultar-ne el contingut.

El següent fragment de codi us mostra com dur a terme aquesta tasca:

```
1 public KeyStore loadKeyStore(String ksFile, String ksPwd) throws Exception {
2     KeyStore ks = KeyStore.getInstance("JCEKS");
```

```
    File f = new File (ksFile);
    if (f.isFile()) {
        FileInputStream in = new FileInputStream (f);
        ks.load(in, ksPwd.toCharArray());
    }
    return ks;
  }
```

Un cop s'ha carregat el magatzem, ja és possible interactuar amb ell per poder accedir a les seves entrades, indexades amb un àlies. Això es fa cridant els mètodes que correspongui d'entre els definits a la classe `KeyStore`. Entre els més significatius, podeu trobar:

- `int size()`: consulta el nombre d'entrades al magatzem.
- `Key getKey(String alias, char[] password)`: obté la clau (simètrica o privada, el que correspongui per l'entrada) associada a un àlies, si existeix. Si la seva entrada està protegida amb contrasenya, cal proporcionar-la al segon paràmetre. Una clau simètrica és de tipus `SecretKey`, mentre que una de privada és de tipus `PrivateKey`.
- `Certificate getCertificate(String alias)`: permet extreure un certificat desat a un àlies concret.

De totes formes, el més recomanable és consultar directament la documentació de l'API de Java per aquesta classe per veure tots els mètodes disponibles per accedir a les dades contingudes.

2. Aplicacions Java segures a Internet

El rol d'Internet és increïblement important en el desenvolupament d'aplicacions en Java. Aquest fet es fa molt evident només observant el *boom* de les aplicacions mòbils, si bé tampoc cal oblidar-se de les aplicacions d'escriptori.

Per una banda, la xarxa de xarxes s'ha convertit en un canal de distribució ideal per aplicacions i biblioteques de classes. Aquest escenari s'aplica tant si sou algú que simplement vol baixar-se un joc o un desenvolupador que no vol haver de reinventar la roda i simplement desitja aprofitar un conjunt de classes que algú altre ha fet i ha publicat a Internet, llestes per ser descarregades. Ara bé, és segur executar en el vostre ordinador, sense la més mínima precaució, un codi que no heu fet vosaltres? Potser estaria bé que existís algun mecanisme per executar codi Java aliè en un entorn controlat, monitoritzant els accessos als recursos.

D'altra banda, i aquest és el cas segurament més evident, moltes aplicacions es basen actualment en les comunicacions en xarxa a través d'Internet per dur a terme les seves tasques. Aquestes aplicacions també han de garantir la seguretat de les dades intercanviades, però comptant que estan treballant en un entorn de sistemes heterogeni. Per tant, cal garantir la compatibilitat dels protocols emprats. Caldria una biblioteca que simplifiqui tot aquest procés, sense haver d'executar algorismes criptogràfics a baix nivell.

2.1 Seguretat a la plataforma Java

La màquina virtual es troba dins l'entorn d'execució de Java (o JRE, *Java Runtime Environment*)

Un dels trets més distintius de Java és el fet que es tracta d'un llenguatge interpretat. Tot i que el codi font dels vostres programes s'ha de compilar, aquest no es transforma en codi objecte directament interpretable per l'ordinador. En realitat, es tradueix a un llenguatge binari intermedi anomenat *bytecode*. Aquest necessita d'un intèrpret, que ha d'estar instal·lat a tots els equips on es vulgui executar el programa: la màquina virtual de Java (o JVM, *Java Virtual Machine*).

El benefici més evident d'usar un llenguatge interpretat és poder executar aplicacions en entorns heterogenis, de manera que les aplicacions siguin multi-plataforma. Mentre l'entorn d'execució estigui instal·lat a l'equip, no importa el seu sistema operatiu o model de processador. Les contrapartides són que cal una configuració prèvia del sistema abans no es pot executar cap programa i que les aplicacions són menys eficients en l'ús de recursos i velocitat d'execució, ja que el codi no s'executa directament sobre el maquinari.

Des del punt de vista exclusivament de seguretat, l'ús d'un intèrpret també aporta un seguit d'avantatges addicionals. Ja que tot programa en Java ha de ser processat per aquest intèrpret abans de que es doni cap ordre directa al maquinari, això

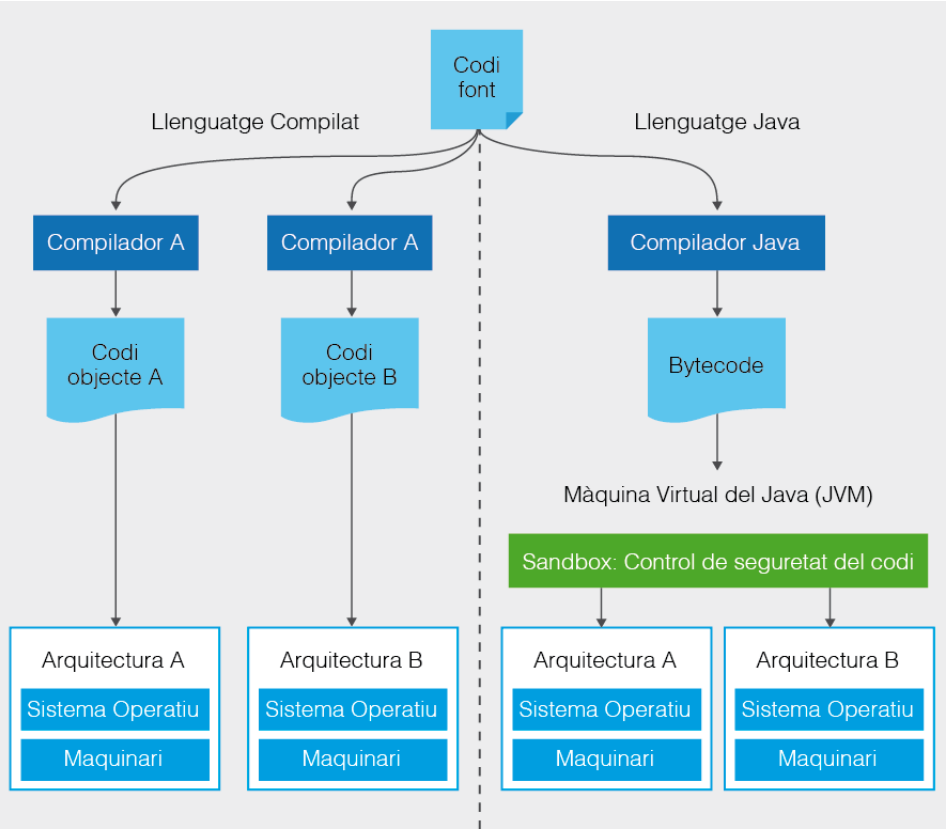
permet establir certs controls sobre els programes. Per una banda, pot arbitrar les accions de les aplicacions i el tipus de classes usades, vetant-ne l'accés a cert tipus, si escau. D'altra banda, també permet dur a terme comprovacions sobre el propi *bytecode* abans de ni tan sols carregar-lo i executar-lo, per evitar certs tipus de paranys.

Una capsa de sorra, un entorn controlat on poder jugar.

La **capsa de sorra del Java** (o *sandbox*) és com s'anomena popularment l'entorn controlat en el qual es s'executen les aplicacions fetes en aquest llenguatge, de manera que sigui possible restringir certes accions, per garantir-ne la seguretat.

La diferència entre el procés de generació d'una aplicació i la seva execució entre Java i els llenguatges compilats tradicionals, amb la participació de la capsa de sorra del Java, es mostra a la figura 2.1.

Figura 2.1. Generació d'executables i control del codi en temps d'execució



Si es treu profit d'aquesta funcionalitat de Java, ens trobem davant d'un gran avantatge, ja que permet executar les aplicacions dins un entorn inicialment controlat. Un cop tinguem clar que no són malicioses, llavors podem obviar aquests controls o fer-los més laxes.

2.1.1 El gestor de seguretat

Un tallafocs (o *firewall*) és un programa que permet controlar el trànsit de xarxa que surt o entra d'un ordinador.

Si bé la màquina virtual de Java té la capacitat de establir controls sobre el codi que s'hi vol executar, per defecte, no en força cap. Per tant, d'entrada, totes les aplicacions tenen accés lliure a qualsevol recurs de l'ordinador en el qual s'executen sense cap altra restricció que les imposades pel propi sistema operatiu. Per exemple, les proteccions al sistema de fitxer o els tipus de connexions permeses per un tallafocs.

Sempre que es vulgui establir algun tipus de control, caldrà indicar-ho mitjançant l'assignació d'un gestor de seguretat al programa que es vol executar.

> Un **gestor de seguretat**, o *security manager* en anglès, és una classe mitjançant la qual es pot establir una política de seguretat sobre una aplicació quan s'executa. O sigui, indicar quin és el conjunt d'accions que té permès, o no, dur a terme.

En el cas de Java, per indicar a una aplicació que ha d'usar un gestor de seguretat concret, cal fer-ho usant l'opció d'execució `-D` a la seva línia de comandes, que serveix per assignar valors a les propietats del sistema.

```
1 java -Djava.security.manager=NomGestor [altres opcions] nomAplicació
```

Opcions d'execució i Netbeans

Les opcions d'execució de l'intèrpret de Java s'indiquen afegint-les a la pròpia comanda quan s'executa. Ara bé, en el cas d'usar eines en les quals aquest procés queda ocult, com pot ser l'edició mitjançant un IDE, caldrà cercar a les seves opcions com afegir-les. El procés pot variar per cada IDE.

En el cas de les versions 7.x del Netbeans, cal obrir les propietats del projecte, dins del menú *Fitxer* (o *File*), anar a l'opció d'*Execució* (o *Run*) i afegir l'opció dins de la capsa de text anomenada *Opcions* (*VM Options*).

