MEMÒRIA DE L'APLICACIÓ XAT SEGUR v0.1

1 – OBJECTIU

1.1 Presentació del problema

L'objectiu de l'aplicació es facilitar una eina de comunicació entre usuaris (o Xat) però utilitzant algoritmes de xifratge simètric per tal de codificar els missatges enviats d'un usuari a l'altre, fent aquesta informació inaccessible per a usuaris no desitjats.

La eina ha de permetre a un usuari d'establir una connexió amb un altre usuari, de manera que la comunicació entre ambdós es faci amb un xifrat de dades. A més cada usuari ha de poder mantenir més d'una conversa alhora.

Es vol realitzar una interfície gràfica senzilla però complerta que permeti als usuaris realitzar diferents tasques de comunicació, control d'usuaris i multi converses, que es desenvoluparà al final del projecte i si el factor temporal ho permet.

Es vol generar una sèrie de fitxers per tal de mantenir les claus ja acordades amb altres usuaris, per tal d'economitzar el temps que es triga a decidir aquestes claus, i per tant, en quant ja s'hagi realitzat una conversa amb un usuari, es vol guardar la clau i utilitzar-la en cas d'iniciar una conversa en un altre moment.
De la mateixa manera, hi ha la intenció inicial d'utilitzar XML per tal d'extreure la informació de configuració que en un principi es hardcodeja a l'aplicació, com temps de TIMEOUT, IP de destí o port, per tal de fer l'entorn parametritzable, però en no ser un objectiu del projecte, també serà un punt que restarà obert pel cas d'assolir els objectius principals dins del temps establert.

S'utilitzarà el mètode Diffie-Hellman per tal de cercar la clau privada que codifiqui/descodifiqui els missatges.

S'utilitzarà un algoritme de xifratge simètric de 128 bits, com 3DES, per tal de realitzar aquesta codificació.

Caldrà encapsular les funcionalitats de cada part de l'aplicació (intercanvi de claus, xifratge, clients...) en diferents classes que ens permetin reutilitzar el codi des de diferents punts.

El client del Xat ha de poder realitzar les funcions bàsiques d'un Xat.

Cada client ha de poder mantenir una conversa amb un o més clients a l'hora, de manera que el procés client ha de ser capaç de generar diversos Threats cada cop. Un per cada usuari amb qui es vulgui contactar.
Aquesta diversitat de processos ha d'estar ben gestionada i no s'ha de permetre que es perdin missatges.

Els valors públics rebuts d'altres clients en el moment de realitzar la connexió amb el procés Diffie-Hellman, seran emmagatzemats en un fitxer per tal de poder reutilitzar-los en el moment de posar-nos en contacte amb els demés usuaris.

1.2 Protocols i comunicació

La idea general de l'aplicació es facilitar a l'usuari un producte de comunicació de missatges per missatgeria instantània, com poden ser MSN Messenger, Yahoo! Messenger, ICQ o Jabber. La missatgeria instantània, a diferència dels programes que usen el protocol IRC (*Internet Relay Chat*) com mIRC, X-Chat, Chatzilla, utilitza el protocol TCP-IP per tal de realitzar l'enviament de missatges.

El protocol TCP-IP (Transmission Control Protocol-Internet Protocol) està dissenyat per enrutar la informació i te un elevat grau de fiabilitat. Es a més el protocol d'Internet i permet enllaçar màquines de tot tipus sense cap mena problema. Es el protocol de comunicacions més potent en grans transferències de dades, sent pitjor en xarxes de poc tràfic on NetBEUI o IPX/SPX son més senzills de configurar i mantenir.

El Xat que volem desenvolupar utilitzarà aquest protocol de transport per enviar els seus missatges, però per tal de garantir que la informació que els usuaris envien es complertament segura, es vol enviar codificada amb els mètodes criptogràfics adequats.

Les trames del protocol TCP-IP son enviades a la Xarxa de manera desendreçada i passen per molts computadors connectats entre si, que redrecen les trames cap al destí. Aquests computadors poden ser computadors maliciosos que mitjançant *sinffers* als ports de connexió, poden capturar els missatges associats a les trames, i aconseguir desxifrar-ne la informació.
Per tal que això no succeeixi es vol codificar la informació enviada mitjançant un algoritme de xifratge simètric de 128 bits.

1.3 Xifratge simètric

Els algoritmes de xifratge simètric utilitzen una clau per xifrar i desxifrar els missatges. Aquesta clau es coneguda tant per l'emissor com pel receptor, de manera que la seguretat recau realment en la clau a utilitzar, i no tant en quin algoritme de xifratge que s'està usant.

Les claus usades a l'actualitat son de 128 bits, la qual cosa dona una quantitat de 2^{128} possibles claus diferents sobre un missatge obtingut per tècniques d'Sniffing. Aquest nombre és realment elevat i aquí és on radica la seguretat d'aquest mètode.
Tenint amb compte la capacitat de càlcul que poden arribar a tenir els processadors actuals i la quantitat de possibilitats que caldria calcular, fan que aquesta sigui una feina del tot inabastable. I a més, tenint amb compte la velocitat a la que augmenta aquesta capacitat de procés, ho seguirà sent durant força temps.

1.4 Diffie-Hellman

Per tant s'haurà d'escollir un protocol d'intercanvi de claus que sigui el suficientment secret i fiable, i el que reuneix tots els requisits (i per això el més usat avui dia) és el protocol Diffie-Hellman.

Aquest protocol basa la seva seguretat a la extremada dificultat (no demostrada) de calcular logaritmes discrets en camp finit.

El funcionament és molt senzill;
s'estableixen un nombre primer *p* i una base generadora (*g*) que pertanyi a *p*. L'usuari A escull un nombre **n** que sigui <p i calcula $\mathbf{X=g^n \bmod p}$ i envia **X** a l'usuari amb qui vol iniciar la comunicació, B.

De la mateixa manera, B escull un **m** < p i calcula $\mathbf{Y=g^m \bmod p}$ i envia **Y** al usuari A.

Cal adonar-se doncs, que ara tenim $X^m = (g^n)^m = g^{mn} = (g^m)^n = Y^n$, i per tant $Y^n \bmod p = X^m \bmod p$.

Així doncs A pren el nombre Y que envia l'usuari B i fa $Y^n \bmod p$ i per la seva banda, B pren el nombre X que envia l'usuari A i fa $X^m \bmod p$, i tots dos tenen un nombre igual que els serveix per codificar/descodificar els missatges.

Cal adonar-se que els nombres g i p i X i Y son públics i qualsevol pot trobar-los, però que es del tot impossible cercar **n** i **m** que son els nombres secrets escollits per cada usuari, i es sobre que recau la seguretat de tot el sistema.

1.5 Aplicacions

Per tant, caldrà generar un mètode de càlcul del protocol Diffie-Hellman que es posi d'acord en qüestió de claus públiques entre usuaris i que esculli els nombres adients per acabar cercant la clau privada amb la que es codificarà/descodificarà tota la informació.

Apart caldrà generar un mètode que posi en pràctica l'algoritme de xifratge simètric de 128 bits que realitzi aquesta acció de codificació.

2 – TECNOLOGÍA - FONAMENTS

2.1 Eines a utilitzar en el desenvolupament

Per tal de procedir a la codificació de l'aplicació utilitzarem el llenguatge de programació Java. Concretament la versió 1.4.2 del J2RE de Java.

Dintre de Java, s'utilitzaran les llibreries java.net i java.io pel desenvolupament de les comunicacions i de les interfícies gràfiques respectivament.

S'utilitzarà l'algoritme de cerca de clau privada Diffie-Hellman per tal d'aconseguir una clau privada a partir de la clau pública que compartiran els diferents clients.

S'utilitzarà un algoritme de xifratge simètric de 128 bits com 3DES, AES, Blowfish, IDEA... per la codificació/descodificació dels diferents missatges enviats i rebuts per part dels clients.

La presentació del projecte es realitzarà amb la eina de Microsoft PowerPoint.

2.2 Programació Orientada a Objectes

Dintre de l'àmbit de la programació d'aplicacions, ens trobem dos grans mètodes:

- Programació Estructurada
- Programació Orientada a Objectes

La programació *Estructurada* és aquella programació de caràcter "lineal" on un programa comença i acaba seguint una execució, una darrera l'altre, de totes les instruccions del mateix, seguin els tres grans tipus d'estructures:

- Seqüencial
- Selectiva
- Iterativa,

sense permetre la utilització d'estructures de transferència incondicional (tipus GOTO).

Per la seva banda, la programació *Orientada a Objectes*, és aquella programació que encapsula el codi en diferents grups anomenats Objectes que no son sinó una representació de models reals.

Aquests Objectes, anomenats Classes, contenen atributs (dades) i mètodes (funcions), de manera que tota classe te els seus propis atributs i els seus propis mètodes, o manera de tractar els atributs.
Els objectes son definits per la combinació dels seus estats (atributs), el seu comportament (mètodes) i la seva identitat (propietat de l'Objecte que el diferencia de la resta).

D'aquesta manera, amb la programació *Orientada a Objectes* es tenen diferents objectes, de diferents tipus, que realitzen diferents accions i contenen diferents tipus de dades, de manera que uns interactuen amb els altres per assolir l'objectiu del programa.

Aquest tipus de programació és molt potent, doncs estructura el codi de manera que és senzill d'entendre i per tant, de mantenir i programar, i a més permet la reutilització de codi d'una manera molt eficaç.

Així doncs, un programa *OO* és una estructuració d'objectes interactuant entre ells amb un objectiu comú, de manera que la manera de relacionar-se permetrà obtenir un resultat o altre.

Cada objecte te mètodes propis i no accessibles per cap altre més objecte (private), mètodes accessibles a alguns objectes de classe o descendents (protected), i mètodes accessibles per tots (public), que son les interfícies de relació entre un objecte i altre.

D'aquesta manera diferents objectes utilitzen mètodes d'altres per tal de combinar dades o fer càlculs i obtenir així un resultat.

La gran potència de la programació *OO* és l'herència.

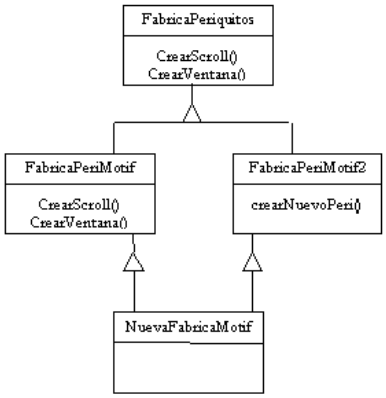
2.2.1 Herència

L'herència a la programació *OO* és la capacitat que tenen uns objectes d'utilitzar els atributs i mètodes d'un altre o altres objectes, modificar-los i crear-ne de nous.

A partir d'un Objecte determinat, es pot realitzar una herència per tal de millorar-ne o modificar-ne alguns aspectes per tal d'adaptar la funcionalitat inicial de l'Objecte del que s'hereta (anomenat pare) a les necessitats pròpies del nou projecte a desenvolupar.

Aquest comportament dona una potència inimaginable a la programació, doncs permet reutilitzar codi amb una gran facilitat. Ara però, això obliga al programador a fer una codificació correcta i coherent, i fa que un objecte hagi de ser força ben plantejat d'inici, determinant de manera clara quins mètodes han de ser públics, privats i protegits, així com les funcionalitats i les relacions entre ells.

És possible realitzar herència múltiple (això no passa a alguns llenguatges com Java, tot i que hi ha la manera d'aconseguir-ho implementat Interfaces) per tal de combinar les possibilitats de dos objectes en un de sol.

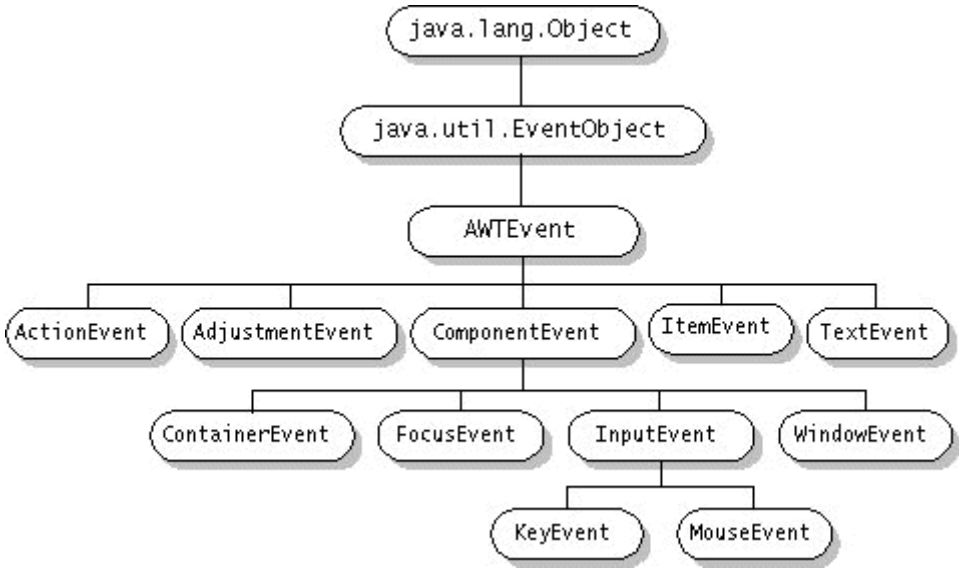


Apareix el concepte de mètodes abstractes, que son aquells que han de ser implementats en els Objectes descendents (fills) dels objectes que els contenen.
Un objecte que te tots els seus mètodes abstractes és una interfície. Les interfícies realment son objectes que marquen com ha de treballar, però no implementen la manera de fer-ho. Pot ser per adaptar el codi a depenent quin entorn, o potser, com a Java, per tal de realitzar herència múltiple.