Tot i que Java permet implementar els vostres propis gestors de seguretat, de manera que s'adaptin a les vostres necessitats, ens centrem ara en el gestor que es proporciona per defecte, la classe `java.security.manager.SecurityManager`. Per usar aquest gestor per defecte, n'hi ha prou amb usar l'opció esmentada, però sense indicar cap nom.

```
1 java -Djava.security.manager [altres opcions] nomAplicació
```

Un cop assignat el gestor de seguretat, l'aplicació veurà limitats els recursos als quals pot accedir durant la seva execució. En cas d'intentar accedir-hi igualment, succeirà un error d'execució (`AccessControlException`).

El gestor de seguretat controla les accions dels programes.

Pel cas del gestor proporcionat per defecte, les limitacions són les següents, a grans trets:

- No es permet la lectura del sistema de fitxers fora de la carpeta de l'aplicació.

- No es permet en cap cas l'escriptura al sistema de fitxers.
- No es poden consultar propietats del sistema vinculades a l'usuari o al sistema de fitxers. Per exemple, quina és la seva carpeta personal (carpeta *HOME*).
- No es poden dur a terme certes connexions de xarxa.
- No es poden llençar comandes locals mitjançat la crida `Runtime.exec`.
- No és possible llençar codi en biblioteques natives, no fetes amb Java.

Una prova molt senzilla per veure això és la següent. Genereu un fitxer de text anomenat Test.txt, amb qualsevol contingut, a l'arrel del vostre sistema de fitxers. Per exemple, en un sistema basat en Unix, a la carpeta "/" i en un basat en Windows, a la "C:\". Executeu el codi que es mostra a continuació sense assignar un gestor de seguretat i, tot seguit, torneu-ho a fer assignant el gestor per defecte.

```
public class TestFileAccess {

  public static void main(String[] args) throws Exception {
    File f = new File ("/Test.txt");
    try (FileInputStream in = new FileInputStream(f)) {
      in.read();
    }
  }
}
```

Si no hi ha cap gestor assignat, el programa actuarà tal com s'espera. Però si està activat el gestor, donarà un error del següent estil:

```
Exception in thread "main" java.security.AccessControlException: access denied
    (
"java.io.FilePermission" "\Test.txt" "read")
        ...
```

En canvi, si copieu el fitxer a la carpeta del vostre programa i modifiqueu el codi font de manera que s'accedeixi a Test.txt enlloc de /Test.txt, el programa funcionarà de nou sense problemes. Per tant, aquí podeu veure que el gestor de seguretat, per defecte, controla l'accés als recursos de manera que un programa no pot accedir lliurement a qualsevol part del sistema en el qual s'executa.

2.1.2 Assignació de permisos

El conjunt de restriccions a les quals es veu sotmesa una aplicació Java quan és supervisada pel gestor de seguretat per defecte és bastant gran. Normalment, però, voldreu establir un seguit d'excepcions a la llista de controls, de manera que l'aplicació sí que tingui accés a alguns recursos, o pugui dur a terme certes accions concretes.

Per poder establir quines accions pot dur a terme una aplicació controlada pel gestor de seguretat, cal assignar-li també un **fitxer de polítiques** (o *policy file*, en anglès). Aquest conté una llista de les accions que sí que pot dur a terme sense problemes.

Per defecte, Java cerca el fitxer anomenat .java.policy, a la carpeta principal (*HOME*) de l'usuari que executa l'aplicació. Noteu que, en sistemes Unix, és un fitxer ocult. Normalment, aquest fitxer no existeix d'entrada, un cop instal·lat el Java. L'ha de generar i editar cada usuari, indicant, a títol individual, quina mena de controls vol dur a terme sobre les aplicacions que executa.

De totes formes, també és possible indicar un altre fitxer diferent a aquest amb l'opció "-D". Alerta amb el doble igual (==) a la comanda.

```
java –Djava.security.manager –Djava.security.policy==rutaFitxer nomAplicació
```

Un fitxer de polítiques permet incloure opcions de diversa índole i complexitat. Aquest es composa d'una llista de blocs `grant { ... }`, dins dels quals es llisten, mitjançant la comanda `permission`, els diferents permisos garantits. La comanda `permission` requereix tres paràmetres:

- El **tipus de permís**. Java organitza els diferents tipus de permisos mitjançant la definició de classes, que representen una acció possible que cal supervisar, a nivell genèric.

- El **nom del seu objectiu** (o *target name* en anglès), que és un identificador del recurs concret a controlar.

- El conjunt d'**accions** que es permet dur a terme sobre aquest objectiu. Els valors possibles per aquests dos casos solen dependre del tipus de permís, ja que no sempre tenen sentit per qualsevol tipus.

A títol d'exemple, l'estructura d'un fitxer de polítiques sol ser la següent:

```
//Un bloc de permisos
grant {
  permission NomTipusPermís1 "NomObjectiu1", "Acció1.1, Acció1.2,...";
  ...
  permission NomTipusPermísN "NomObjectiuN", "AcciónN.1, AcciónN.2,...";
};

//Un altre bloc de permisos
grant {
  permission NomTipusPermís1 "NomObjectiu1", "Acció1.1, Acció1.2,...";
  ...
  permission NomTipusPermísN "NomObjectiuN", "AcciónN.1, AcciónN.2,...";
};

...
```

Policytool

Per tal de no haver de barallar-se amb la sintaxi específica d'un fitxer de permisos, l'entorn de desenvolupament de Java ofereix una eina molt útil: un editor de polítiques.

L'eina **Policytool** permet generar conjunts de permisos dins de fitxers de polítiques sense haver de conèixer la seva sintaxi. Tot es fa mitjançant menús desplegables en una interfície gràfica.

El gran avantatge d'aquesta eina és que, en molts casos, ja enumera les opcions disponibles i no cal cercar la seva llista a la documentació. Per exemple, mitjançant menús desplegables, ja mostra tots els tipus de permisos, o el conjunt d'accions disponibles.

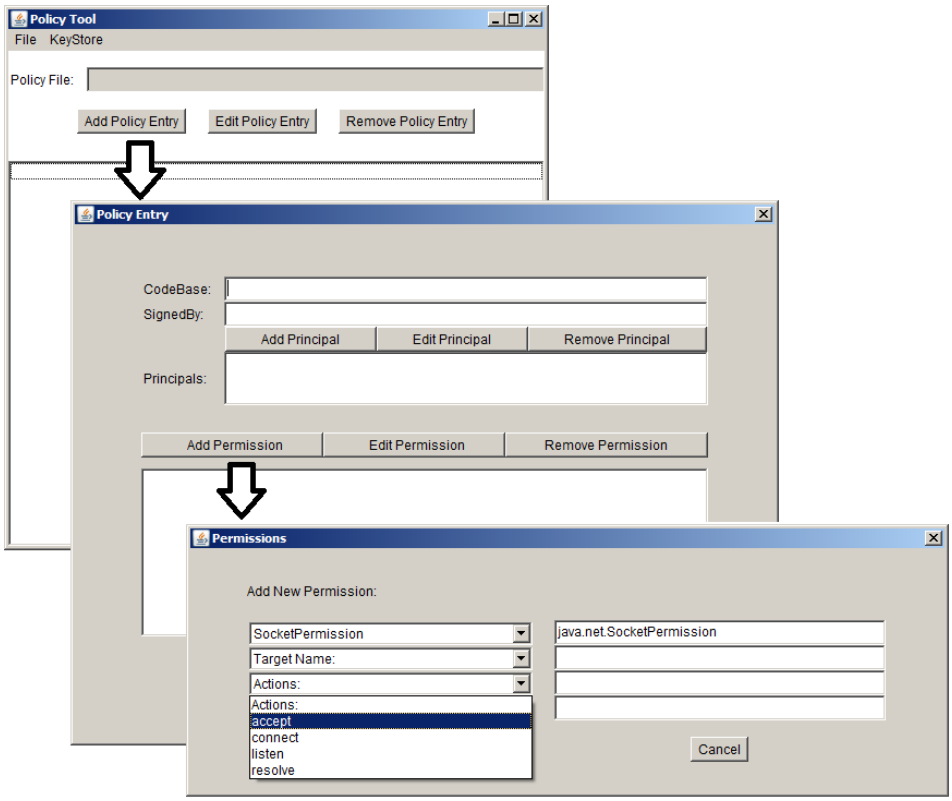
A grans trets, el seu funcionament és el següent, esquematitzat també a la figura 2.2.

Quan s'obre l'eina, es parteix d'un document buit. El menú *File* s'usa per desar el fitxer de polítiques un cop finalitzeu, generar-ne de nous o obrir d'altres que es vulguin editar.

A nivell d'opcions, les més destacades són:

- Allò que l'eina anomena *Entry* és un bloc `grant`. Se'n poden afegir de nous amb el botó *Add Policy Entry*, editar els existens amb *Edit* i esborrar-ne amb *Delete*.
- Dins de l'editor d'*Entries*, s'afegeixen els permisos. Novament, es poden afegir (*Add*), editar (*Edit*) o eliminar (*Delete*).
- El diàleg d'edició d'un permís es basa en tres menús desplegables. Un enumera tots els permisos possibles (*Permissions*), de manera que només cal triar l'adient. L'altra llista totes les accions (*Actions*) pel tipus de permís seleccionat. Finalment, el nom de l'objectiu s'ha d'escriure en un quadre de text (*Target Name*).

Figura 2.2. Ús de l'eina Policytool per afegir un nou permís



Tipus de permisos

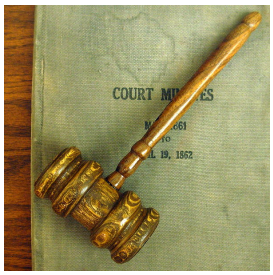
Les classes associades als permisos solen estar definides al *package* relatiu al tipus d'acció a controlar (fitxers: `java.io`, xarxa: `java.net`, etc.)

Un cop sabeu com editar fitxers de polítiques, anem a veure un resum de quin és el conjunt de classes que defineixen tipus de permisos dins les biblioteques de Java. La llista es pot trobar a la taula 2.1.

Taula 2.1. Classses associades a cada tipus de permís

Tipus de permís	Recurs controlat
`java.io.FilePermission`	Sistema de fitxers
`java.net.SocketPermission`	Connexions de xarxa a través de sòcols
`java.util.PropertyPermission`	Propietats del sistema
`java.lang.RuntimePermission`	Màquina virtual de Java, classes i fils d'execució
`java.awt.AWTPermission`	Interfície gràfica amb l'usuari
`java.net.NetPermission`	Connexions de xarxa a través d'HTTP
`java.lang.reflect.ReflectPermission`	Estructura interna de les classes
`java.io.SerializablePermission`	Seriació d'objectes
`java.security.SecurityPermission`	Mecanismes criptogràfics i polítiques de seguretat

Els valors possibles pels paràmetres vinculats als noms d'objectius i acció concreta permesa varien segons el tipus de permís, ja que per cada cas només tenen sentit un seguit de valors o uns altres. Per poder disposar d'una llista exhaustiva, caldrà mirar la documentació de cada classe.



Els permisos dins d'un fitxer de polítiques estableixen què pot fer una aplicació Java.

Tot seguit es mostra amb més detall tres d'aquests tipus de permisos, els més habituals en una aplicació.

java.io.FilePermission

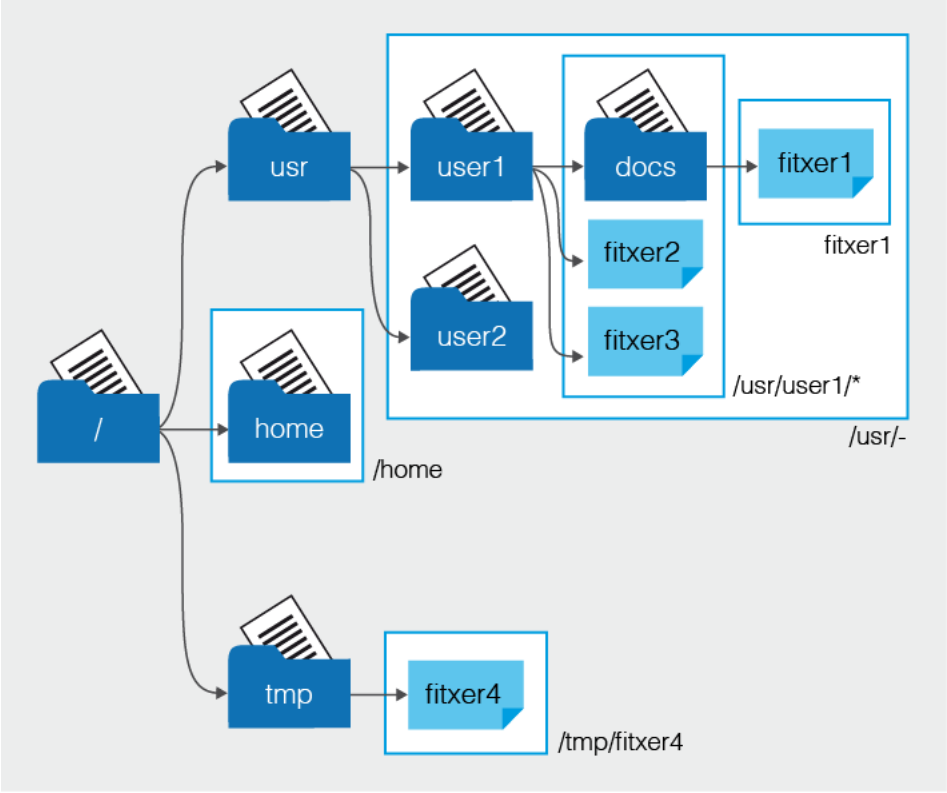
Aquest tipus de permís és un dels més utilitzats, ja que és el que permet establir excepcions en el control d'accés al sistema de fitxers.

El nom de l'objectiu que accepta aquest tipus de permís és el nom de la ruta, absoluta o relativa a l'aplicació, a la qual es vol permetre accedir. La sintaxi també accepta alguns caràcters especials per referir-se de manera senzilla a certs casos comuns. Posar al final de la ruta un asterisc (*) indica tot el contingut dins una carpeta concreta. Posar un guió (-) indica tots els fitxers i subcarpetes successives sota la ruta indicada. A part, existeix la cadena espacial `<<ALL FILES>>` que serveix per dir que s'aixequen les restriccions sobre tots els fitxers existents al sistema, sigui quina sigui la seva ruta.

Els noms d'objectiu d'una `FilePermission` també accepten, si es vol, un caràcter especial `${/}`. Aquest es tradueix al caràcter de separació dels elements d'una ruta que correspongui, segons el sistema operatiu de la màquina. A un sistema basat en Unix, es traduirà a "/" mentre que a un sistema Windows serà a "\".

La figura 2.3 mostra alguns exemples de rutes seguint la sintaxi esperada. En aquesta figura, es parteix de la premissa que el programa en Java s'executa a la carpeta `/usr/usr1/docs`, de cara a resoldre rutes relatives.

Figura 2.3. Especificació de rutes per seguint la sintaxi de l'objectiu d'un FilePermission



«ALL FILES»

El paràmetre que indica el valor d'aquest tipus de permís pot controlar quatre operacions possibles:

- `read`: permís de lectura.
- `write`: permís de lectura/escriptura. Es poden afegir o esborrar dades.
- `execute`: permís d'execució. És possible invocar un programa executable des del codi Java.
- `delete`: permís d'esborrament. El fitxer es pot eliminar des del codi Java.

java.net.SocketPermission

Aquest tipus de permís controla les connexions que fa una aplicació a Internet usant el mecanisme de sòcols. En aplicacions en xarxa és molt útil, ja que restringeix els equips als quals l'aplicació es pot connectar. Això pot evitar que un programa generi connexions a màquines no autoritzades.

Per referir-se a l'equip local on s'executa l'aplicació, es pot usar l'identificador *localhost*, o no posar-ne cap.

En aquest cas, el nom de l'objectiu que accepta aquest tipus de permís és el nom d'una connexió a Internet, o sigui, un parell de valors formats pel nom DNS de l'equip, o l'adreça IP directament, més el rang de ports del servei destí, separats per dos punts. Per exemple, si es vol donar permís perquè una aplicació es connecti al servidor web de l'IOC, usaria el nom d'objectiu "ioc.xtec.cat:80". També és possible usar el caràcter especial "*" per indicar qualsevol nom d'equip dins un domini.

En el cas del paràmetre que indica les accions que permet controlar aquest tipus de permís, hi ha les següents possibilitats. Aquestes es corresponen en major part a les diferents accions que es poden dur a terme amb les classes `java.net.Socket` i `java.net.ServerSocket`:

Un *bind* és el procés d'assignació d'un servei escoltant un port obert concret.

- `accept`: s'accepten connexions originades des de l'equip i port objectiu.
- `connect`: es permet establir connexions des de l'equip local a l'objectiu.
- `listen`: permet a l'objectiu posar en marxa un sòcol de servei (`java.net.ServerSocket`) per escoltar connexions entrants. O sigui, dur a terme el que s'anomena un *bind*.
- `resolve`: ens permet resoldre la traducció DNS del nom de màquina a una adreça IP.

Noteu que cap de les accions lligades al control mitjançant el gestor de seguretat tenen relació amb l'intercanvi de dades. Només es controla l'establiment de connexions.

java.util.PropertyPermission

Finalment, un altre tipus de permís que val la pena veure amb una mica de detall és la capacitat de llegir o alterar les propietats del sistema en el qual s'executa Java. Una propietat s'estructura com un parell: nom d'entrada = valor assignat. Per exemple : `java.vm.version = 23.6-b04`.

La llista de propietats és útil a la màquina virtual i a les aplicacions de cara a poder establir el sistema en el qual s'executen i saber com actuar en certes situacions. Per exemple, saber la versió de Java que s'està usant, la carpeta on està instal·lat o la carpeta personal de l'usuari que executa l'aplicació. Per un atacant poder obtenir, o fins i tot canviar, els valors d'aquestes propietats pot ser una bona oportunitat per dur a terme comportament maliciosos. Per exemple, si vol robar informació personal, saber la vostra carpeta personal és de molta utilitat per anar per feina i no haver de cercar per tot el disc.

Si voleu donar una ullada a les propietats del vostre sistema, es pot fer de manera senzilla amb el codi:

```
Properties props = System.getProperties();
props.list(System.out);
```

De cara a generar entrades al fitxer de polítiques d'acord amb la seva sintaxi, el nom de l'objectiu que cal posar és el del nom de la propietat. Les accions possibles són dues:

- `read`: permet consultar el valor d'una propietat. Això es pot fer amb el mètode `System.getProperty`.
- `write`: permet modificar en temps d'execució el valor d'una propietat. Això es pot fer amb el mètode `System.setProperty`.

Propietats del sistema al fitxer de polítiques

A diferents ordinadors, la ruta a la carpeta de l'usuari o el lloc on es troba instal·lat Java pot ser diferent. Això fa que un fitxer vàlid per un sistema concret no serveixi per un altre. Les propietats del sistema resulten molt útils per evitar això i crear fitxers de polítiques portables.