L'**herència múltiple a Java** es fa implementant interfaces. Una classe Java només pot heretar d'una altra Classe, però una Classe pot implementar més d'una interfície, i d'aquesta manera es pot generar herència múltiple.
D'aquesta manera es genera una Interfície que hereti de la superclasse inicial, i després s'implementa aquesta Interfície al fill que ja està heretant d'una altra classe.

Com a paradigma d'herència tenim el propi llenguatge Java, doncs les classes Java estam estructurades de forma que totes hereten d'una superclasse comuna: Object.

Tota classe utilitzada a Java hereta d'algun altre objecte, i el pare de tots és Object, portant a l'extrem el concepte d'herència:



Com a exemple, el clàssic de la classe vehicle:

Tenim una classe vehicle que te com a atributs nombre de rodes i places i com a mètodes afegir_passatger, treure'n...

Una possible especialització seria crear per herència els vehicles moto, cotxe, camió, on definiríem nous atributs, com nombre de portes al cotxe i al camió, i nous mètodes com calcular la càrrega del camió... d'aquesta manera utilitzaríem els mètodes comuns des de la classe vehicle, i els propis de cada tipus de vehicle.

2.3 Client/Servidor

L'arquitectura Client/Servidor és una manera de dividir les responsabilitats d'una aplicació de manera que es divideix entre una part gràfica d'usuari i una part de gestió de dades.

L'arquitectura separa clarament el que son tasques a realitzar pels usuaris del sistema, client, del que son pròpiament les estructures internes de programa, moviment d'informació, connectivitat... que son els processos servidor.

Un Servidor dona serveis a una sèrie d'usuaris que son els que exploten el sistema, ja sigui de Base de Dades, de Xarxa Interna, de Internet... una part client utilitza les eines que proporciona la part servidora.

En el nostre cas, el Servidor serà l'encarregat de posar en contacte els diferents usuaris, i de fer de punt d'entrada per a totes les comunicacions, readreçant-les cap als destins que calgui en cada moment.
Per la seva part, els clients s'encarreguen d'enviar i rebre missatges utilitzant els canals de comunicació que s'obren gràcies a la classe Socket de Java.

El programa ha d'actuar a mode de **Client/Servidor** amb *Client pesant*, de manera que el Servidor resti a l'escolta pel port 2001 (com a homenatge a Kubrik i al fantàstic Sir Arthur C. Clarke) de totes les entrades que es produeixin, i els clients enviïn informació a aquest port del Servidor (cal pensar

que els primers 1024 ports es reserven normalment per processos estàndard o del Sistema Operatiu).

El Client es un *Client Pesant* doncs s'encarregarà, en principi (i a excepció de modificacions durant el desenvolupament) de realitzar el procés de codificació/descodificació de missatges, previ intercanvi de claus i elecció de clau privada per realitzar aquest xifratge.

Per tal d'assolir aquesta comunicació entre diferents processos, utilitzarem el potent paquet java.net de Java, el qual ofereix les classes Socket i ServerSocket.

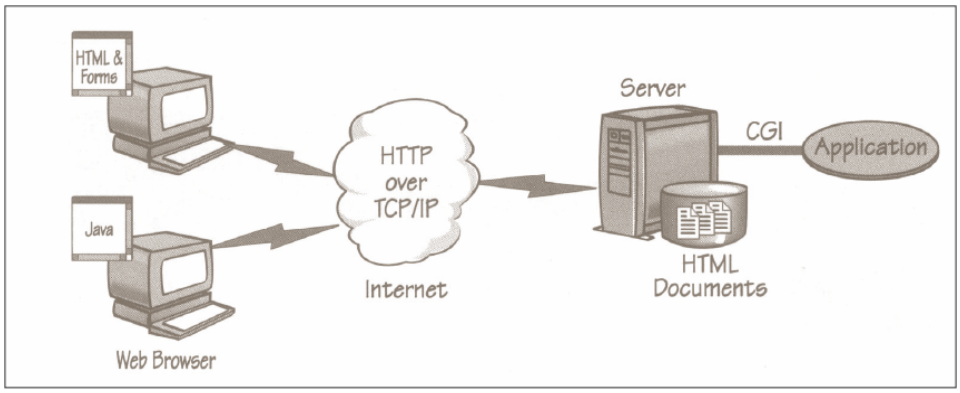
2.4 Socket

2.4.1 Introducció

Fonaments de Xarxes

El protocol IP és el protocol de xarxa utilitzat per tal d'enviar dades entre els computadors connectats a Internet o entre computadors que pertanyen a una mateixa Intranet.
Els computadors que formen part d'aquestes xarxes s'identifiquen mitjançant una o varies adreces IP, que és una adreça de 32bits (4 bytes segons el protocol IPv4). Cada un d'aquests bytes son un sencer entre 0 i 255.

Aquesta adreça IP també pot ser representada per un DNS (*Domain Name System*) que és un nom que es relaciona amb aquesta IP.
A cada servidor de dominis, hi ha una relació entre DNS i la IP corresponent.



PORTS

D'aquesta manera podem identificar un computador destí de la nostra comunicació per la seva IP o DNS, però no el servei al qual volem accedir.
Per tal de accedir a una aplicació o altre remotament, els computadors ofereixen PORTs d'entrada, als quals hi ha els programes a l'escolta.
Depenent del port a que accedim, accedirem a una aplicació o una altra.
D'aquesta manera, el servei HTTP escolta peticions al port 80, i si volem accedir al servei HTTP d'un computador, ho podem fer amb IP:80 (o IP:8080 que és un PORT mapejat que s'utilitza molt sovint en comunicacions via HTTP).

D'aquesta manera, hem de fer que el nostre Servidor escolti a IP:PORT totes les peticions que rebi per tal de posar-nos en comunicació amb ell i assolir el nivell de comunicació.

De fet un Port és una adreça de 16 bit associat normalment a un protocol d'aplicació.

Cada Interfase d'un computador està dividit en 65536 PORTS diferents, estant els primers 1024 reservats per aplicacions de sistema.

Existeixen alhora dos protocols de transport: el TCP i l'UDP, que s'encarreguen d'enviar dades d'un port a l'altre dins d'una xarxa de computadors, per tal de fer possible les comunicacions entre aplicacions dels diferents computadors.

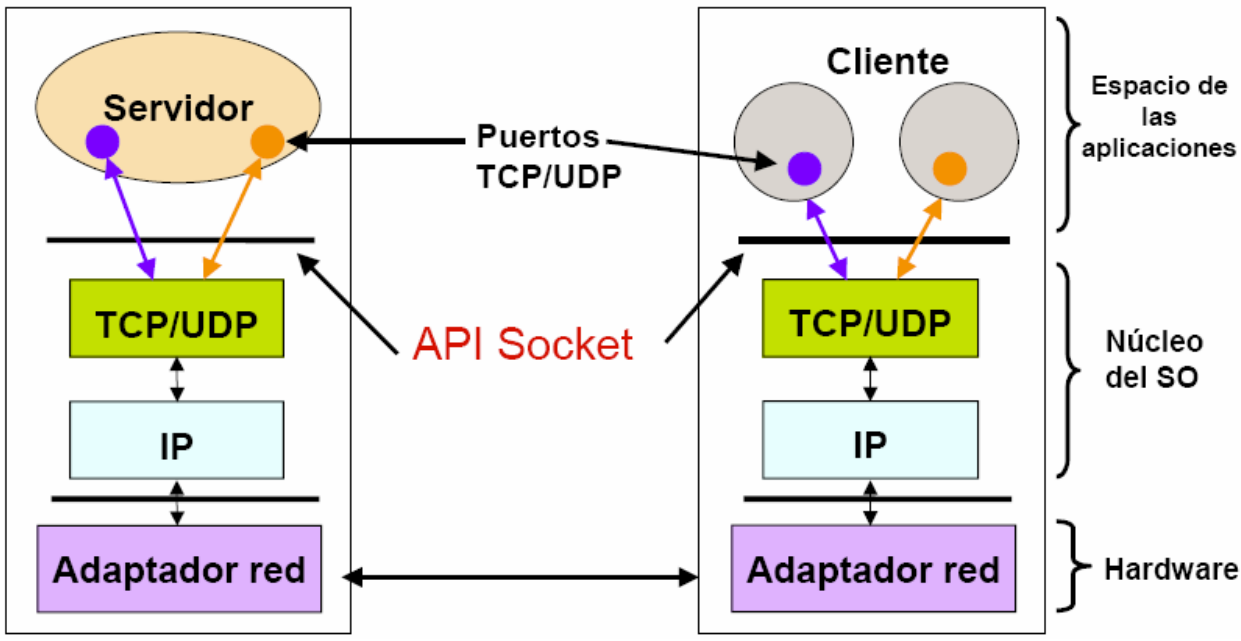
Les aplicacions de xarxa han de tenir amb compte que el seu repte principal és la comunicació entre diferents parts de la aplicació, o la comunicació entre diferents aplicacions, per tal d'assolir l'objectiu final pel que han estat programades.

Per tal de possibilitar aquesta comunicació entre diferents sistemes o diferents parts d'un sistema, cal establir una sèrie de paràmetres d'actuació o maneres de procedir, doncs una aplicació d'un sistema Unix pot estar interessat en comunicar-se amb una aplicació d'un sistema Mac-OS i per això, calen una sèrie de **protocols** que permeten aquesta comunicació.

A les telecomunicacions, els protocols s'agrupen en **famílies de protocols**, i d'elles la TCP/IP és la més coneguda, doncs és la família sobre la que es basa tota la xarxa Internet.

Gràcies a aquesta família, un telèfon portàtil és capaç de connectar-se amb un servidor HTTP i accedir a continguts WAP, o podem connectar-nos via FTP a altres computadors amb altres SOs...

Dins de la família TCP/IP, els protocols més coneguts son el TCP i el UDP.

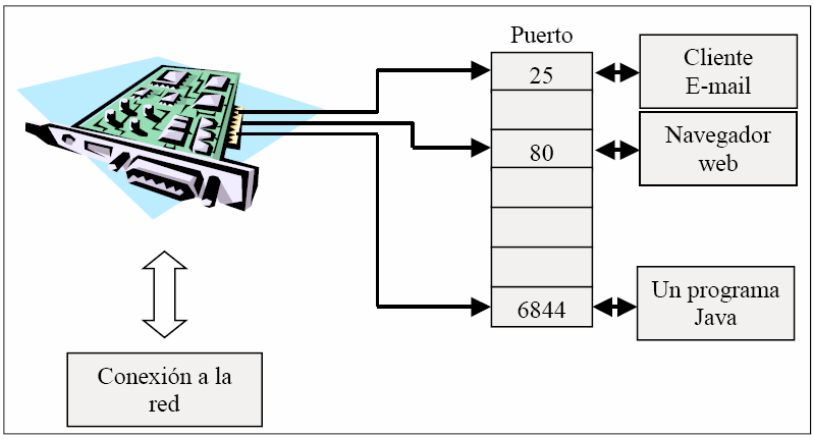


Per tant, en quant al desenvolupament d'una aplicació que utilitza la xarxa com via de comunicació entre processos, o com és el nostre cas, entre clients d'un mateix servei, necessitarem establir una sèrie de **protocols** de comunicació entre les diferents parts, i utilitzarem una sèrie de **ports** per tal de realitzar aquesta comunicació sobre el servidor (IP) que dona el servei.

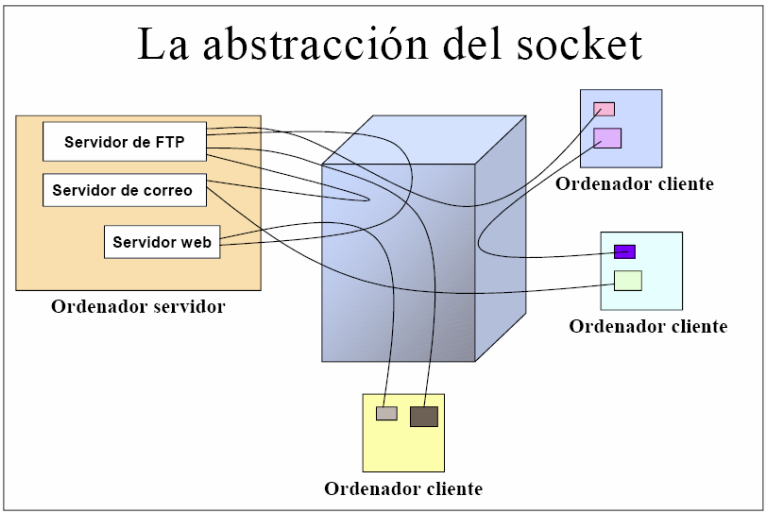
Socket

Aquesta utilització dels PORTs d'un Servidor per la comunicació entre aplicacions, és el que s'aconsegueix amb la utilització dels **sockets**.

Els sockets son les representacions dels PORTs en el moment de la programació, i cada socket es connecta amb un PORT per tal de comunicar-se amb el socket-port del client que es connecta a la màquina servidora.

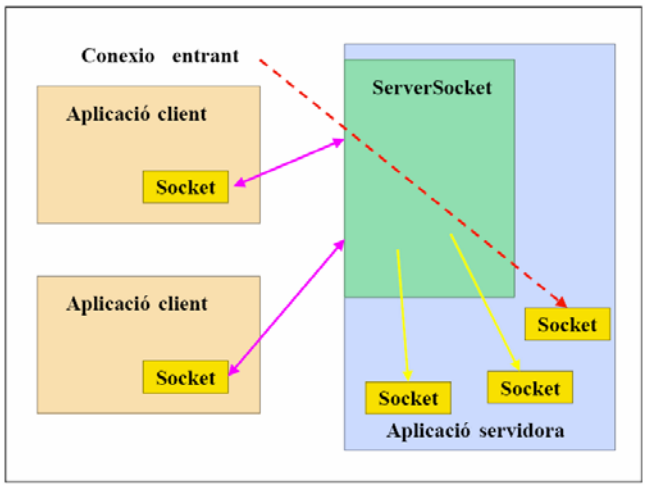


Cada socket de cada màquina és una espècie d'entrada al tub que connecta una màquina amb una altre. Amb el socket podem enviar i rebre informació d'una banda a l'altre d'aquest "tub" imaginari.