La sintaxi d'aquests fitxers també accepta incloure una propietat del sistema dins del paràmetre del nom d'objectiu, en el format `${nomPropietat}`, posant el nom de la propietat amb un signe de dòlar a l'inici i entre claus. Quan Java troba aquesta expressió, abans de prosseguir el tradueix al valor de la propietat corresponent.

Per exemple, si es vol donar permís per llegir els fitxers d'una carpeta anomenada *tmp* dins la carpeta de l'usuari que executa l'aplicació, es podria usar el nom d'objectiu `${user.home}${/}tmp`.

2.1.3 Codi signat

Les biblioteques de classes se solen empaquetar totes en un fitxer JAR.

Conèixer l'autor d'un codi Java pot ser un factor important de cara a decidir si el codi serà fiable i establir si cal executar-lo sota un gestor de seguretat i un seguit de polítiques. Aquest aspecte no es limita a saber qui el va generar originalment, sinó també poder estar segurs que ningú altre ha modificat la versió original, potser amb males intencions.

Una manera d'evitar aquesta mena de problemàtica seria disposar d'algun mecanisme que permeti controlar que una biblioteca, un cop generada, no ha estat alterada posteriorment. A la vegada, també caldria poder establir la identitat del seu creador, de manera que, abans d'instal·lar-la, puguem garantir que ha estat compilada per algú de confiança. I si fa coses insospitades, no pugui negar la seva autoria de manera que es dissuadeixi a atacants a generar codi maliciós, o com a mínim que sigui possible identificar-los fàcilment un cop hagin fet mal alguna vegada.

O sigui, es necessita garantir els serveis d'integritat, autenticació i no repudi als fitxers CLASS de les biblioteques Java. Per tant, la solució la trobem firmant digitalment el codi.

Jarsigner

L'entorn de treball de Java proporciona una eina per poder oferir unes mínimes garanties sobre l'origen del codi que s'executa.

L'eina **Jarsigner** permet signar digitalment i validar fitxers de biblioteques de classes, JAR. Aquesta es proporciona com un executable auxiliar, per línia de comandes, inclòs en totes les versions del *Kit de Desenvolupament de Java* (JDK).

La sintaxi d'ús del **Jarsigner** és molt senzilla i es resumeix de la següent forma.

Totes les opcions del Jarsigner queden llistades en pantalla si s'executa sense cap paràmetre.

```
jarsigner [opcions] nom_biblioteca alias_magatzem_claus
```

El següent exemple signa un fitxer JAR anomenat MyLib.jar mitjançant la clau privada emmagatzemada a l'àlies `dev1`, dins del fitxer dev1.jks (amb l'opció `-keystore`). El fitxer signat resultant queda desat amb el nom dev1-mylib.jar (amb l'opció `-signedjar`), de manera que no se sobreescriu l'original.

```
jarsigner -keystore dev1.jks -signedjar dev1-mylib.jar MyLib.jar dev1
```

Un cop un fitxer JAR està signat, la màquina virtual de Java en validarà la integritat mitjançant la firma abans de carregar-lo. Si la firma no és vàlida, prova que el fitxer ha estat modificat posteriorment de la generació de la firma, s'avorta l'execució del programa i es mostra un error del tipus `SecurityException`.

Validació mitjançant certificats de confiança

A part de veure si les dades que conté un JAR signat no s'han modificat posteriorment (integritat), també cal garantir la identitat de qui ha signat i la validesa del seu certificat. Ja que és possible generar certificats autosignats, aquest procés no és autocontingut. A partir només un certificat digital, i res més, no és possible saber si les dades que conté són reals o inventades. Cal veure qui l'ha emès i decidir si es considera algú fiable.

D'entrada, el procés de visualitzar la identitat continguda dins del certificat del signant d'un fitxer JAR es pot dur a terme manualment, abans d'executar el codi, mitjançant la mateixa eina Jarsigner. Aquesta també permet validar fitxers JAR signats i inspeccionar els certificats usats per signar el codi. Per fer-ho, cal usar l'opció `-verify` i indicar el nom del fitxer signat, amb les opcions `-verbose` i `-certs`.

Un exemple de sortida d'un fitxer signat correctament podria ser el que es mostra tot seguit. En cas de que el JAR no sigui vàlid, es mostra un error del tipus `java.lang.SecurityException`, indicant quin és el fitxer compromès detectat.

```
jarsigner -verbose -certs -verify dev-mylib.jar

...

sm      1650 Mon Apr 29 09:02:30 CEST 2013 ShowProperties.class

      X.509, CN=Developer 1, OU=DAM, O=IOC, L=Barcelona, ST=Catalunya, C=ES
      [certificate will expire on 28/07/13 08:46]
      [CertPath not validated: Path does not chain with any of the trust
          anchors]

...

jar verified.

Warning:
This jar contains entries whose signer certificate will expire within six
    months.
This jar contains entries whose certificate chain is not validated.
```

L'eina Jarsigner mostra una gran quantitat d'informació sobre els fitxers dins el JAR: estructura de carpetes, dates, etc. Però entre totes aquestes dades, hi ha uns quants aspectes destacables que val la pena comentar amb més detall.

Primer de tot, el resultat de tota la comanda es pot resumir en la breu frase, gairebé al final, de `jar verified`. O sigui, tot està en ordre des del punt de vista de la integritat del fitxer. No ha estat modificat després de la generació de la firma. Tot seguit, també es pot veure com, per cada fitxer protegit, es llista el certificat digital del signant. Finalment, dins del resultat del procés de validació, al final de tot de la sortida, hi ha un parell d'avisos, o *warnings*.

L'avís realment important, és el segon. Aquest diu, moltíssima atenció, que el fitxer ha estat signat amb la clau associada a un seguit de certificats que no estan validats (*`whose certificate chain is not validated`*). El significat del missatge és que els certificats i les firmes són correctes des d'un punt de visa criptogràfic, però Java no és capaç de saber si es pot fiar que les dades que conté el certificat siguin verídiques, ja que no coneix al seu emissor.

Un cas com aquest no vol dir que el signant sigui un atacant o que el certificat té dades inventades. Potser és un certificat generat sota un estricte control de seguretat, o potser se l'ha fet algú a casa seva amb Keytool. Simplement, no se sap. Com que Java no té prou informació per prendre una decisió de manera automàtica, llavors us informa al respecte perquè preneu les mesures que creieu adients. La decisió final és vostra.

Per poder prendre una decisió d'aquest tipus ell mateix, Java necessita disposar d'una llista d'emissors de certificats que vosaltres considereu de confiança (*trusted*). La decisió de quines entitats emissores són o no de confiança, i quins criteris usar per fer-ho, ja és cosa vostra. Amb qualsevol certificat emès per algú dins aquesta llista, Java considerarà que les dades que conté són automàticament verídiques. Per tant, per poder validar totalment la identitat d'algú que signa codi, abans us cal obtenir de manera segura el certificat digital del seu emissor.

Alerta: si un atacant aconsegueix afegir un emissor de confiança a la vostra llista, us podrà passar identitats falses impunement!

La llista de certificats de confiança es desa dins d'un magatzem de claus, amb el seu àlies corresponent. Aquesta tasca es pot fer amb l'eina Keytool i l'opció `-import`.

El certificat de l'emissor d'un certificat autosignat és ell mateix.

Un cop teniu clar que `Developer 1` és algú de confiança, que fa codi no maliciós, obtindríeu el certificat del seu emissor i l'incloureu al vostre magatzem de claus de confiança. Aquest podria ser el d'una autoritat de certificació, o en el cas d'un certificat autosignat, el mateix certificat directament.

Si el vostre magatzem de claus es troba al fitxer user-cert.jks i el certificat de l'emissor el teniu al fitxer issuer.cer, el podeu importar dins el magatzem a l'àlies `dev1` amb la comanda:

```
keytool -import -alias dev1 -keystore user-certs.jks -file issuer.cer
```

Un cop es disposa d'un magatzem de claus amb una llista d'emissors de confiança, es pot passar al Jarsigner perquè el tingui en compte de cara a la validació amb l'opció `-keystore`.

```
jarsigner -keystore user-certs.jks -verbose -certs -verify dev-mylib.jar

...

smk     1650 Mon Apr 29 09:02:30 CEST 2013 ShowProperties.class

      X.509, CN=Developer 1, OU=DAM, O=IOC, L=Barcelona, ST=Catalunya, C=ES (
          dev1)
      [certificate will expire on 28/07/13 08:46]

...

jar verified.

Warning:
This jar contains entries whose signer certificate will expire within six
    months.
```

Si tot és correcte, com en aquest cas, la sortida ja no mostra els avisos relatius a la manca de confiança dels certificats.

Gestió de codi signat

Mitjançant la firma digital i la llista d'emissors de confiança, és possible establir, amb unes mínimes garanties, la identitat d'algú. Ara bé, sovint, en el nostre dia a dia, la identitat d'una persona en si no és una fita, sinó una eina per dur a terme alguna decisió.

Per exemple, suposeu que algú us demana permís per entrar al vostre portal. El seu nom en si, i ja està, no serveix de gran cosa. El que és important realment és que vosaltres relacionareu aquesta identitat amb algú que coneixeu o no, i decidireu si el voleu deixar passar. O potser fins i tot el tipus d'acció que li deixareu dur a terme serà diferent. Si és el carter, el rebeu a l'entrada i accepteu el paquet que us dóna. Si és un amic, el deixeu entrar a dins de casa. Si és algú de molta, molta confiança, potser ni tan sols us molestarà que obri la nevera i us agafi pel seu compte un refresc.

Aquest escenari també es pot dur a terme a nivell dels permisos que s'atorguen a una aplicació. Si s'executa codi signat, mitjançant un gestor de seguretat, és possible especificar diferents tipus de permisos per cada codi, depenent de la identitat del signant. Potser les aplicacions de `Developer 1` no volem que tinguin cap accés a la xarxa, mentre que les de `Developer 2` sí que volem que puguin resoldre noms, però limitar parcialment el seu accés al sistema de fitxers.

Per gestionar diferents permisos segons els signants d'un codi, Policytool permet configurar un magatzem de claus que serveix com a llista de certificats de confiança al menú principal de l'aplicació. Dins de l'apartat d'edició d'*Entries* es disposa del camp de text `SignedBy` en el qual cal posar el nom de l'àlies del certificat del signant a qui aplica aquest conjunt de permisos.

Per exemple, si es posa `dev1`, el grup de permisos només aplicarà al codi signat pel certificat desat al magatzem de claus sota l'àlies `dev1`, sigui quin sigui el nom de *Subject* del certificat.

Enlloc d'un únic àlies, també és possible afegir una llista d'àlies entre dobles cometes, separats per comes. En aquest cas, per garantir el permís cal que el codi estigui signat alhora pels certificats de tots els àlies. O sigui, si hi ha tres àlies, el codi ha d'estar signat pels tres. Molt de compte, ja que només que un dels àlies de la llista no hagi firmat el codi, es considerarà que no té permisos.

2.2 Connexions segures

L'estàndard del protocol SSL està definit al document RFC 6101: https://tools.ietf.org/html/rfc6101

Avui en dia moltes aplicacions fan ús de la xarxa per poder compartir informació. Per tant, és molt important garantir que aquests intercanvis es fan de manera segura. Això s'assoleix integrant dins les comunicacions en xarxa algun protocol que, fent ús de mecanismes criptogràfics, protegeixi les dades abans de ser enviades. Un dels protocols segurs més estesos actualment dins la indústria és **SSL** (*Secure Sockets Layer*, o capa de sòcols segurs, traduït al català). A manera il·lustrativa, aquest és el protocol que utilitzen tots els navegadors web per connectar-se a servidors segurs, com pot ser la banca o botigues en línia. El Java disposa de biblioteques que us permeten integrar-lo dins de les aplicacions que requereixen de comunicacions en xarxa.

Des del punt de vista d'un desenvolupador d'aplicacions, l'acceptació d'SSL garanteix un alt grau d'interoperabilitat entre les aplicacions, ja que permet comunicar aplicacions creades amb diferents llenguatges. A més a més, les biblioteques per defecte del Java ja disposen d'un conjunt de classes que permeten usar-lo sense gaires complicacions. Ambdós factors representen un gran avantatge respecte haver de dissenyar un protocol a mida per una aplicació concreta.

D'entrada, el protocol SSL garanteix sempre la privadesa i la integritat de les dades. Pel primer cas, això es fa mitjançant un sistema de clau simètrica, de manera que la seva eficiència sigui acceptable. La clau s'intercanvia de manera segura usant un algorisme d'intercanvi de claus. Pel que fa a la integritat, es fa ús d'un algorisme de *hash*.

> Un algorisme d'intercanvi de claus és un protocol criptogràfic mitjançant el qual és possible que dues entitats intercanviïn a distància una clau secreta de manera segura. Entre els més coneguts hi ha l'anomenat *Diffe-Hellman-Merkle*, que és el que usa SSL.

SSL també permet l'autenticació de les parts implicades mitjançant l'intercanvi de certificats digitals, en dos modes d'operació diferent. Per una banda, el més habitual és el mode d'**autenticació simple**, en el qual només s'identifica el servei. Tanmateix, també és possible fer que s'identifiquin tant el client com el servei, en mode d'**autenticació mútua**. Aquest cas és menys freqüent, ja que sovint en el

client s'usen sistemes, més senzills d'usar de cara als usuaris. Per exemple, l'ús de contrasenyes.

2.2.1 La negociació SSL

Quan s'inicia una connexió segura mitjançant SSL, les dues parts implicades (client i servidor) han de dur a terme prèviament un procés de negociació, o *handshake* en anglès, en el qual s'acorden els paràmetres i la informació criptogràfica que caldrà usar (bàsicament, les claus de xifrat o firma).

A la negociació SSL client i servidor es posen d'acord.

Tot seguit es descriuen les etapes d'aquest procés, indicant breument quines tasques es duen a terme en cada cas. El nom de cada etapa es precedeix pel nom de l'entitat qui l'executa (client o servei) i si és opcional o no.

1. **[CLIENT] Hello:** el client indica que vol iniciar una connexió segura SSL. Per fer-ho, indica fins i tot quina versió d'SSL, conjunt d'algorismes criptogràfics i mides de clau suporta.

2. **[SERVEI] Hello:** el servei accepta la petició i tria la darrera versió possible entre les suportades pel client de cara a usar el protocol segur. Llavors, escull l'algorisme i mida de clau més segur entre els disponibles. Un cop fetes aquestes tries, les comunica al client.

3. **[SERVEI, opcional] Enviament certificat:** aquesta etapa només es realitza si s'usa SSL per fer autenticació simple de servei, però cal dir que aquest és un cas molt habitual quan s'usa SSL. El servei envia el seu certificat digital al client, que conté la seva clau pública i la seva identitat.

4. **[SERVEI, opcional] Petició de certificat:** en cas d'usar SSL per fer autenticació mútua amb el client, el servei li demana el seu certificat digital.

5. **[SERVEI, opcional] Intercanvi de clau:** aquesta etapa només s'executa per alguns algorismes criptogràfics, que requereixen informació extra, a part d'una clau pública, per establir un canal de comunicacions segur. El servei envia al client aquesta informació addicional.

6. **[SERVEI] Hello finalitzat:** el servei passa el testimoni al client de cara a continuar amb el procés de negociació del protocol.

7. **[CLIENT, opcional] Enviament de certificat:** en cas que el servei demani el certificat del client a l'etapa 4, el client envia el seu certificat digital. Cal dir que aquest és un cas menys freqüent quan s'usa SSL.

8. **[CLIENT, opcional] Intercanvi de clau:** aquesta etapa és idèntica a l'etapa 5, però resolta per la part del client.

9. **[CLIENT, opcional] Validació certificat del client:** en el cas que es requereixi autenticació del client (veure etapes 4 i 7), s'envia al servei un missatge signat digitalment amb la clau privada del client. Això serveix

per demostrar que el client realment disposa de la clau privada associada al certificat que ha enviat, i no ha enviat el d'una altra entitat.

10. **[CLIENT] Canvi a mode xifrat:** el client avisa al servei que passa a mode xifrat de comunicacions. Fins ara, totes les comunicacions eren en clar.

11. **[CLIENT] Senyal de fi:** el client indica que la seva part de la negociació ha finalitzat i està llest per iniciar les comunicacions en mode segur. Aquesta etapa marca el final de la negociació SSL.

12. **[SERVEI] Canvi a mode xifrat:** el servei avisa que ell també passa a mode xifrat de comunicacions.

13. **[SERVEI] Senyal de fi:** el servei indica que, per la seva banda, també ha finalitzat la negociació. Aquesta etapa marca el final del *handshake* SSL.

14. **[CLIENT-SERVEI] Intercanvi de dades:** un cop finalitzada la negociació, el client i el servei porten a terme l'intercanvi de dades segur usant els paràmetres escollits a la negociació, fruit de les etapes anteriors.

15. **[CLIENT-SERVEI] Tancament de connexió:** un cop l'intercanvi de dades finalitza, es tanca la connexió a totes dues bandes.

Un cop la negociació finalitza i les dues parts ja han acordat tots els paràmetres, l'ús del protocol segur és transparent des del punt de vista de les aplicacions. Només cal gestionar l'enviament, recepció i processat de les dades intercanviades, sense haver-se de preocupar de cap aspecte criptogràfic. El protocol resol tots aquests detalls automàticament.

Tot i que normalment només s'usa quan s'inicia la connexió segura, la negociació SSL pot ser reiniciada en qualsevol moment dins d'un procés d'intercanvi de dades, de manera que es puguin canviar els paràmetres de manera dinàmica, si el client o el servei ho creuen adient.

2.2.2 Connexions SSL amb Java

Java disposa d'un seguit de biblioteques que permeten generar connexions sota SSL de manera relativament senzilla. Les classes implicades es troben a la biblioteca `javax.net.ssl`. D'entre aquestes, les més importants són `SSLServerSocket` i `SSLSocket`. Aquestes hereten de les classes que gestionen serveis no segurs dins la biblioteca `java.net`: `ServerSocket` i `Socket`, respectivament. En els dos casos, la primera serveix per escoltar connexions entrants a un servei, mentre que la segona serveix per gestionar l'intercanvi de dades, tant al servei com al client.

De fet, les diferències entre la versió de cada classe en format SSL o sense comunicacions segures només afecten al procés de negociació SSL i com configurar la validació dels elements criptogràfics intercanviats. Un cop la connexió s'ha

establert, la gestió de l'intercanvi de dades és exactament el mateix que si les dades s'enviessin sense seguretat.

Per veure més clarament com funciona l'ús d'SSL en una aplicació Java, el millor és veure un exemple d'aplicació client i servidor. Per començar, cal veure com generar una connexió amb autenticació simple, el més habitual. En aquest mecanisme, les dades estan xifrades i el client només accepta l'establiment de la connexió amb un servidor remot en el qual confiï. Si no és capaç d'establir aquesta confiança, interromp la connexió, ja que existeix el perill que algú l'estigui suplantant.

Després de veure això, cal afegir el codi per poder generar connexions autenticades mútuament.

Desplegament d'un servei amb autenticació simple

Per disposar d'un magatzem amb una clau privada i un certificat, es pot usar l'eina Keytool.