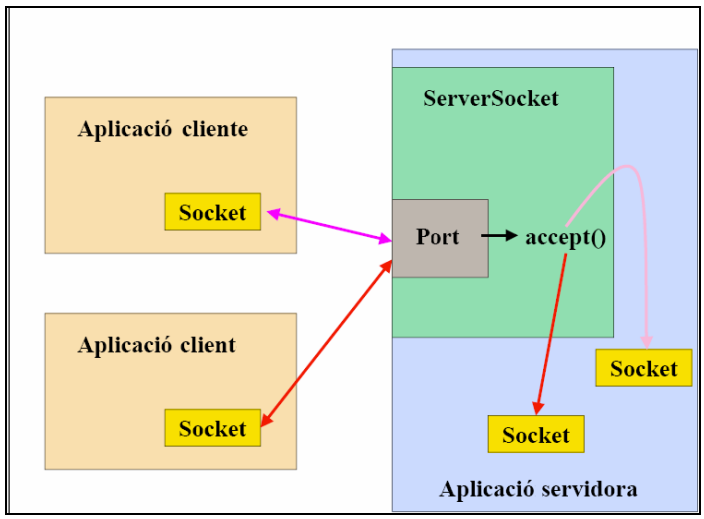
Un **SOCKET** és un punt terminal o un extrem a un canal de comunicació entre dues aplicacions. Les aplicacions es comuniquen mitjançant l'enviament i la recepció de **missatges** mitjançant **sockets.**

La paraula **socket**, que literalment significa "connector" o "endoll", prové directament dels panells de connexions que utilitzaven les centraletes telefòniques antigament, on les connexions es realitzaven a ma.



Els sockets es separen en dos tipus:

- Sockets Actius : que son aquells que poden enviar i rebre informació.
- Sockets Passius: que son aquells que resten esperant una connexió entrant, i quan aquesta es produeix li assignen un Socket Actiu.



Bruce Eckel descriu els sockets, al seu llibre *Thinking in Java 3rd Edition* com *"l'abstracció del software utilitzada per a representar els terminals d'una connexió entre dues màquines. Per una connexió donada, existeix un socket a cada màquina, i hom es pot imaginar un fil hipotètic corrent entre ambdues, amb cada extrem del fil connectat a un socket de cada màquina. Per descomptat, el hardware físic i el cablejat entre màquines és complertament desconegut. El punt fonamental de l'abstracció és que no necessitem conèixer més del necessari."*

Els Sockets son creats pel SO i ofereixen una API (*Aplication Programming Interface o interfície de programació d'aplicacions*) mitjançant la qual les aplicacions poden enviar missatges a altres aplicacions, tant locals com de xarxa. Les operacions dels Sockets son, per tant, crides al SO. Son part del nucli del SO.
En llenguatges OO, les classes de sockets son implementades sobre les pròpies funcions oferides per l'SO.

Per tal que es produeixi comunicació, la part client ha de connectar els seus Sockets als Sockets de la part Servidor de l'aplicació, seguint les següents passes a tota comunicació TCP:

- Es creen els Sockets de Client i Servidor
- El Servidor estableix el PORT que ofereix el servei.
- El Servidor roman a l'escolta de les peticions que puguin arribar al port establert a l'apartat anterior.
- Un client connecta amb el Servidor.
- El Servidor accepta la connexió.
- Es realitza el intercanvi de dades.
- O el Client o el Servidor tanquen la connexió.

Cal esmentar una sèrie de particularitats per possibilitar el funcionament del procés:

- El servidor ha d'estar arrencat abans que els clients intentin connectar-se amb ell, doncs és el que proporciona el port d'entrada
- El Client ha de generar un Socket actiu on s'especifiquin la IP del Servidor i el Port des del qual es proporciona el servei.
  Mitjançant el Socket el client comença una connexió TCP amb el servidor
- El intent de connexió TCP del client genera un Socket actiu al Servidor, que serà exclusiu per la comunicació amb aquest client durant el temps de vida d'aquesta comunicació, i no pot ser assignat a cap més client.
- Finalment s'obté el canal de comunicació que va des del Socket del client fins al Socket Actiu del Servidor.

Java

Java ha agafat força popularitat degut a la seva clara orientació a Internet, popularitat evidentment merescuda si tenim amb compte la interface orientada a objectes de sockets, la qual facilita moltíssim la tasca de crear aplicacions en Xarxa.

Java va ser el primer llenguatge de programació en el qual l'entrada i sortida des de Xarxa es realitzava igual que des de fitxer. Així doncs, simplement creant uns canals d'entrada i sortida sobre els mètodes que ofereix Java, podem aconseguir llegir/escriure missatges a un altre procés d'un altre computador mitjançant la Xarxa.

Aquestes classes son molt senzilles d'utilitzar, a diferència d'altres llenguatges de programació com C o C++, que son realment eficaços per treballar en aplicacions de xarxa, però el codi necessari per tal de treure'n tot el suc, és realment complicat i susceptible a errors si el comparem amb el seu equivalent a Java.

Per tant la senzilla utilització d'aquests potentíssims canals de comunicació entre aplicacions i/o computadors serà el que passarem a estudiar tot a continuació:

2.4.2 *Sockets* a Java

Com s'ha comentat, les comunicacions a Java son realment senzilles d'aconseguir, degut bàsicament a les classes que el llenguatge proporciona per tal de fer accessibles els ports del sistema, relacionant-los amb les aplicacions, i facilitant-ne la connectivitat per tots aquells programes o parts del programa que tinguin la necessitat de connectar-s'hi.

A Java les comunicacions es basen en els sockets, doncs aquests permeten comunicar-se a través de fluxos de dades com si d'operacions d'E/S d'arxius es tractés.

Depenent de si el servei utilitza protocols de comunicació TCP o UDP, els sockets es divideixen en:

- sockets de flux (TCP)
- sockets de datagrama (UDP)

En els *sockets de fluxe* es te un objecte socket a la part client, i un o més objectes socket associats a un objecte ServerSocket a la part servidor.
La E/S es realitza utilitzant els mètodes InputStream i OutputStream dels objectes Socket.

Per contra, els *sockets de datagrama* permeten un intercanvi d'informació sense connexió, on el missatge de l'emissor es descompon en paquets UDP (datagrames UDP), i cada datagrama s'envia de la manera que sigui i pel canal que sigui. L'emissor realment ignora si els datagrames han arribat correctament o ni tant sols si han arribat al destinatari.

Ens centrarem als *sockets de flux*, doncs son els utilitzats en l'aplicació de **Chat Segur** que ens ocupa, doncs com és obvi per tractar-se d'una aplicació de converses electròniques, els clients estan connectats entre sí.

En el cas dels *sockets de flux,* el Servidor arrenca un ServerSocket, o Socket passiu. És anomenat ServerSocket doncs s'executa sempre a la part Servidor, doncs és el Socket que inicia la comunicació i el que resta escoltant peticions de connexió per part dels possibles clients que vulguin utilitzar els serveis que proporciona.

Ara be els Sockets actius també actuen a la part de Servidor com a canal de comunicació amb els Sockets de la part de client, per tant el nom de ServerSocket, si no incorrecte, és almenys, confús.

Així doncs, l'aplicació a la part del Servidor, genera un ServerSocket o Socket passiu que escolta a un port determinat, amb la instrucció:

ServerSocket serverSocket = new ServerSocket (PORT);

Posteriorment aquest serverSocket te un mètode que accepta les connexions entrants, i si la connexió és correcte genera un Socket, que es connecta amb el Socket client que ha realitzat la connexió:

Socket = new serverSocket.accept ();

A aquest nou socket se'l assigna un PORT lliure, normalment correlatiu a partir del port PORT on s'inicia la comunicació.

Aquesta assignació dinàmica de ports es realitza de manera seqüencial fins que no queden ports lliures. En aquest moment l'aplicació roman pendent d'assignar el Port que s'alliberi a cada moment, no permetent connectar cap entrada nova.

Per part del client, el que cal fer es connectar el nou Socket a la IP destí i el port adequat. En el nostre cas el PORT i Localhost com IP d'entrada faran la feina:

Socket socketClient = new Socket ("localhost", PORT);

Un cop generada la connexió entre socketClient i socket del Servidor, ja es pot procedir a realitzar la comunicació utilitzant els canals adients d'entrada i sortida, que la classe Socket embolcalla en els mètodes getInputStream i setOutputStream, que tractarem posteriorment en aquest mateix apartat.

2.4.3 Tipus de Sockets

Els Sockets a Java es diferencien en:

- **Socket:** Es l'objecte bàsic de tota comunicació a través de xarxa, utilitzant el protocol de comunicacions TCP. Aquesta classe implementa mètodes que fan molt senzilles les operacions d'entrada i sortida, utilitzant streams que realitzen la lectura i escriptura al canal de comunicació.

- **ServerSocket**: Es l'objecte passiu utilitzat únicament a la part servidor, i que escolta peticions d'entrada per part dels clients de l'aplicació, proporcionant el punt d'entrada a la comunicació. Aquest objecte no proporciona funcionalitat al client, si no que genera el Socket que es comunicarà amb aquest, fent exclusivament una tasca de mediador de les entrades a l'aplicació.

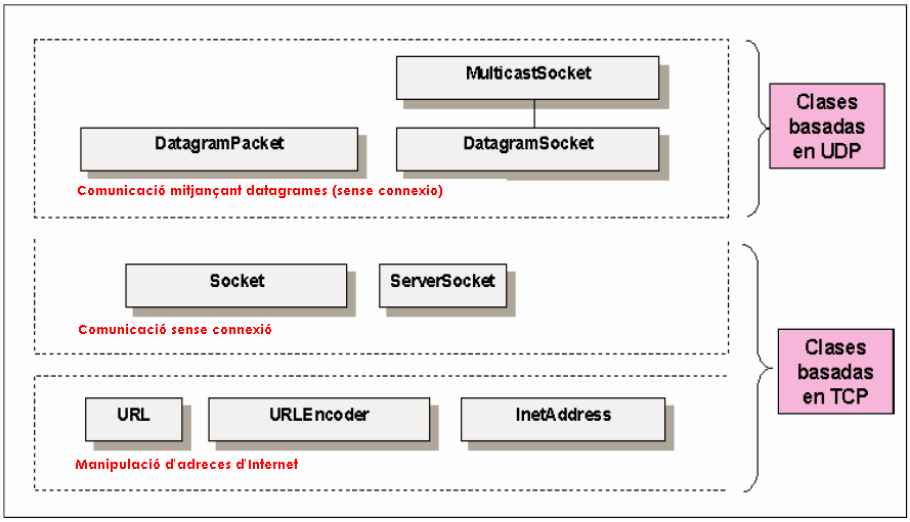
- **DatagramSocket**: És la classe utilitzada per tal de generar Sockets no fiables ni endreçats, és a dir, de Sockets que treballen sota el protocol de comunicacions UDP. L'abantatge real d'aquest tipus de Sockets és la velocitat, ja que no cal establir una comunicació entre el client i el servidor, evitant tots els protocols de connexió necessaris per tal efecte.

- **MulticastSocket**: Classe utilitzada per tal de generar una versió multicast dels DatagramSocket. Es tracta de múltiples clients/servidor que poden transmetre informació aleatòriament a un grup multicast (es tracta d'un grup d'adreces IP que comparteixen el PORT de la comunicació).

- **NetworkServer**: Es tracta d'una classe creada per tal d'implementar mètodes i variables utilitzades a la creació d'un servidor TCP/IP.

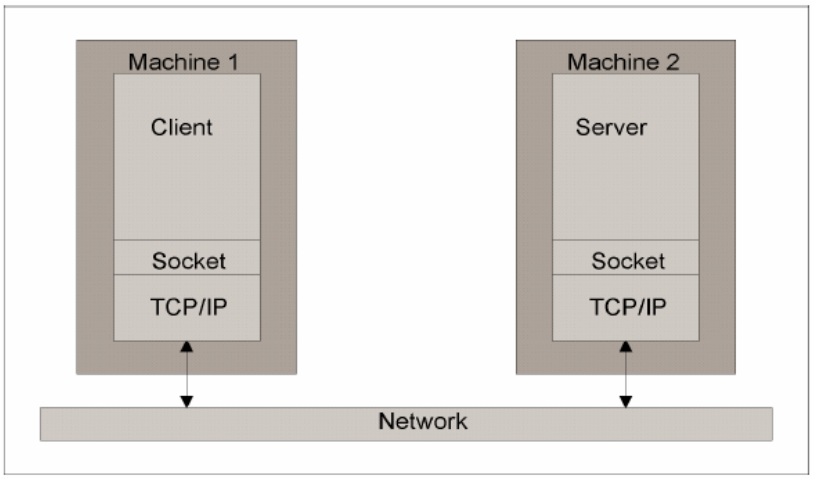
- **NetworkClient**: Es tracta d'una classe creada per tal d'implementar mètodes i variables utilitzades a la creació d'un client TCP/IP.