Abans de poder posar en marxa un servei fet en Java que accepti comunicacions mitjançant SSL, hi ha un procés previ de configuració de l'entorn. Aquest es deu al fet que tot servei que executi aquest protocol ha de comptar amb una clau privada i un certificat digital, que és el que usarà per dur a terme la negociació SSL amb qualsevol client que s'hi connecti. Aquests han d'estar desats a un magatzem de claus Java.

El magatzem ha de complir un seguit de condicions. Primer de tot, només ha de tenir una única entrada, amb un àlies qualsevol. Aquesta entrada conté el parell clau privada i certificat. Finalment, cal que la contrasenya del magatzem i la de l'entrada siguin exactament la mateixa. Si no es compleixen aquestes dues condicions, les classes que gestionen SSL a Java no sabran obtenir la informació que els cal per executar el protocol segur.

Un cop es disposa d'aquest magatzem, cal indicar al vostre programa la seva ruta dins el sistema de fitxers i la seva contrasenya. Això es fa mitjançant l'assignació dels valors adients a les propietats del sistema `javax.net.ssl.keyStore` i `javax.net.ssl.keyStorePassword`, respectivament.

Per assignar aquestes dues propietats del sistema, o bé amb el paràmetre `-D` de la línia de comandes a l'executar l'intèrpret de Java, o amb una crida al codi al mètode `System.setProperty`.

Fora d'aquest procés de configuració de les claus criptogràfiques del servei, la resta del procés és idèntic de com es programaria un servidor sense protocol segur, amb les classes `ServerSocket`, pel sòcol que accepta connexions, i `Socket`, pel sòcol de dades. Simplement són substituïdes per les classes `SSLServerSocket` i `SSLSocket`.

A nivell esquemàtic, les passes per configurar un servei amb connexió segura dins el codi Java són les següents:

1. Obtenir una fàbrica de sòcols, ja que aquests no es poden instanciar directament amb la sentència `new`.

2. Instanciar un `SSLServerSocket` usant el mètode `createServerSocket` sobre la fàbrica. En aquest pas s'indica el port del servei en el qual cal connectar-se.

3. Fer successives crides a `accept` per acceptar connexions entrants.

4. Servir al client a través del `SSLSocket` obtingut a l'acceptar una connexió entrant, al pas anterior.

Tot seguit es mostra un exemple de servidor mitjançant protocol segur SSL. Recordeu que per poder-lo provar correctament, cal generar un magatzem de claus. El més senzill és que useu un certificat autosignat generat automàticament amb l'eina Keytool.

```
keytool -genkey -alias "srvAlias" -keyalg RSA -keystore ServerKeyStore.jks -
    keysize 2048
```

El certificat del servidor web de l'IOC tindria com a *Common Name* el valor `ioc.xtec.cat`.

Al generar un certificat per un servei, normalment es recomana que el camp de *Common Name* del seu nom distingit sigui el nom DNS de la màquina en la qual s'executa. Si ho proveu tot en local, podeu posar `localhost`.

En aquest cas, per establir la informació relativa al magatzem de claus, s'ha triat l'ús de crides a `System.setProperty`. El magatzem està a la mateixa carpeta del projecte de l'aplicació i s'anomena ServerKeyStore.jks. La contrasenya del magatzem i de la seva entrada amb la clau del servidor és *serverks*.

```
public class SSLServer {

  public static void main(String[] argv) throws Exception {

    System.setProperty("javax.net.ssl.keyStore", "ServerKeyStore.jks");
    System.setProperty("javax.net.ssl.keyStorePassword", "serverks");

    try {
      SSLServerSocketFactory sslFactory = (SSLServerSocketFactory)
          SSLServerSocketFactory.getDefault();
      SSLServerSocket srvSocket = (SSLServerSocket) sslFactory.
          createServerSocket(4043);

      int numClient = 1;

      while (true) {
        SSLSocket cliSocket = (SSLSocket) srvSocket.accept();

        Scanner reader = new Scanner (cliSocket.getInputStream());

        String text = reader.nextLine();
        while (!text.equals("<<FI>>")) {
          System.out.println("[Client " +numClient + "] " + text);
          System.out.flush();
          text = reader.nextLine();
        }
        System.out.println("[Client " +numClient + "] Tancant connexió...");
        cliSocket.close();
        numClient++;
      }
    } catch (Exception ex) {
      System.out.println("Error en les comuncacions: " + ex);
    }
```

```
32    }
33 }
```

Concretament, aquest servei va atenent a clients un a un, de forma no concurrent. A cadascun li assigna un identificador successivament i mostra per pantalla les dades que rep d'ell. La cadena de text `<<FI>>` serveix per indicar que el client ha acabat i, per tant, pot tancar la connexió i servir al següent.

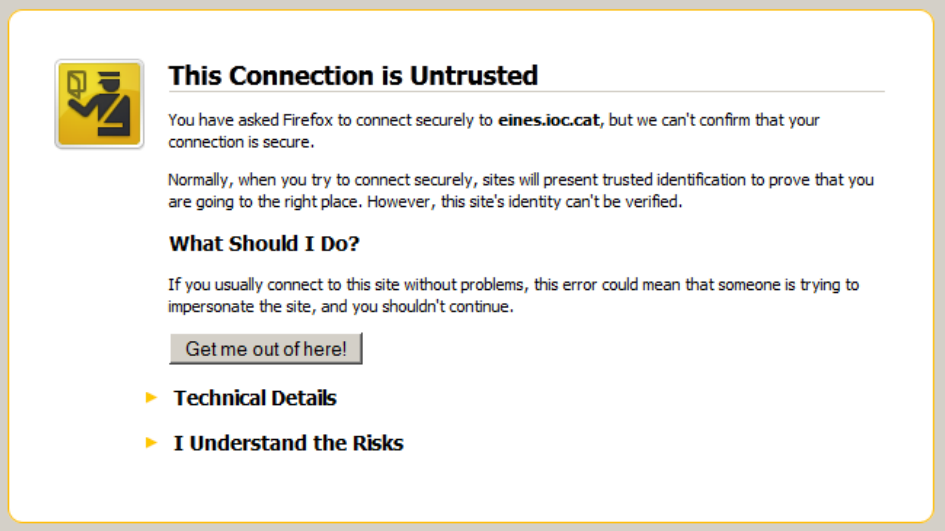
Desplegament d'un client amb autenticació simple

De la mateixa manera que passa amb un servidor, el client també necessita una configuració prèvia. En aquest cas, allò que cal especificar és la llista d'emissors de certificats digitals en els quals confia. D'aquesta manera, només s'acceptarà la creació d'una connexió cap a un servei que es consideri fiable. Si un servei s'identifica mitjançant SSL usant un certificat que el client no considera de confiança, la negociació fracassarà i no s'establirà cap connexió.

Els certificats d'emissors de confiança cal desar-los en un magatzem de claus Java especial, anomenat *truststore*, o magatzem de confiança. Aquest només conté entrades amb els certificats de les autoritats de certificació, o certificats autosignats, en els quals confiem. El client només es podrà connectar a serveis que hagin estat configurats amb un certificat emès per alguna de les entitats incloses al *truststore*.

Aquest comportament és comú a totes les aplicacions que usen SSL. Per exemple, la figura figura 2.4 mostra l'error que apareix al Navegador Mozilla Firefox (versió 22.0) quan es connecta un servidor web configurat amb un certificat que no es troba a la seva llista de confiança.

Figura 2.4. Error d'autenticació per la manca de confiança d'un servidor web en el navegador Mozilla Firefox 22.0.



Com passava amb el cas del servei, per indicar tota la informació vinculada a aquest magatzem de clau, cal usar propietats del sistema. La propietat `javax.net.ssl.trustStore` indica la ruta del fitxer, mentre que

`javax.net.ssl.trustStorePassword` indica la seva contrasenya. Novament, es pot optar per usar el flag `-D` per línia de comandes, o fer-ho a través de codi.

Fora de les particularitats lligades a la negociació SSL i les claus criptogràfiques, des de la perspectiva del client, l'establiment de la connexió també és gairebé idèntic a quan no s'usa protocol segur. Les passes serien les següents:

1. Obtenir una fàbrica de sòcols, ja que aquests no es poden instanciar directament amb la sentència `new`.
2. Instanciar un `SSLSocket` usant el mètode `createSocket` sobre la fàbrica. En aquest pas s'indica l'adreça i port del servei al qual cal connectar-se.
3. Iniciar la negociació SSL per establir la connexió segura.

Un cop fet, ja es pot iniciar l'intercanvi de dades.

El següent exemple mostra un client capaç de comunicar-se amb el servei anterior. Per fer-lo funcionar, abans cal configurar correctament el seu *truststore* Per això, haureu d'exportar el certificat autosignat des del magatzem de claus del servei i importar-lo al magatzem de confiança del client. Això es pot fer amb les comandes adients de Keytool.

Per exportar el certificat del magatzem del servidor, cal fer:

```
keytool -export -keystore ServerKeyStore.jks -alias "srvAlias" -file server.crt
```

A partir del fitxer resultant amb el certificat, per importar-lo es podria usar la comanda següent. Si el magatzem de claus no existeix, el crearà:

```
keytool -importcert -file server.crt -keystore ClientTrustStore.jks -alias "
    srvAlias"
```

Si voleu assegurar-vos que el contigut d'un magatzem de confiança és correcte, es pot usar la comanda:

```
keytool -list -v -keystore ClientTrustStore.jks
```

En el codi final, les propietats del sistema associades al *truststore* es configuren amb crides. El magatzem de confiança es considera que està al fitxer ClientTrustStore.jks, protegit amb la contrasenya `clientts`.

```
public class SSLClient {

  public static void main(String[] argv) throws Exception {

    System.setProperty("javax.net.ssl.trustStore", "ClientTrustStore.jks");
    System.setProperty("javax.net.ssl.trustStorePassword", "clientts");

    try {
      SSLSocketFactory sslFactory = (SSLSocketFactory) SSLSocketFactory.
          getDefault();
      SSLSocket cliSocket = (SSLSocket) sslFactory.createSocket("localhost",
          4043);

```

```
12        Scanner reader = new Scanner (System.in);
13        PrintStream writer = new PrintStream(cliSocket.getOutputStream());
14
15        System.out.println("Deixa una línia en blanc per acabar:");
16        String text = reader.nextLine();
17        while (!text.equals("")) {
18          writer.println(text);
19          writer.flush();
20          text = reader.nextLine();
21        }
22        writer.println("<<FI>>");
23        writer.flush();
24        cliSocket.close();
25      } catch (Exception ex) {
26        System.out.println("Error en les comuncacions: " + ex);
27      }
28    }
29  }
```

Aquest client simplement llegeix dades pel teclat i les envia al servidor en format text, tal com les ha llegides. Si en algun moment es pitja només la tecla de retorn, deixant la línia en blanc, aquest acaba. Abans, però, es duu a terme una desconnexió ordenada enviant el text `<<FI>>`, que serveix per avisar el servei que el client ha acabat.

Quan es fan proves amb clients i serveis amb SSL, sovint pot passar que hi hagi algun error en la negociació SSL i no es vegi a simple vista. Per casos com aquest, hi ha una propietat del sistema anomenada `javax.net.debug` que permet posar en marxa la depuració. El valor que cal assignar per fer-ho és:

```
1  System.setProperty("javax.net.debug", "SSL,handshake");
```

Desplegament d'autenticació mútua

En un intercanvi de missatges autenticat mútuament, tots els clients també s'han d'identificar davant del servidor mitjançant un certificat digital de confiança. En cas contrari, el servei rebutjarà l'intent de connexió, al no poder garantir la identitat del client. Aquest mecanisme és menys freqüent que l'autenticació simple ja que implica que tots els clients han de disposar d'una clau privada i un certificat digital per identificar-se. Tradicionalment, per identificar els clients se sol recórrer a eines com ara contrasenyes, que sovint es consideren més còmodes d'usar.

Des del punt de vista del codi Java, les rols de client i servidor s'inverteixen, i per tant l'autenticació mútua implica desplegar tant al client com al servei totes les dades que en l'autenticació simple només calia a un extrem:

- El client ha de disposar d'un magatzem criptogràfic amb una clau privada i un certificat digital.
- El servei ha de disposar d'un magatzem on desar els certificats de les entitats de certificació de confiança. Només acceptarà connexions de clients que proporcionin un certificat emès per alguna de les autoritats incloses en aquest magatzem.

En un cas com aquest, sovint se sol usar un únic fitxer que fa alhora de magatzem de claus privades i de certificats de confiança.

La manera com es desplega aquesta informació és igual que abans, mitjançant propietats del sistema. Per exemple, en el client ara caldria afegir les propietats que declaren quin és el magatzem de claus i la seva contrasenya, tal com mostra el següent fragment de codi.

```
public class SSLClient {

  public static void main(String[] argv) throws Exception {

    //Magatzem de confiança del client
    System.setProperty("javax.net.ssl.trustStore", "ClientTrustStore.jks");
    System.setProperty("javax.net.ssl.trustStorePassword", "clientts");
    //Clau del client
    System.setProperty("javax.net.ssl.keyStore", "ClientKeyStore.jks");
    System.setProperty("javax.net.ssl.keyStorePassword", "clientks");
    ...
```

Pel cas del servei, seria exactament igual, però amb els seus fitxers.

Un cop totes les parts implicades disposen de tota la informació criptogràfica, cal configurar el servei perquè requereix autenticació mútua, ja que per defecte actua amb autenticació simple. Això es fa invocant el mètode `setNeedClientAuth` sobre el seu sòcol de servei, l'objecte `SSLServerSocket`.

Si un client s'intenta connectar a un servei que requereix autenticació mútua, la negociació SSL fracassarà. En aquest cas, les dades que rebrà el client si estableix cap tipus d'intercanvi de dades amb el servei seran sempre invàlides.

```
public class SSLServer {

    ...

    try {
      SSLServerSocketFactory sslFactory = (SSLServerSocketFactory)
          SSLServerSocketFactory.getDefault();
      SSLServerSocket srvSocket = (SSLServerSocket) sslFactory.
          createServerSocket(4043);
      srvSocket.setNeedClientAuth(true);
      ...
```

També és possible configurar un servei de manera que demani als clients autenticar-se, però només de manera opcional. Si els clients no disposen de cap certificat, ja que no estan configurats per desplegar autenticació mútua, la negociació segueix igualment endavant. Per fer-ho, cal usar el mètode `setWantClientAuth`. Si es criden els dos mètodes dins del codi del programa, `setNeedClientAuth` sempre té prioritat, independentment de l'ordre com s'invoquin.

2.2.3 Gestió personalitzada de la informació criptogràfica

Les biblioteques d'SSL que incorpora Java permeten crear connexions segures de manera relativament simple. Aquestes s'encarreguen de gestionar de manera automàtica la gran majoria dels detalls vinculats a les dades criptogràfiques. Com a desenvolupadors, només cal indicar la ruta dels magatzems criptogràfics

corresponents, les seves contrasenyes, i ja està. En realitat, no cal saber els detalls sobre com funciona tot plegat internament.

Tot i així, sempre cap la possibilitat que el comportament per defecte de les biblioteques SSL de Java no siguin suficients pels requeriments d'una aplicació, ja que el seu funcionament per defecte és relativament simplista. Per una banda, les claus i certificats han d'estar sempre a un fitxer local. D'altra banda, el model d'autenticació es basa exclusivament en si el certificat aportat per un extrem ha estat emès per una entitat de confiança. Si és així, es permet la connexió i en cas contrari, es descarta. No hi ha possibilitat d'usar altres mecanismes més complexos per gestionar claus i certificats, o filar més prim a partir de la informació continguda al certificat per decidir si s'accepta la connexió.

Afortunadament, Java ofereix una via d'accés a les dades criptogràfiques a més baix nivell, permetent la personalització d'alguns aspectes vinculats a la gestió de la informació criptogràfica i a la validació dels certificats.

Inspecció de la identitat als extrems

Un cop s'ha establert una connexió mitjançant SSL, pot ser interessant poder consultar la informació continguda dins el certificat digital de l'altre extrem, ja sigui servei o client. Per exemple, poder saber saber quina és la identitat exacta de l'extrem a través de les dades contingudes al nom distingit del propietari o l'emissor del certificat: el seu nom (al camp *Common Name*), la seva organització (al camp *Organization*), etc.

> La negociació SSL només permet garantir que l'extrem disposa d'una identitat validada per un emissor de certificats de confiança, però no quina és aquesta identitat, o quines accions pot dur a terme.

Per tant, es pot donar el cas que, tot i saber que un extrem té accés i es pot connectar, també sigui necessari saber-ne la identitat exacta per actuar en conseqüència. El cas més fàcil de conceptualitzar és quan un client es connecta a un servei. Per exemple, en un cas molt senzill, potser es vol donar un aspecte personalitzat a l'aplicació, mostrant el nom de l'usuari. En un entorn més complex, pot donar-se el cas que, diferents usuaris que poden accedir a l'ítem, i que disposen de certificats admesos, disposin de diferents privilegis depenent del departament. Per tant, què poden fer depèn dels camps adients del certificat presentat.

Per consultar la informació vinculada a la identitat de l'extrem en una connexió segura SSL, existeix la classe `SSLSession`. Tota connexió segura disposa d'un objecte d'aquest tipus associat, accessible mitjançant la invocació del mètode `getSession()` sobre el sòcol segur de dades (l'objecte `SSLSocket`).

A la documentació de l'API de Java podeu trobar els mètodes de les classes `Certificate` i `X500Principal`, per tal d'accedir a la informació que contenen.

Entre els seus mètodes destacats podeu trobar:

- `String getCipherSuite()`: permet veure quin algorisme ha decidit usar Java per protegir les dades, com a resultat de la negocicació SSL.

- `Certificate[] getPeerCertificates()`: permet obtenir el certificat que ens ha enviat l'altre extrem amb qui s'estan intercanviant missatges. Per exemple, si s'executa al client, ens diu quin certificat ha enviat el servei, i viceversa en el cas d'autenticació mútua.

- `Principal getPeerPrincipal()`: obté el nom distingit dins del *Subject* del certificat de l'altre extrem. Això permet obtenir la identitat de l'extrem sense haver de processar el certificat. Normalment, retorna un objecte de tipus `javax.security.auth.x500.X500Principal`.

- `String getPeerHost()`: obté el nom de l'equip amb qui s'intercanvien missatges.

Personalització del gestor de confiança

També existeix la interfície `TrustManager`, que és aplicable a altres tipus de credencials a part de certificats.

Una altra opció força interessant que ofereixen les biblioteques d'SSL de Java és permetre redefinir els criteris per considerar si un certificat aportat per l'altre extrem a les comunicacions s'ha de considerar admissible o no. Això es fa a través de la implementació de la interfície `javax.net.ssl.X509TrustManager` i la integració de la nova classe al motor d'SSL.

Java ja proporciona una implementació per defecte, que és tot just la que executa el procediment mitjançant el qual es consulten les propietats del sistema `trustStore` i `trustStorePassword`. S'obre el fitxer corresponent i carrega la llista amb tots els certificats d'emissors de confiança que a partir de llavors, s'usa com a referència per veure si cal acceptar o no un certificat. Però en una implementació personalitzada es pot fer una classe que llegeixi els certificats d'un altre lloc, com un servei remot, o que apliqui altres polítiques de validació.