- **SocketImpl:** Es una interfície que ens permet crear-nos el nostre propi model de comunicacions. Els seus mètodes han de ser implementats a mesura que els necessitem per tal d'assolir les nostres pròpies necessitats, i les funcionalitats que desitgem donar al nostre mètode de connexions.
  En cas que necessitem desenvolupar una aplicació amb uns requeriments especials i propis, com pot ser per exemple la implementació d'un *Firewall* o accedir a equips especials (un lector de codis de barres), necessitarem evidentment la nostra pròpia classe socket.



2.4.4 Pas d'informació via Socket

En aquest punt tractarem el pas d'informació dins del canal de comunicacions generat per Socket.



Tenim connectat el Servidor amb el Client, però cal saber quins son els mètodes a utilitzar per aconseguir la comunicació entre ambdós.

Veurem la manera més estàndard de comunicacions, així com la problemàtica en que ens em trobat per haver utilitzat aquest tipus de comunicació estàndard i no haver plantejat d'entrada una solució més, diguem, elaborada.
Cal tenir amb compte que com tot a aquesta vida, la solució senzilla i ràpida no és mai la millor.

Així doncs les comunicacions amb Sockets es realitzen utilitzant els mètodes que ofereix la classe getInputStream i getOutputStream, que connecten amb els canals de comunicació, i realitzen el pas d'informació a baix nivell tot utilitzant els protocols de comunicació TCP, que enviaran d'una a altre part del canal de comunicació allò que hi enviem.

Utilitzem *getInputStream* per obtenir des d'un extrem de la connexió, el que ens és enviat des de l'altre extrem.

Utilitzem *getOutputStream* per tal d'enviar a l'altre extrem de la connexió, una informació des de la nostre posició.

Per tal de poder tractar aquesta informació hem de realitzar un embolcall sobre els mètodes de la classe Socket, per tal de tractar els canals com si d'un sistema d'E/S normal es tractés. Treballarem amb els canals com si estiguéssim llegint i escrivint a un fitxer de text normalment, gràcies a aquest embolcall sobre els tipus originals que retorna.

Així doncs, getInputSteram retorna un InputStream i getOutputStream retorna un OutputStream.

Com hem comentat al inici d'aquest punt, podem tenir la necessitat d'enviar diferents tipus de dades, tant cadenes de text per comunicacions senzilles com tipus natius de Java, per comunicacions més complexes.

Era d'esperar que en un chat que utilitza protocols de seguretat, seria necessari d'enviar altres tipus de dades que no fossin estrictament cadenes de text, però en un principi i per error es va decidir de anar pel camí senzill, i això ha passat factura.

Les comunicacions amb embolcall amb cadenes de caràcter es pot fer amb **BufferedReader** per llegir i amb **PrintWriter** per escriure.

Així doncs,

```
BufferedReader lector = new BufferedReader (new InputStreamReader (socket.getInputStream()));
PrintWriter escriptor = new PrintWriter (socket.getOutputStream ());
```

generen dos objectes lector i escriptor que poden ser utilitzats de manera molt senzilla.

BufferedReader ofereix mètodes de lectura a nivell de caràcter, realment molt senzills, que permeten accedir a les dades.
Read () retorna un char i *readLine* que retorna una línea sencera, son els dos mètodes més senzills i utilitzats, sobretot el *readLine()*.

D'altra banda, PrintWriter permet escriure al canal de sortida amb un senzillíssim println (). L'embolcall de PrintWriter permet que el println() enviï la informació al canal de sortida, que és el getOutpuStream del Socket, i aquest envia la informació al Socket destí.

Com es pot observar, és una manera força fàcil d'utilitzar els canals de comunicació, però te una restricció evident **¿i si volem enviar bytes?**

Tot treballant amb l'aplicació de seguretat pel projecte que ens comporta, arriba el fatídic moment en el que volem enviar les claus generades per Diffie-Hellman, així com la PublicKey que enviem al destinatari per tal que calculi la seva SecretKey en funció d'aquesta (com veurem en successius apartats). En aquest moment ens apareix la necessitat de passar la PublicKey en forma binaria.
Tenim aquesta necessitat doncs la classe PublicKey ens dona un mètode per tal de recuperar els valors en binari: getEncoded (). Però es realitza la codificació a binari utilitzant el mètode de codificació propi de la classe, que és x509, i que no podem utilitzar per recuperar d'String a binari, en cas que ho passem per PrintWriter (que ens convertiria el binari a un String).

Per aquest motiu necessitem passar les dades en forma nativa i no en cadenes de caràcter. Necessitem els **DataInputStream** i **DataOutputStream**.

Aquestes dues classes funcionen de la mateixa manera que les anteriors BufferedReader i PrintWriter, amb la diferència que permeten passar tot tipus de dades natives de Java. Contenen mètodes per passar caràcters, Strings, Longs,... i evidentment, bytes.

Amb els mètodes read (byte[] b) o read (byte[] b, int off, int len) es llegeixen els bytes enviats par socket, des del *DataInputStream.*

Amb el mètode write (byte[] b, int off, int len), escrivim els bytes al canal de comunicació, amb el *DataOutputStream.*

La seva implementació seria:

```
DataInputStream entrada = new DataInputStream (socket.getInputStream());
DataOutputStream sortida = new DataOutputStream (socket.getOutputStream());
```

Cal tenir amb compte que aquestes classes ofereixen un seguit de funcionalitats més, que no son objecte d'aquest projecte d'estudiar.

2.4.5 java.net

El paquet java.net proporciona una inteface OO per crear Sockets, connexions HTTP, localitzadors URL, ... i compren classes que es poden dividir en dos grans grups:

a) classes corresponents a les API dels Sockets: Socket, ServerSocket, DatagramSocket...
b) classes corresponents a les eines per treballar amb URL: URLConnection, URLEncoder...

Pel que ens ocupa, ens interessa aprofundir a les classes Socket i ServerSocket.

ServerSocket es un Socket passiu que s'executa al Servidor, i que resta a l'escolta de crides al port que se li assigna.
Socket es un Socket actiu (que es pot executar tant a client com a servidor) que intercanvia missatges amb altres programes mitjançant el Port associat per ServerSocket.

El funcionament és el següent:

Un procés crida mitjançant el constructor Socket (HOST, PORT) a un Port del Host, si aquest Host te un ServerSocket escoltant per aquest Port, el rep amb el mètode accept (), i crea un objecte Socket al servidor que es posa en contacte amb el Socket del client.
Els ports associats els genera automàticament la VMJ, en cas que n'hi hagi de disponibles (si no llença una excepció).
En aquest moment, amb els mètodes getInputStream i getOutputStream es poden enviar i rebre paquets.

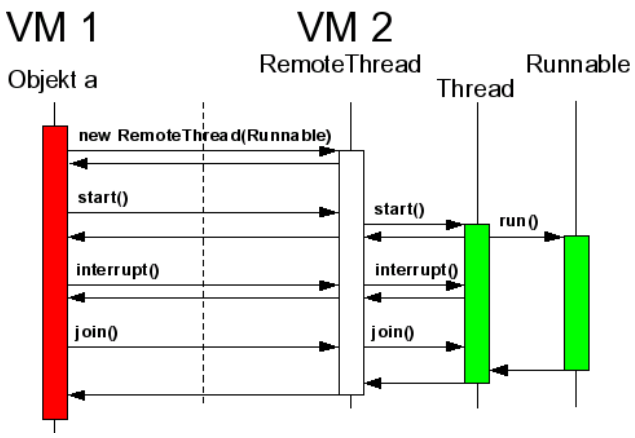
2.5 Threads

2.5.1 Introducció

En el moment de treballar amb Sockets, per connectar diferents clients alhora per tal que es comuniquin entre ells, ja sabem, o hauríem de saber, que treballarem amb una aplicació *multi-thread.*

Una aplicació *multi-thread* és aquella que s'executa en diferents plans d'execució. És aquella que en cada moment pot tenir un o més fils d'execució que actuen de manera concurrent.

Els Threads son diferents processos independents executant-se de manera concurrent.



Moltes aplicacions precisen d'execucions concurrents per tal d'assolir els seus objectius. Sense anar més lluny, els propis Sistemes Operatius estan farcits de fils d'execució paral·lels, per tal de donar resposta fiable i ràpida a les peticions dels usuaris, així com als diferents processos que s'executen a cada moment.

Un Sistema Operatiu pot estar donant servei de xarxa, executant un full de càlcul i reproduint música digital en el mateix moment, i això combinat amb els seus propis processos interns. Tot això s'aconsegueix gràcies als diferents fils d'execució que aquest Sistema Operatiu és capaç d'executar de manera concurrent.

Com és obvi i immediat, a més fils d'execució concurrents, menys memòria resta disponible per a altres aplicacions, doncs cada fil d'execució utilitza memòria pròpia per tal de realitzar les seves tasques.

Com defineixen **Elisa Viso Gurovich i Francisco Solsona Cruz** a la WEB http://www.mcc.unam.mx/~cursos/Algoritmos/javaDC99-1/Java_threads/resumen.html:

"¿Qué es un thread (hilo de control o simplemente hilo)? Un hilo -algunas veces llamado contexto de ejecución o proceso ligero- es un flujo de control secuencial dentro de un programa. Un único hilo es similar a un programa secuencial; es decir, tiene un comienzo, una secuencia y un final, además en cualquier momento durante la ejecución existe un sólo punto de ejecución. Sin embargo, un hilo no es un programa; no puede correr por sí mismo, corre dentro de un programa. Un hilo por si mismo no nos ofrece nada nuevo. Es la habilidad de ejecutar varios hilos dentro de un programa lo que ofrece algo nuevo y útil; ya que cada uno de estos hilos puede ejecutar tareas distintas."

2.5.2 Necessitats dels Threads

En el cas de l'aplicació de Chat Segur, és evident la necessitat de concurrència de processos, tant a la part del Servidor com a la part dels successius clients que poden aparèixer.

Cal tenir amb compte que el programa constarà d'un servidor que ha de permetre donar entrada a noves peticions i tractar les actives, de manera concurrent, és a dir, que ha d'estar interactuant amb un o més clients alhora, sense deixar-ne un penjat fins que finalitzi el seu tractament d'un altre, si no tractant a tots els usuaris per igual en cada moment donat.
A més, ha d'estar donant entrada als clients que es connecten a cada moment, sense deixar de mantenir les connexions actives.

Per la part del client també és força clar que un usuari ha de poder veure actualitzada la llista d'usuaris connectats a l'aplicació, així com mantenir la o les converses que pugui tenir actives en cada moment donat, i per tant també necessitarà concurrència de processos.

Està clar que hi haurà una complexa jerarquia de fils de procés a cada computador que executi el programa, sigui Client o Servidor, que canviarà en funció del nombre d'usuaris connectats, nombre de converses actives...

Aquesta jerarquia de Threads, que es defineix per el nostre cas concret en punts posteriors, està intrínsecament lligada al treball en sockets, doncs per cada Socket que s'activi a cada moment donat, tindrem un nou Thread a cada banda del Socket.
Per tant, podem afirmar que en el nostre cas, per cada parell de Sockets tindrem un nou parell de Threads corrent paral·lelament.

En el nostre cas, un Servidor escolta peticions a ServerSocket, i cada petició d'entrada que es produeix, genera un socket per la petició, que assigna a un ThreadServidor.

De mode similar, cada client que intenta una connexió al servidor, quan aconsegueix generar-la i crear un socket, es crea alhora un ThreadClient.

Per tant, i com s'ha resumit en aquest punt, la aplicació Chat Segur basarà gran part del seu potencial a l'ús i gestió dels Threads.

2.5.3 Funcionament dels Threads a Java

Dins aquest apartat definirem a brevetat el funcionament dels Threads.

Per començar, a Java els Thread son classes. Com tot a Java, els Thread son una Classe que hereta d'Object, i que cal ser "Extesa" per tal de ser utilitzada.

Una classe generada per nosaltres, per tal que sigui multi-fil, cal declarar-se com segueix:

public class ThreadServidor extends Thread

Amb això estem generant una classe ThreadServidor que és una especialització de la classe Thread, que és la que proporciona les funcionalitats de multi-fil a Java.

Un Thread, el que fa realment, és duplicar la línia d'execució.

El programa seqüencial arriba a un punt en el que crida a una altre classe, i aquesta nova classe, en el seu constructor, activa el multi-fil, de manera que pel programa "seqüencial", detecta que la funcionalitat de la classe cridada ha acabat i continua executant el seu codi normalment. Però en realitat, la classe cridada ha començat a executar-se a un àmbit paral·lel, executant les seves pròpies instruccions i actuant com un programa seqüencial diferent del primer.

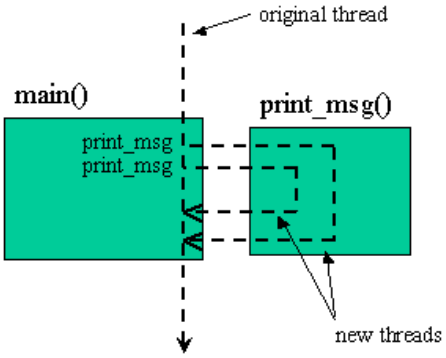
El mètode que realitza aquesta diferenciació, és el mètode run ().

Tot allò que s'escriu dintre del mètode run() d'una classe que extends Thread, s'executarà a un fil paral·lel al que s'estava executant prèviament.

La crida a run() s'ha de fer amb el mètode start().

```
public class Servidor (){
...
(cos de programa)
ThreadServidor thS= new ThreadServidor().start();
(continua l'execució)
...
}

public class ThreadServidor extends Thread (){
...
(cos de programa)
public void run(){
        (comença el fil paral·lel)
}
...
}
```



A partir d'aquest punt, totes dues classes s'estan executant paral·lelament, de manera que la segona resta pendent de la primera, és a dir, si la classe ThreadServidor finalitzés la seva execució, voluntàriament o involuntària, la classe Servidor continua la seva execució sense problemes. Ara bé, si la classe que finalitza sobtadament fora Servidor, ThreadServidor seria automàticament aturada.

2.5.4 Compartició de dades entre Threads

En casos podem tenir la necessitat d'accedir a la mateixa variable des de totes les instanciacions d'una classe, i per tant fem que aquesta variable sigui **static**.

Una variable **static** és aquella que manté el seu valor per a totes les instanciacions d'una classe, de manera que es fa comuna a totes elles.

De la mateixa manera, tot mètode que s'utilitzi per tal de llegir-la ha de ser static i tot mètode que s'utilitzi per modificar-la ha de ser **synchronized**.

Un mètode **synchronized** és aquell mètode que modifica una variable static, i es sincronitza per tal que dos intanciacions de la classe no puguin accedir-hi al mateix temps.

Això es fa per un motiu ben senzill, tenint en compte que es modifica el valor comú, si dos instancies intenten modificar-ne el valor alhora, una d'elles perdrà la modificació. De manera que si es sincronitza el mètode, funciona a mode de semàfor i realitza les modificacions de manera seqüencial, no permetent una instancia d'accedir al mètode fins que l'altre l'ha alliberat.

D'aquesta manera funciona el fil del Servidor de l'aplicació Chat Segur. Es creen unes variable static i uns mètodes synchronized per tal de mantenir la llista de Threads actius, usuaris d'una conversa... de manera que es puguin relacionar clients fàcilment, accedint a aquestes llistes estàtiques.

2.6 Diffie-Hellman

El protocol Diffie-Hellman permet l'intercanvi de claus entre dos clients que no han entrat mai en contacte, per un canal insegur i d'una manera anònima.

Aquest protocol fou presentat per Whitfield Diffie i Martin Hellman al "New directions in cyptography", *IEEE Transactions on Information Theory* 22 (1976).

Es fonamenta sobre el problema del **logaritme discret**, que és un problema del que es creu, fins avui dia, que és irresoluble computacionalment.

El problema del logaritme discret consisteix a solucionar l'equació següent:

$x = a^y \bmod n$

on *x*, *a* i *n* son constants i *y* és la incògnita que s'intenta cercar.

$y = log\ disc_a(x)$

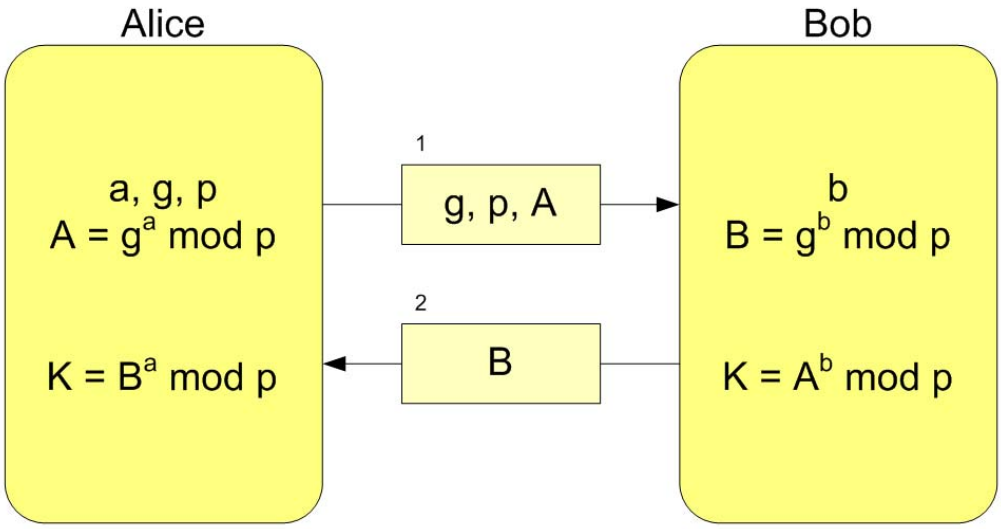
La utilització de l'aritmètica modular introdueix una complexitat enorme al problema, fent que el logaritme discret sigui utilitzat a cartografia, utilitzant-se com a base d'algoritmes com el DSA, o el propi Diffie-Hellman.

En més detall:

Donat x, 0 < x < p, existeix un únic y tal que 0<= y <= p-2, tal que:
$X = a^y \pmod p$ y a és una arrel primitiva a mod p.

Aquest protocol és utilitzar les claus simètriques que seran utilitzades per codificar/descodificar els missatges que seran enviats entre dos usuaris, a una comunicació entre ambdós.

Així doncs, dos usuaris aconsegueixen mitjançant aquest protocol d'establir una parella de claus secretes, amb les quals podran realitzar el xifratge de la informació de manera complertament segura, doncs aquest protocol basa la seva seguretat a l'extrema dificultat (conjeturada però no demostrada) de calcular logaritmes discrets a un espai finit.



$K = A^b \bmod p = (g^a \bmod p)^b \bmod p = g^{ab} \bmod p = (g^b \bmod p)^a \bmod p = B^a \bmod p$

Prenem com exemple, dues parts A(lice) i B(ob) intenten establir una clau secreta i un adversari E escolta la comunicació amb la intenció d'assolir la informació.

S'estableixen un nombre primer ***p*** i un generador ***g*** que pertany a Z_p. Aquests nombres son públics, coneguts per A, B i per que no, E.

A escull a l'atzar un nombre ***x*** que pertany a Z_{p-1}, calcula $\mathbf{X} = g^x \pmod p$, i envia aquesta X a B.
B escull a l'atzar un nombre ***y*** que pertany a Z_{p-1}, calcula $\mathbf{Y} = g^y \pmod p$, i envia aquesta T a A.

Un atacant E que tingui *p, g, X* i *Y,* podria calcular el secret compartit si tingués un dels valors privats (x o y), o aconseguís invertir la funció. Però calcular x donat X és el problema de l'**algoritme discret**.

En canvi, el protocol es sensible a l'atac d'"home al mig", que és quan un atacant es posa enmig de la comunicació, fent-se passar per l'interlocutor de cara a cada participant de la comunicació, establint les claus amb cadascun d'ells, xifrant i desxifrant la informació, i poden accedir així a la informació.

Aquest atacant es pot fer passar per l'altre part de la comunicació tant de cara al destinatari com a l'emissor, i d'aquesta manera cada part de la comunicació pensa estar establint les claus amb l'altre, quan en realitat ho fan amb l'atacant. Això és així doncs aquest protocol no proporciona cap mitjà per validar la identitat dels participants a la comunicació.

2.6.1 Diffie-Hellman a Java

A Java tenim tota una sèrie de classes per tal de realitzar les tasques de criptografia necessàries, i per tant les tenim també per generar claus utilitzant el protocol Diffie-Hellman.

Entre aquestes classes tenim els generadors de paràmetres, que generen els valors públics, els generadors de claus (públiques i privades i la clau secreta) i les classes necessàries per tal de recuperar els valors que l'altre usuari de la comunicació ens envia en binari.

Així doncs els diferents tipus de classes, que ens acabaran proporcionant la clau necessària pel xifratge, son:

- Generació de paràmetres

  AlgorithmParameterGenerator:
  Aquesta classe és una classe motor que genera una sèrie de paràmetres utilitzats en un cert algoritme, que és indicat a la classe mitjançant el mètode getInstance(String protocol_a_utilitzar).
  Posteriorment, amb el mètode generateParameters (), es retorna un objecte AlgoritmParameter amb els paràmetres generats.

  AlgorithmParameter:
  Aquesta classe s'utilitza com una representació opaca dels paràmetres criptogràfics generats amb el motor anterior.
  Per tal de fer accessibles aquests paràmetres, haurem de cridar al mètode getParameterSpec, que genera un Spec del tipus AlgorithmParameterSpec, el qual el castejarem a un DHParameterSpec, per tal de representar els valors en format Diffie-Hellman.
  Això ho farem així:
  DHParameterSpec dhSpec = (DHParameterSpec)params.getParameterSpec(DHParameterSpec.class);

  DHParameterSpec:
  Aquesta classe defineix el grup de paràmetres que es necessiten per treballar amb el protocol Diffie-Hellman.
  Els paràmetres que recupera son el primer p (getP()), la base generadora g (getG()) i la longitud en bits del valor privat l (getL()).

- Generació de claus

Amb els valors recuperats per les classes anteriors, generarem la parella de claus, pública y privada.

KeyPairGenerator
Aquesta classe és l'encarregada de generar la parella de claus pública/privada, segons l'especificació que se li passi.
En principi, li indiquem amb el mètode getInstance ("DH"), que generarà una parella de claus de tipus Diffie-Hellman, i posteriorment, amb una nova instanciació de DHParameterSpec, amb els valors P, G i L recuperats anteriorment, li indiquem a la classe generadora quins son els paràmetres que ha d'utilitzar per tal de realitzar el càlcul, amb el mètode:

keyGen.initialize(dhSpec);

on keyGen és la classe i dhSPec la classe DHParameterSpec creada.

Finalment prenem els valors públic i privat amb:

privateKey = keypair.getPrivate();
publicKey = keypair.getPublic();

- Recreació de claus des de binari

  Per tal de posar-se d'acord les dues parts en els valors del protocol DH, s'ha de passar la publicKey d'un usuari a l'altre, i aquest pas es fa en binari.
  Per tal de passar la clau s'utilitza el mètode getEncoded () de la classe PublcKey, la qual ens retorna el valor de la publicKey en binari.
  Aquesta codificació es fa utilitzant el format estàndard de codificació X509.

  Per tant, per tal de recuperar-lo per part de qui el rep, cal utilitzar la següent classe:

  X509EncodedKeySpec:
  Aquesta classe s'utilitza per tal de tornar a codificar el binari rebut al format PublicKey original.

  Amb aquesta instrucció generem un objecte de la classe, amb els bytes de la publicKey original.
  X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(publicKeyBytes);

  Per tal de crear la classe original necessitem una classe regenereadora (factory) com és **KeyFactory** que amb la classe X509EncodedKeySpec recrearan la PublicKey de la següent manera:

  KeyFactory keyFact = KeyFactory.getInstance("DH");
  l_publicKey = keyFact.generatePublic(x509KeySpec);

- Generació de clau secreta

  Per generar la clau secreta, hem d'utilitzar la classe **KeyAgreement**.

  KeyAgreement

  Es genera una classe d'aquest tipus, amb el mètode getInstance() que indica que utilitzem el protocol DH.

Posteriorment indiquem amb el mètode init() quin és el nostre privateKey associat, i posteriorment amb el doPhase, els diferents publicKeys dels diferents usuaris amb els que establim la connexió.
En cas que només sigui un es farà:
Ka.doPhase (public, true);

En cas que siguin més d'un (diguem un parell):
Ka.doPhase (public1, false);
Ka.doPhase (public2, true);

Finalment amb el mètode generateSecret (String nom_algoritme_encriptacio), recuperem la clau secreta que utilitzarem pel xifratge. En el nostre cas l'algoritme serà Triple Des.

2.7 Criptografia

La criptografia és la ciència de xifrar i desxifrar la informació utilitzant tècniques matemàtiques que fagin possible l'intercanvi d'informació de manera que només sigui accessible per la persona o les persones a qui va dirigida.

La finalitat de la criptografia és garantir la seguretat de la informació enviada, de manera que l'emissor i el receptor pugin estar segurs que la informació rebuda és de qui ha de ser, i que no ha estat modificada maliciosament.

La informació original, anomenada **text en clar**, ha de ser transformada a un text sense cap mena de sentit, anomenat **text xifrat o criptograma** i sense que sigui possible la seva deducció per força bruta.

Hi ha mètodes de xifratge de dades prou senzills per que un atacant maliciós sigui capaç de desxifrar-lo utilitzant tècniques de força bruta, que no son sinó anar provant fins que no es troba un resultat coherent.

Aquestes tècniques son funcionals per a xifratges senzills, com reposicionar la lletra utilitzada cada moment per un pos(n)+x posicions a partir d'aquesta lletra.
Si la lletra és "a" i la reposició 3, en comptes de tenir una "a" tindrem una "d".
Aquest exemple senzill i aparentment absurd, es semblant a les primeres tècniques utilitzades pels romans a l'època de Juli Cèsar.

Generalment, l'aplicació concreta d'una algoritme de xifrat es basa a l'existència d'una clau, la qual és secreta i adapta cada algoritme per l'ús propi.

El xifratge és el procés d'obtenció del text xifrat a partir del text en clar, utilitzant la clau de conversió.

Les dues tècniques més bàsiques del xifratge a la criptografia clàssica son:

- la substitució
  que consisteix a substituir el significat dels elements bàsics del llenguatge, ja siguin lletres, símbols o dígits.

- la transposició
  consisteix a realitzar una reposició d'aquests elements bàsics.

Tot i que la gran majoria de les tècniques clàssiques de xifratge es basaven a la unió d'ambdues tècniques.

El desxifratge per la seva part és la tècnica de recuperar, mitjançant la clau, el text pla a partir del text xifrat.

El **Protocol Criptogràfic** determina la manera d'utilitzar els algoritmes i les claus per tal de realitzar aquest procés de xifratge.

El conjunt de protocols, algoritmes de xifrat, processos de gestió de claus i actuacions dels usuaris, son el que globalment constitueixen un criptosistema, que és com l'usuari final treballa.

Algoritmes

Existeixen dos grans grups d'algoritmes:

- Aquells que utilitzen una única clau tant en el procés de xifratge com en el de desxifratge – **xifratge de clau simètrica**
- Els que utilitzen una clau per xifrar la informació i una clau diferent per tal de desxifrar-los – **xifratge de clau asimètrica** (que son els més utilitzats actualment).