Per implementar la interfície `X509TrustManager` cal fer el codi dels següents mètodes:

- `void checkClientTrusted(X509Certificate[] certs, String tipus) throws CertificateException`: el codi que s'escriu en aquest mètode serveix per indicar si un certificat subministrat per un client remot es considera vàlid o no. Per tant, només té sentit implementar-lo a un servei. Si es considera correcte, el mètode simplement acaba (no hi ha valor de retorn). Per indicar que el certificat no es considera vàlid, es llança una excepció del tipus `CertificateException`. El seu primer paràmetre, `certs`, és un *array* amb el certificat del client i del seu emissor. En cas de ser un certificat autosignat, només hi haurà un únic certificat. El paràmetre `tipus` és una cadena de text que indica l'algorisme emprat per l'autenticació. Per exemple, si les claus del client són RSA, seria el text `RSA`.

- `void checkServerTrusted(X509Certificate[] chain, String authType) throws CertificateException`: aquest és el cas per validar un certificat obtingut des d'un servei remot. Per tant, només té

sentit posar codi a si s'usa a una aplicació client. La resta de detalls són exactament els mateixos que al mètode anterior.

- `X509Certificate[] getAcceptedIssuers()`: el codi d'aquest mètode ha de retornar un *array* amb una llista dels certificats d'emissor considerats de confiança.

El següent codi mostra un gestor personalitzat molt simple. Per una banda, com que el mètode `checkServerTrusted` sempre llança una excepció, això vol dir que si el gestor s'usa a un client, aquest mai acceptarà cap certificat d'un servei com a vàlid. D'altra banda, ja que el mètode `checkClientTrusted` no fa res i, per tant, mai llança cap excepció, això vol dir que si s'usa el gestor en un servei, aquest acceptarà qualsevol certificat subministrat per qualsevol client. Alerta, ja que aquest mètode comprova la validesa global del certificat, no només si el seu emissor és de confiança. Per tant, amb codi buit, accepta certificats encara que tinguin una signatura incorrecta (i per tant, falsos) o una data d'expiració invàlida, s'acceptarà.

```
public class MyTrustManager implements X509TrustManager {

  @Override
     public void checkClientTrusted(X509Certificate[] chain, String authType)
                                    throws CertificateException {
     }

  @Override
     public void checkServerTrusted(X509Certificate[] chain, String authType)
                                    throws CertificateException {
       throw new CertificateException();
     }

  @Override
     public X509Certificate[] getAcceptedIssuers() {
         return new X509Certificate[0];
     }
}
```

En cas que un dels mètodes de validació no s'hagi d'implementar, ja que no s'usarà, sempre és millor que al seu codi es llanci la `CertificateException`. O sigui, si s'està creant un gestor per una aplicació client, fer que el codi del mètode `checkClientTrusted` tingui només el codi `throw new CertificateException()`. Això evita que si s'usa per error en un tipus d'aplicació incorrecta, aquesta mostri un error enlloc d'acceptar qualsevol certificat sense adonar-vos-en.

Una biblioteca oculta no està documentada a l'API de Java, però es pot accedir igualment a les seves classes.

Normalment, una implementació de *X509TrustManager* tindrà mètodes amb codi. Ara bé, una dificultat important al fer-ho és que, a part de les comprovacions personalitzades que vosaltres vulgueu fer, també ha de fer totes les vinculades a la correcció del certificat. En aquest sentit, no només cal comprovar que l'emissor és de confiança. A més a més, cal veure si les dades contingudes al certificat són correctes: la signatura digital, la data d'expiració, etc. Això no és trivial de fer, ja que significa aplicar operacions criptogràfiques. Afortunadament, Java ofereix la classe `Validator` a la biblioteca oculta `sun.security.validator`. Aquesta

permet validar certificats fàcilment. De fet, és la que usa el gestor de confiança per defecte.

Per instanciar una classe validadora, cal usar el mètode estàtic `getInstance` de la classe `Validator`. Aquest té tres paràmetres. Pel cas de certificats digitals, el primer sempre és el definit per la constant `Validator.TYPE_PKIX`. El segon també es basa en una altra constant, que indica si estem executant la validació a una aplicació client (`Validator.VAR_TLS_CLIENT`) o a un servei (`Validator.VAR_TLS_SERVER`). El tercer és la llista de certificats de confiança. Els certificats d'aquesta llista poden provenir d'on vulgueu, no necessàriament d'un únic magatzem de claus, com passa al gestor per defecte.

Un cop fet, la validació del certificat es fa amb el mètode `validate`. El paràmetre és un *array* amb el conjunt de certificats aportat per l'altre extrem de les comunicacions, i que cal validar. De fet, aquest valor ja us el proporciona el motor d'SSL a través del primer paràmetre de `checkClientTrusted` i `checkServerTrusted`.

Concretament, el mètode `validate` comprova:

- El format del certificat
- La data d'expiració
- Si el seu emissor es correspon a algun de la llista proporcionada
- La firma digital del certificat, a partir de la clau pública de l'emissor localitzat

Si alguna cosa no és correcta, llança una excepció del tipus `CertificateException`.

El següent codi mostra un exemple de gestor de confiança una mica més complex, pensat només per una aplicació client. Aquest carrega els certificats d'emissors de confiança des d'un magatzem de claus i els usa per validar el certificat d'un servidor. Si el certificat és correcte, mostra per pantalla el del seu emissor. Estudieu-lo amb atenció per veure com s'extreuen els certificats de confiança del magatzem de claus original, i després s'usen als diferents mètodes per validar el certificat d'un servei.

```
public class MyTrustManager implements X509TrustManager {

  private ArrayList<X509Certificate> certs = new ArrayList<X509Certificate>();

  public MyTrustManager() throws Exception {
    KeyStore keyStore = KeyStore.getInstance("JKS");
    keyStore.load(new FileInputStream("SSL/ClientKeyStore.jks"), "client".
        toCharArray());
    Enumeration<String> aliases = keyStore.aliases();
    while(aliases.hasMoreElements()) {
      String a = aliases.nextElement();
      if (keyStore.isCertificateEntry(a)) {
        certs.add((X509Certificate)keyStore.getCertificate(a));
      }

    }
  }
```

```

  @Override
     public void checkClientTrusted(X509Certificate[] chain, String authType)
                 throws CertificateException {
       throw new CertificateException("Aquest gestor només és per aplicacions
           client.");
     }

  @Override
     public void checkServerTrusted(X509Certificate[] chain, String authType)
                 throws CertificateException {
         try {
           Validator val = Validator.getInstance(Validator.TYPE_PKIX, Validator
               .VAR_TLS_CLIENT, certs);
           X509Certificate[] result = val.validate(chain);
           for (X509Certificate c: result) {
             System.out.println("El següent certificat de confiança valida el
                 servei:");
             System.out.println(c);
           }
         } catch (Exception e) {
           throw new CertificateException("El certificat del servei no és de
               confiança.");
         }
     }

  @Override
     public X509Certificate[] getAcceptedIssuers() {
       X509Certificate[] issuers = new X509Certificate[certs.size()];
       for (int i = 0; i< issuers.length; i++){
         issuers[i] = certs.get(i);
       }
       return issuers;
     }
}
```

Un cop es disposa d'una implementació correcta de la interfície `X509TrustManager`, cal integrar-la al motor d'SSL de les biblioteques de Java, de manera que reemplaci al comportament per defecte. Això es fa amb l'ajut de la classe `SSLContext` i el seu mètode `init`. Un petit inconvenient és que aquest sistema obliga a indicar quin gestor de claus, o *KeyManager*, cal usar.

La interfície usada per personalitzar un gestor de claus és la `X509KeyManager`.

De la mateixa manera que amb el gestor de confiança, Java permet personalitzar també el mecanisme de gestió de claus, associat a les propietats `keyStore` i `keyStorePassword`, de manera que s'obre la porta a usar altres sistemes que un magatzem de claus amb una única entrada amb una clau privada. Ja que la interfície associada és força més complicada que la del gestor de confiança, la solució recomanada és crear una instància del gestor de claus per defecte i assignar-lo, enlloc de fer-ne un de personalitzat.

La integració es fa tal com mostra el següent fragment de codi, generant un objecte `SSLSocketFactory` especial. En lloc d'usar el que es proporciona per defecte amb la crida estàtica `SSLSocketFactory.getDefault()`, se'n genera un a partir de la classe `SSLContext`. El mètode clau és `init`, que és el que serveix per configurar un context d'SSL perquè utilitzi gestors personalitzats. El tercer paràmetre és un objecte `SecureRandom` qualsevol.

```
//Inicialització d'un gestor de claus per defecte.
  //Les antigues propietats "keyStore" i "keyStorePassword" són paràmetres
      directament
  KeyManagerFactory kmf = kmf = KeyManagerFactory.getInstance("PKIX");
```

```
4   FileInputStream fin = new FileInputStream("SSL/ClientKeyStore.jks");
5   KeyStore ks = KeyStore.getInstance("JKS");
6   ks.load(fin, "client".toCharArray());
7   kmf.init(ks, "client".toCharArray());
8
9   //Creació del gestor de confiança personalitzat
10  TrustManager[] myTM = new TrustManager[] { new MyTrustManager() };
11
12  SSLContext sslContext = SSLContext.getInstance( "SSL" );
13  sslContext.init( kmf.getKeyManagers(), myTM, new java.security.SecureRandom()
        );
14
15  SSLSocketFactory sslFactory = sslContext.getSocketFactory();
16  Socket cliSocket = sslFactory.createSocket("localhost", 10000);
17  ...
```

Un cop fet, tots els sòcols generats a partir d'aquest context usarà els nou gestor de confiança. Pel cas d'un sòcol de servei, caldrà usar el mètode `getServerSocketFactory()` de `SSLContext` a la darrera línia.