Com a algoritmes de xifratge tenim AES, RSA, DES/TripleDES, TEA/XTEA, ARC4, DSA, ECDSA, MD5, ROT-13, Enigma i Base64.

Sistemes de xifrat simètric

El sistemes de xifratge simètric son aquells que utilitzen la mateixa clau per tal de xifrar la informació com per desxifrar-la.
El principal problema de seguretat resideix a el intercanvi de claus entre emissor i receptor, per la qual cosa cal cercar canals de comunicació segurs, o be utilitzar mètodes de seguretat com el cas de Diffie-Hellman.

És molt importat que la clau sigui molt difícil d'esbrinar, ja que el potencial enorme de les estacions computacionals d'avui dia tenen la capacitat de realitzar grans càlculs en temps molt reduïts.

Per exemple, un algoritme de xifrat DES utilitza una clau de 56bits i per tant hi ha 72 mil bilions de claus possibles.
Actualment existeixen computadors especialitzats capaços de comprovar totes aquestes claus en qüestió d'hores, i per tant ja s'estan utilitzant claus de 128 bits, que augmenten les possibles claus fins a 2 elevat a 128, de manera que el seu càlcul encara és inabastable per un computador de "força bruta".

Sistemes de xifrat asimètric

També anomenats sistemes de xifrat de clau pública.
Aquest sistema utilitza dues claus, una pública que pot ser enviada a qualsevol persona i una altra que s'anomena clau privada, que ha de ser inaccessible per tothom.

Per tal d'enviar la informació l'emissor utilitza la clau pública del receptor per codificar el missatge. Un cop codificat el missatge, només pot ser desxifrat per la clau privada del receptor.

Per aquest motiu es pot donar a conèixer la clau pública a tothom que es vulgui comunicar amb el destinatari.

Un sistema de xifrat de clau pública basat a la factorització de nombres primers es basa a que la clau pública te un nombre compost per dos nombres primers molt grans. Per xifrar un missatge, l'algoritme de xifrat utilitza aquest compost per xifrar el missatge.

Per desxifrar-lo, l'algoritme requereix conèixer els factors primers i la clau privada te un dels factors, amb la qual cosa pot fàcilment desxifrar el missatge.

Amb els computadors actuals és molt senzill multiplicar dos nombres grans per tal d'aconseguir-ne un compost, però és molt complicada l'operació inversa.

Es recomana que actualment una clau pública sigui de 1024 bits.

Sistemes de xifratge híbrids

És aquell sistema que utilitza tant els sistemes de clau simètrica com els sistemes de clau asimètrica pel procés de xifratge.

Funciona mitjançant un xifrat de clau pública per compartir una clau pel xifratge simètric.
A cada missatge, la clau simètrica es diferent per la qual cosa si una atacant aconseguís descobrir la clau simètrica, li serviria per aquell missatge i no per la resta.

Tant PGP com GnuPG utilitzen aquest sistema de xifratge.

La clau simètrica és xifrada amb la clau pública, i el missatge sortint és xifrat amb la clau simètrica, tot combinat automàticament a un sol paquet.

El destinatari utilitza la seva clau privada per tal de desxifrar la clau simètrica i després utilitza la clau simètrica per desxifrar el missatge.

2.7.1 Algoritme TripleDES

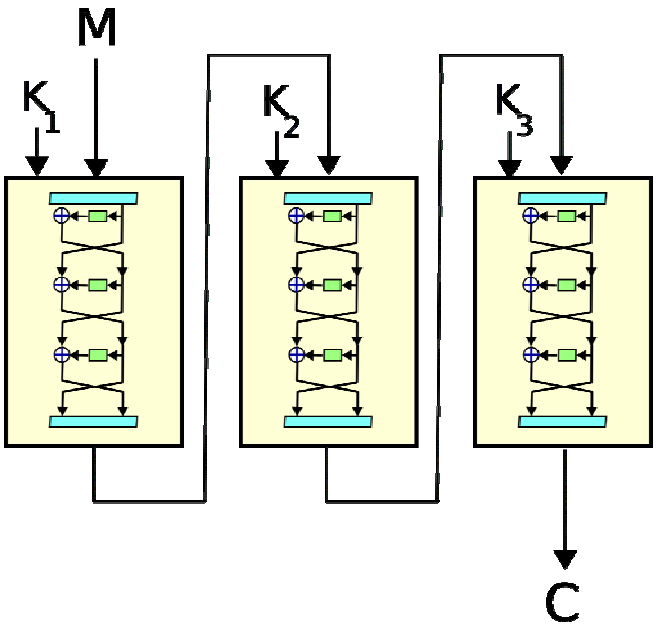
També conegut com el TDES, es basa senzillament a l'aplicació de l'algoritme DES tres cops seguits, per augmentar-ne la seguretat.
Fou desenvolupat per IBM al 1978.

No arriba a ser un xifratge múltiple doncs no son independents totes les subclasses. Aquest fet es basa a que DES te la característica matemàtica de no ser un grup, la qual cosa implica que si es realitza el xifratge al mateix bloc dos cops amb dues claus diferents s'augmenta la mida efectiva de la clau.

La variant més senzilla del Triple DES funciona de la següent manera:

$$C = E_{DES}^{k3}\left(D_{DES}^{k2}\left(E_{DES}^{k1}(M)\right)\right)$$

On M és el missatge a xifrar i k1, k2 i k3 les respectives claus de xifratge.



En quant es va descobrir que una clau de 56 bits no era suficientment resistent per aguantar un atac de "força bruta", TDES va ser escollit com forma de fer més gran la clau sense la necessitat de canviar d'algoritme de xifratge.
Aquest mètode de xifratge és resistent a un atac a "mig camí", doblant la longitud efectiva de la clau, però en canvi és necessari triplicar el nombre d'operacions de xifratge, fent aquest mètode incomparablement més segur que el DES.

El Triple DES està desapareixent lentament, doncs està sent reemplaçat per l'algoritme AES.
Tot i així, la majoria de targetes de crèdit i altres mitjans de pagament electrònic tenen com estàndard l'algoritme Triple DES.
Pel seu disseny DES i Triple DES son lents. AES pot arribar a ser fins a 6 vegades més ràpid i fins ara no s'ha trobat cap vulnerabilitat a aquest algoritme de xifratge.

2.7.2 Criptografia a Java

Java proporciona les seves pròpies classes per tal d'utilitzar els mètodes i els algoritmes criptogràfics existents.

El paquet javax.crypto.* proporciona un seguit de classes que faciliten enormement la tasca de xifrar i desxifrar les dades enviades per xarxa d'un usuari a un altre.

En el nostre cas em utilitzat únicament les classes Cipher i SecretKey.

Cipher

Aquesta classe proporciona les eines necessàries per tal de codificar i descodificar un missatge, utilitzant l'algoritme de codificació TripleDES, que és el que hem decidit d'utilitzar.

Per utilitzar-la, generem dues instàncies de la classe, una per la codificació i l'altre per la codificació, de la manera que segueix:

```
ecipher = Cipher.getInstance("TripleDES");
dcipher = Cipher.getInstance("TripleDES");
ecipher.init(Cipher.ENCRYPT_MODE, key);
dcipher.init(Cipher.DECRYPT_MODE, key);
```

Primer instanciem les dues classes tot indicant quin serà el nostre algoritme de xifratge, i posteriorment indiquem a cada classe si es tracta d'un codificador (*ENCRYPT_MODE*) o un descodificador (*DECRYPT_MODE*), i afegim la clau *key* que és el SecretKey utilitzat per la codificació.

Per tal de realitzar el **xifratge**, primer passem l'String a bytes utilitzant el charset d'UTF8 :

```
byte[] utf8 = str.getBytes("UTF8");
```

i posteriorment encriptem en binari utilitzant els bytes obtinguts anteriorment.

```
byte[] enc = ecipher.doFinal(utf8);
```

Es retorna el valor en format Base64 per tal d'enviar-lo:

```
return new sun.misc.BASE64Encoder().encode(enc);
```

Per a realitzar el **desxifratge**, cal realitzar el procés invers utilitzant la funció doFinal() de la instancia descodificadora, tot i que prèviament cal recodificar a bytes el String en Base64 rebut:

```
byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(str);

byte[] utf8 = dcipher.doFinal(dec);
```

Finalment passem els bytes a String en format UTF8, que és el que havíem utilitzat per codificar a binari en la funció de codificació:

```
new String(utf8, "UTF8");
```

2.8 JDOM i XML

Per tal de dotar d'agilitat a l'aplicació, es decideix guardar a una agenda els paràmetres de connexió que s'utilitzen per cada client amb que ens posem en contacte, agilitzant així el càlcul de les claus a utilitzar en cada moment pel procés de xifratge.

Per comoditat, funcionalitat i evident organització, es decideix de realitzar aquesta agenda en format XML, ja que no cal la potència d'un gestor de base de dades però sembla feble la funcionalitat d'un fitxer de text pla.

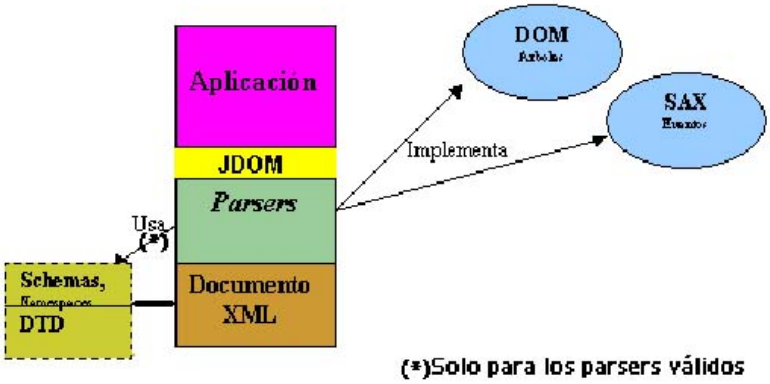
L'XML ens permet de realitzar la lectura/escriptura de dades com si estiguéssim treballant amb arbres, amb els evidents beneficis d'ordre i velocitat que això comporta.

Java ofereix les seves eines DOM i SAX, però aquestes es van idear sense pensar en cap llenguatge concret, la qual cosa les converteix en unes eines difícils d'utilitzar.

Per contra JDOM, que encara no ha estat inclosa a cap versió de Java fins el moment, ofereix unes eines molt senzilles d'utilitzar i permet treballar amb XML molt fàcilment.

Aquestes llibreries son OpenSource i estan disponibles tant per entorns Winx com per entorns Unix.

A www.jdom.org es pot trobar tota la informació necessària així com les llibreries per tal de treballar-hi.



En el nostre cas, amb el simple jdom.jar tenim prou per utilitzar les eines de la llibreria. Aquesta llibreria caldrà incloure-la al Path d'execució en el moment de la instal·lació o recompilar-la dins del nostre paquet.

La utilització és força senzilla. Només cal realitzar el Parser amb un objecte del tipus SAXBuilder, i realitzar el build (String path+nomfitxer);

La classe document del Jdom rebrà el fitxer "parsejat", i permetrà treballar amb ell amb total facilitat, permetent llegir l'arrel (getRootElement()), o els Elements fills de qualsevol Element (getChild(String Nom_del_node)), i llegir-ne el valor en cas de tractar-se d'una fulla (getValue()).

```
Iterator i = nodos.iterator();
Element contacte = (Element)i.next();
ls_Return = contacte.getChild("p").getValue();
```

De manera anàloga, la llibreria ens permet inserir nous nodes amb la informació necessària:

```
Element contacte = new Element(ls_Cont);
contacte.addContent (new Element("p").setText(ls_p));
contacte.addContent (new Element("p").setText(ls_g));
contacte.addContent (new Element("p").setText(ls_l));
arrel.addContent(contacte);
document.removeContent();
document.addContent(arrel);
```

Es crea un nou element que serà el element node, al qual es pengen nous elements que son fulles amb valor.
Posteriorment s'afegeix aquest nou node a l'arrel, i actualitzem el valor de tot l'arbre.

Amb això estem inserint un node amb els seus valors.

```
<arrel>
        <nodes>
        ...
        <node_nou>
                <fulla1>valor</fulla1>
                <fulla2>valor</fulla2>
                ...
                <fullan>valor</fullan>
        </node_nou>
</arrel>
```

2.9 Funcionalitat del client

El client del Xat ha de poder realitzar les funcions bàsiques d'un Xat.

Cada client ha de poder mantenir una conversa amb un o més clients a l'hora, de manera que el procés client ha de ser capaç de generar diversos Threats cada cop. Un per cada usuari amb qui es vulgui contactar.
Aquesta diversitat de processos ha d'estar ben gestionada i no s'ha de permetre que es perdin missatges.

Els valors públics rebuts d'altres clients en el moment de realitzar la connexió amb el procés Diffie-Hellman, seran emmagatzemats en un fitxer per tal de poder reutilitzar-los en el moment de posar-nos en contacte amb els demés usuaris.

3 – DISSENY FUNCIONAL

3.1 Introducció

L'objectiu de la aplicació és la creació d'un Xat de comunicacions entre usuaris, que permeti a dos usuaris mantenir una conversa privada entre ells.

Aquesta privacitat es manifestarà implícitament als protocols de seguretat que utilitzarem per tal d'enviar la informació xifrada, pels canals de comunicació.

La comunicació es realitzarà mitjançant Sockets que permeten una gestió fàcil i ràpida dels ports de comunicació dels computadors actuals.

El xifratge es realitzarà amb algoritme Triple DES de 128 bits amb clau pública compartida, la qual serà decidida entre emissor i receptor en el moment d'iniciar la conversa, mitjançant el protocol d'intercanvi de claus Diffie-Hellman, el qual es basa en clau pública compartida i clau secreta basada en una clau privada, amb la qual es xifren i desxifren les dades. Per tant és un algoritme de xifratge simètric, doncs utilitzem la mateixa clau per ambdós funcions: xifratge i desxifratge.

Un usuari podrà tenir diverses converses actives alhora, i si més no ha estat contemplat el fet de fer converses multitudinàries, seria fàcilment adaptable el codi del programa per tal de fer-ho. Això serà especificat durant aquest apartat de disseny funcional.

3.2 Funcionament de les comunicacions

El programa ha de ser presentat des de dos punts de vista diferents; el punt de vista del **Servidor** i el punt de vista dels successius **Clients** que utilitzaran els protocols que ofereix el Servidor.

SERVIDOR

El servidor ha d'oferir als clients el punt d'entrada al Servei de Xat, per tal que aquests puguin executar-se amb un punt comú que els faci de comunicador.

El Servidor restarà a l'escolta de les possibles peticions dels clients que desitgin utilitzar el servei, i mantindrà un *listener* o escoltador al PORT convingut de manera que accepti totes les peticions que puguin entrar-hi.

El Servidor s'encarrega de rebre les peticions de servei, generar un fil d'execució propi per aquest nou usuari, tot assignant-li un port de comunicacions propi, de manera que l'usuari el pugui utilitzar per la seva comunicació amb el Servidor.
El Servidor ha de demanar el nom d'usuari al Client, i en cas de ser un nom existent, i per no dificultar el programa fora dels paràmetres necessaris del que s'ha demanat, s'afegirà el nombre del port assignat a aquesta comunicació al nick inicial; nick+port (cal tenir amb compte que el nombre del port és únic per a cada comunicació).

El Servidor anirà enviant constantment un llistat dels usuaris connectats en cada moment, per tal que l'usuari pugui veure amb qui pot establir una connexió en un moment donat.

En el moment que un Client demani iniciar una conversa, el Servidor obrirà un nou fil d'execució per l'entitat Conversa, que mantindrà l'usuari que inicia la conversa, i l'usuari amb qui la vol tenir.

Cal deixar obert aquest nou fil per tal d'estar preparat per si calgués que una conversa fos participada per més de dos clients.

El Servidor dona per tant la funcionalitat de comunicació entre usuaris, sabent en tot moment quin usuari està connectat a quin, a qui cal enviar els missatges rebuts i de quina manera tractar les peticions que pugui fer l'usuari.

A més s'encarrega d'enrutar les peticions de connexió i decisió de claus per part del protocol Diffie-Hellman.

Com a funcionalitat aliena a la petició de l'enunciat, s'ha dotat al servidor d'un sistema senzill de logs, que penso que es imprescindible per tota aplicació que dona serveis a peticions procedents de connexions per ports des de l'exterior.

CLIENT

La part del Client s'arrenca cridant a la comunicació del **servidor:port** ja decidit, i en cas d'aconseguir comunicació, mostra una interfície gràfica que demana el nick a l'usuari, i un cop rebut aquest nick el connecta a la xarxa de comunicacions mostrant constantment la llista d'usuaris connectats a una llista al seu efecte.

L'usuari arrencarà una conversa amb un usuari concret realitzant un doble-click al damunt del nom que apareix a la llista d'usuaris connectats.

Un cop l'usuari realitza aquesta acció, es desenvolupa una successió de processos diferents si som nosaltres els que iniciem la conversa o si som els que la rebem.

El Servidor ens assignarà un nou fil amb un nou port de comunicacions, permetent que el usuari esculli una nova conversa sense haver de finalitzar la inicail.
Realitzarà una petició de connexió a l'altre usuari, que en rebre-la obrirà un fil d'execució idèntic al nostre i connectat amb nosaltres amb el seu propi fil d'execució.

En aquest moment s'inicia un intercanvi de missatges per tal de decidir les claus a utilitzar.
En cas que ja haguem iniciat una conversa amb aquest usuari en el passat, agafarem d'un fitxer XML a l'efecte, els paràmetres de les claus i les enviarem, evitant haver-les de tornar a calcular.

Un cop les dos instancies de Client_Conversa ja s'han posat d'acord en les claus a utilitzar, s'inicia el intercanvi d'informació, xifrada/desxifrada per un algoritme TripleDES.

Resum de Comunicacions

Les comunicacions entre les diferents parts es produiran mitjançant Sockets (de la classe java.net).

Així doncs el servidor obrirà un Socket passiu d'entrada de connexions (el ServerSocket de java.net) al port 2001, de manera que restarà escoltant constantment per aquest port les possibles entrades que es produeixin. Això ho farà entrant en un bucle infinit.

El Servidor s'encarregarà d'obrir un *Thread* per cada connexió entrant, de manera que cada *Thread* gestioni un usuari, al qual es proporciona un Socket actiu per tal que es pugui comunicar.

S'utilitzaran mètodes tipus synchronized per tal que tots els clients no utilitzin els mètodes de comunicació a l'hora, cosa que produiria un desgavell considerable, ja que estarien utilitzant tots a l'hora els canals d'E/S.

El Client per la seva part rep del Servidor el socket assignat un cop hem realitzat la crida al Port 2001, pel qual procedirà a comunicar-se.

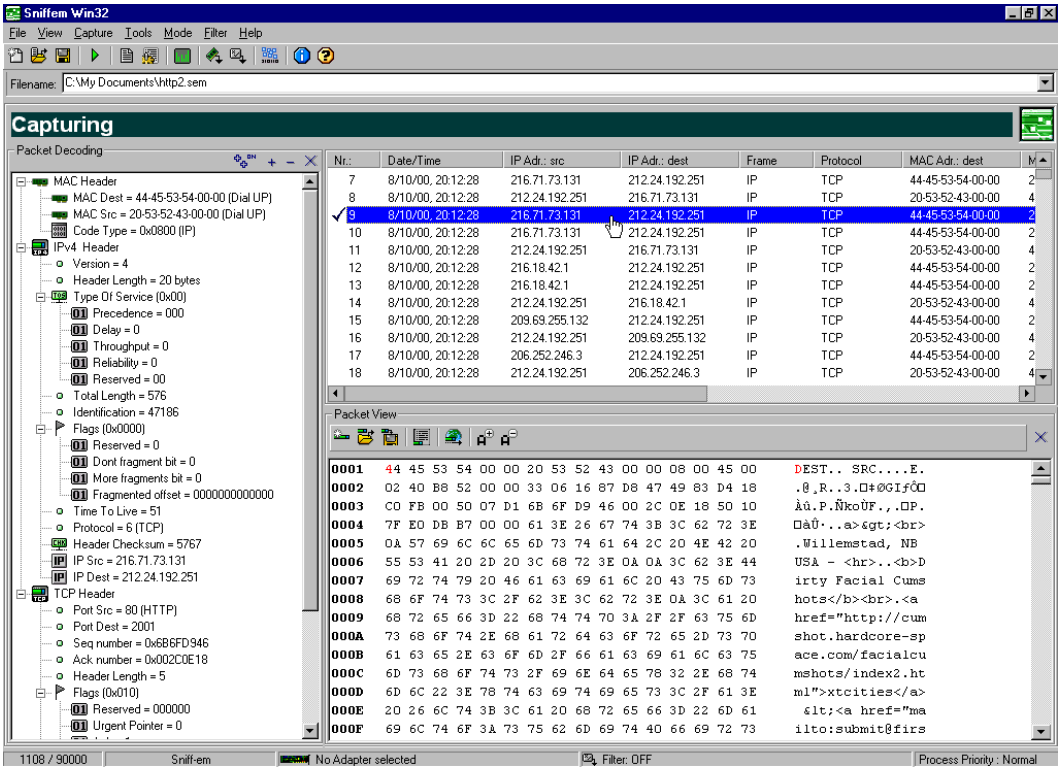
Un cop tenim connectat el client al Socket que ens ha donat el Servidor, procedim a crear un *Thread* de la mateixa manera com ho hem fet al servidor, de manera que aquest sigui l'encarregat de processar la informació.

3.3 Funcionament del xifratge

Cal enviar la informació codificada, com ja s'ha comentat, per tal d'evitar que connexions malicioses puguin tenir accés a la informació que els usuaris puguin enviar per la xarxa, garantint així la confidencialitat dels missatges enviats a cada moment.

Estem a un moment on cada cop es dona més importància a les comunicacions segures, degut el gran abast que tenen les xarxes arreu del mon i la relativa facilitat que hom pot tenir de capturar les trames IP que son enviades arreu del món a la cerca del seu propi destí.

En tot moment aquesta informació passa d'un ordinador a un altre, i qualsevol usuari amb un programa d'Sniffing pot capturar trames IP i treure'n la informació que continguin.



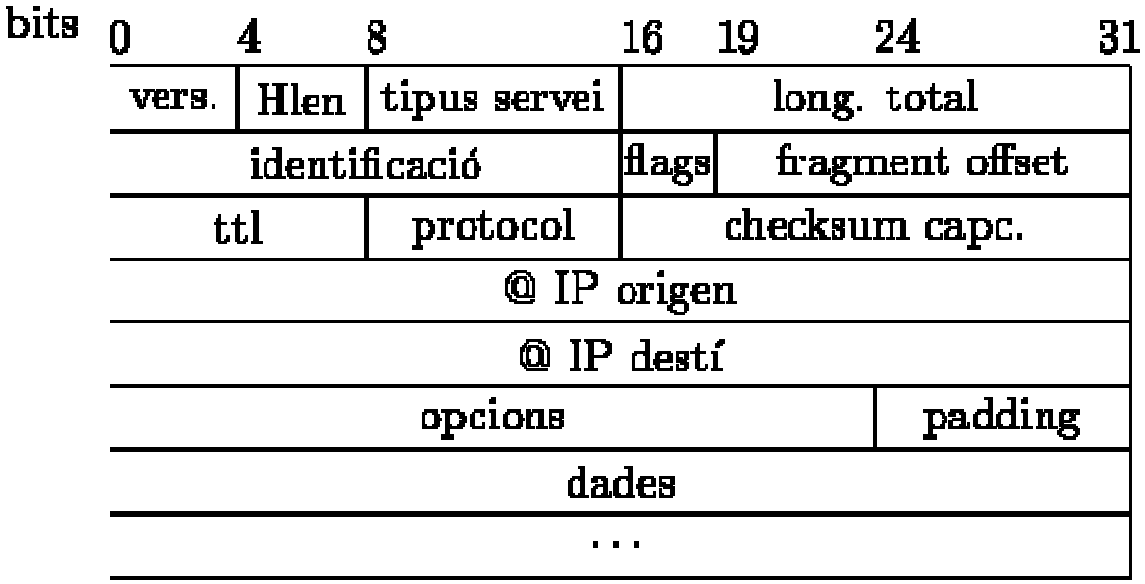
Programes com l'EtherReal, l'SPY, l'Snort o l'Sniffem, capturen trames i permeten el seu tractament d'una manera molt senzilla i accessible per a qualsevol usuari mitjanament avançat.

Per aquest motiu cada cop es dona més importància a la seguretat a les comunicacions, i s'utilitzen més tècniques de xifratge, finger print, protocols de seguretat, etc...

Per aquest motiu la intenció d'aquest Xat és la de garantir als usuaris que l'utilitzin una total confidencialitat a les seves dades, garantint que allò que transmetin serà llegit només per la persona a la que va dirigida.

Per aconseguir aquest objectiu tant important, cal dotar a l'aplicació d'una sèrie de protocols de seguretat, realitzant un xifratge de les dades, que impossibiliti la seva lectura a qualsevol usuari que pogués capturar-ne les trames.

Utilitzarem Sockets que funcionen sobre trames TCP/IP, doncs son comunicacions actives i questes trames poden ser capturades, i les dades queden a l'abast de qualsevol usuari maliciós.



Per aquest motiu cal aconseguir xifrar les dades, garantit que el seu pas pels canals de comunicació basats en TCP/IP, no seran accedits per ningú que no sigui el destinatari de la comunicació.

Per tant utilitzarem un xifratge basat en algoritme simètric TripleDES.

Cal garantir que els dos usuaris que intervenen a la comunicació es posen d'acord a la clau pública que utilitzen, i per aquest motiu el programa ha d'utilitzar el protocol Diffie-Hellman, que permeti garantir que la clau secreta que es genera no pot ser interrompuda per cap usuari maliciós.

L'usuari que origina la comunicació establirà els paràmetres necessaris, generant-los o recuperant-los del fitxer XML del que s'ha parlat anteriorment, enviarà aquests paràmetres i la clau pública al destinatari de la conversa, el qual generarà les seves pròpies claus i retornarà la seva clau pública a l'usuari inicial.

Un cop tots dos usuaris estiguin d'acord amb les claus secretes a utilitzar, s'iniciarà la conversa, la qual garanteix que cap altre usuari pot llegir-ne els missatges.

Em garantit amb això la seguretat complerta a la comunicació.

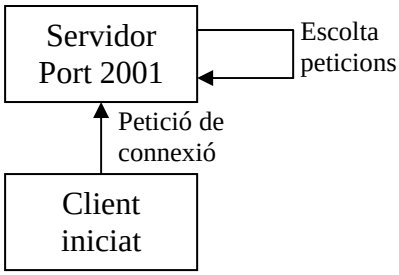
3.4 Disseny Servidor

La funcionalitat del Servidor es garantir la connexió entre els usuaris, i realitzar el intercanvi de missatges entre ells, ja siguin de conversa o de protocols d'establiment de la mateixa.

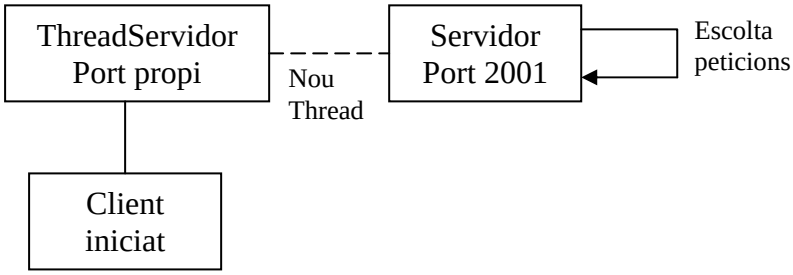
La connexió la mantindrà, mitjançant la potència que ens proporciona el paquet java.net, mitjançant la implementació de Sockets i ServerSockets.

El Servidor generarà un ServerSocket al port 2001 de captura de connexions.

Tota petició que rebi el Servidor al seu port 2001, generarà un nou fil d'execució al qual s'assigna un nou port que es connecta directament amb el Client que ha realitzat la petició.



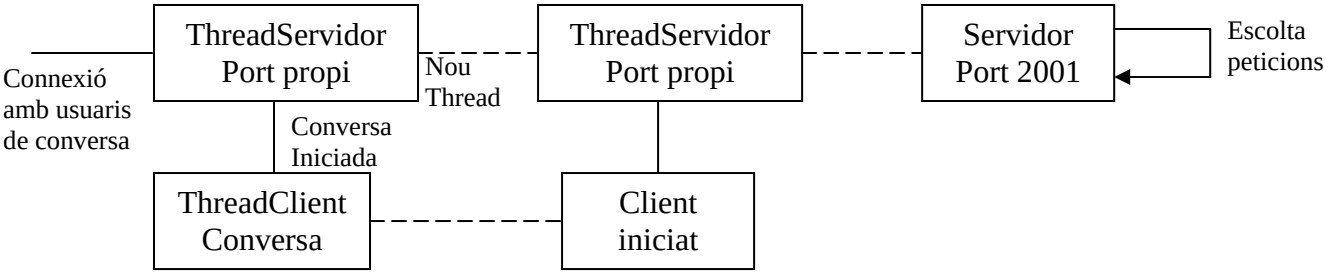
Un cop rebuda la petició de connexió, el Servidor genera un nou fil de execució que donarà servei a aquest usuari, mantenint un port propi per tal de crear el vincle entre Servidor i usuari.



Amb aquest nou fil es donarà servei al client iniciat, i el Servidor segueix escoltant peticions per donar servei als nous usuaris que es vulguin connectar.

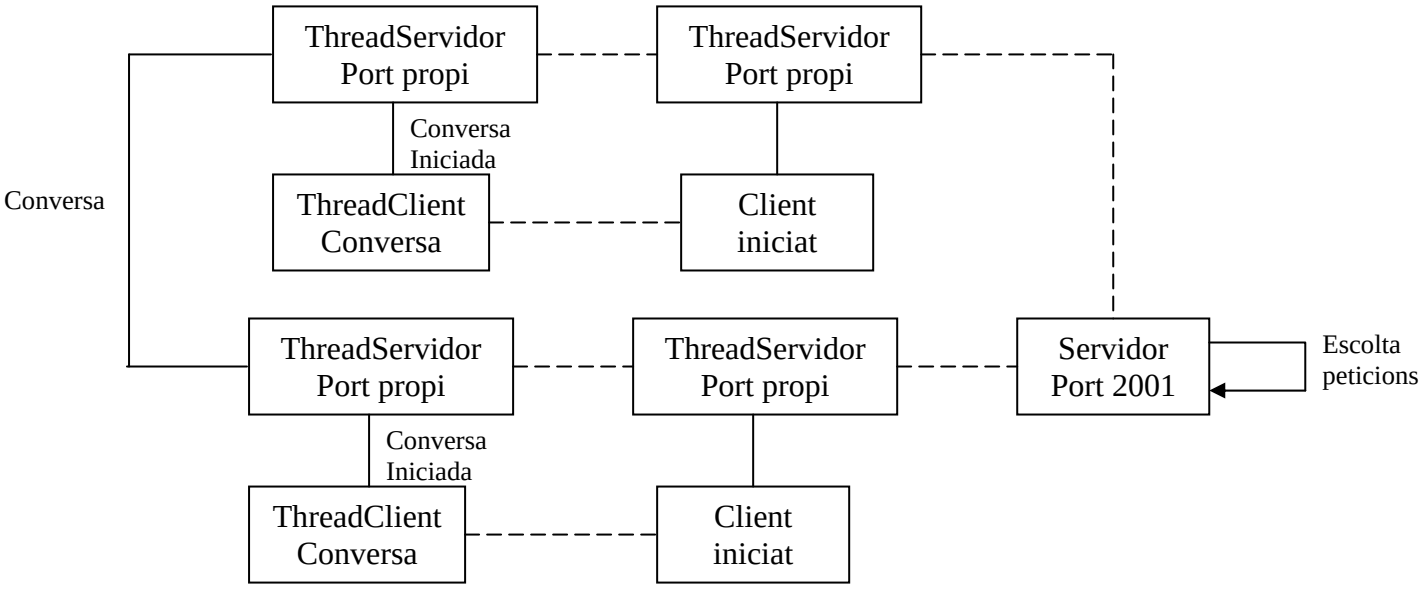
El ThreadServidor per la seva part, dona servei a Client, primer rep el seu nom o nick, i després resta enviant constantment la llista d'usuaris connectats a l'aplicació.

En el moment que Client decideixi iniciar una conversa amb un usuari qualsevol, el ThreadServidor llençarà un nou ThreadServidor que es connectarà amb el ThreadClient adient per mantenir la conversa.



En aquest punt el ThreadServidor manté una llista amb els usuaris que formen part de la conversa, amés d'estar relacionat amb l'usuari amb que està connectat. D'aquesta manera es poden enviar els missatges a tots els usuaris d'una conversa o a un o uns en concret.

El ThreadServidor te com a funció enrutar tots els missatges que rebi, cap als destinataris adients en cada moment.
A més conté els mètodes propis de la conversa, per tal d'enviar i rebre la informació, així com els mètodes de tractament d'usuaris de conversa.



D'aquesta manera s'aconsegueix mantenir múltiples usuaris, mantenint múltiples converses entre ells.

La tasca principal del Servidor és garantir aquesta comunicació de manera constant.

3.5 Disseny Client

La part client explotarà les funcionalitats que ens ofereix el servidor, possibilitant tenir diferents converses, rebre'n, generar-ne, i mantenir la llista de connexions a cada moment, així com gestionar els protocols de càlcul de clau pública i els mètodes de xifratge.

El client ha de sol·licitar una petició de connexió al servidor, i un cop la rep, es manté en connexió constant amb el mateix, rebent la llista d'usuaris connectats.

Aquesta interfície gràfica de Client, mostra la benvinguda que el Servidor dona als clients que es connecten, i mostra la llista esmentada d'usuaris, sobre la qual es possible demanar un usuari amb qui començar la conversa.

Un cop es demana un usuari amb qui conversar s'obre una nova finestra que serà la que manté la conversa amb el client destí, de manera que el client tindrà la pantalla d'usuaris connectats i n pantalles amb les n converses que tingui establertes a cada moment.

En el moment d'obrir una conversa es realitzen els protocols per l'establiment de claus pel xifratge de la informació.

El inici d'una conversa es pot veure des de dos punts diferents:

- Client que inicia la conversa
- Client amb conversa entrant

Client inicia la conversa

Es fa un doble-click al damunt d'un usuari de la llista amb el qual es vol iniciar la conversa.
El procés genera un procés fill que es connecta amb una nova instància del servidor, i omple la llista amb el propi nom i el nom de l'usuari destí de la conversa.

S'inicien els protocols de seguretat, mirant si ja s'havia iniciat la conversa amb aquest usuari per agafar les claus de l'agenda XML.
Es decideixen les claus públiques i s'obté la clau secreta. Quan l'usuari destí dona la seva conformitat, es procedeix al inici de la conversa, xifrant i desxifrant la informació que s'envia o es rep dels sockets.

Client rep la conversa

Quan un client una petició de conversa, llença un fil d'execució paral·lel, es connecta amb el ThreadServidor que es genera per ell, gestiona els usuaris (rep per paràmetre qui ha iniciat la conversa amb ell) i es disposa a rebre la clau pública i els paràmetres per part de l'usuari que ha originat la conversa.

Un cop rebuts, genera la clau secreta, retorna la seva clau pública i es disposa a iniciar la conversa amb l'usuari que l'ha sol·licitat.

3.6 Disseny gràfic

El disseny gràfic ha de correspondre a una interfície gràfica amb una GUI amigable i senzilla d'usar. A ser possible seria interessant seguir la pauta d'altres Xats coneguts per tothom com per exemple el Messenger, doncs un programa amb una interfície que es faci fàcil de reconèixer te més possibilitats d'èxit que una de nova i impactant.

Així doncs la primera idea pot ser generar una interfície que consti d'una gran pantalla de missatges, per la qual es vagin mostrant els missatges entrants, una llista d'usuaris connectats que podria recaure a la dreta de la llista de missatges entrants. També serà necessària una caixa de text on l'usuari pugui escriure i un botó per realitzar l'enviament del missatge. A més caldrà un botó per tancar la finestra.

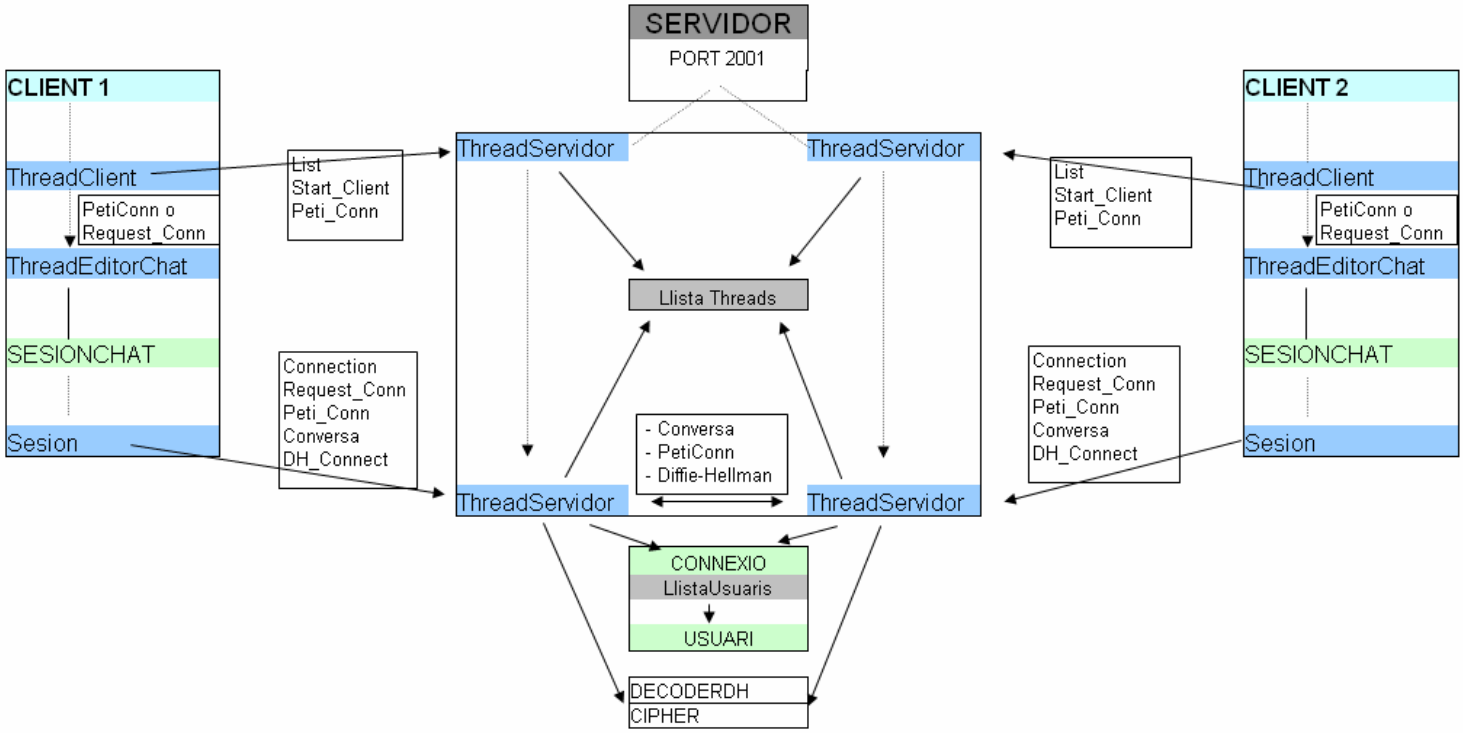
En aquest cas, per possibilitar la concurrència de converses, la llista d'usuaris connectats hauria de permetre de generar una conversa paral·lela, mitjançant un simple doble-click de ratolí al damunt de l'usuari amb el que volem iniciar una nova conversa, de manera que es generi una altre pantalla idèntica a la que tenim.

D'altra banda, això també es podria fer realitzant una pantalla prèvia en la que es mostrin els usuaris connectats, i nosaltres escollim amb quin iniciar la conversa mitjançant el doble-click.
En aquest moment s'obriria la pantalla descrita anteriorment, tornant a la llista d'usuaris en cas de voler obrir una nova conversa.

En tot cas, la conformació gràfica serà la mateixa, i ens decantarem per aquella que cobreixi millor les necessitats en el moment del desenvolupament.

4 – ASPECTES CONCRETS DE LA IMPLEMENTACIÓ

Actualment l'aplicació es basa concretament pel següent esquema, que és una representació molt general del seu funcionament, que tot seguit passem a explicar:



El ***Servidor*** escolta constantment peticions de connexió al port 2001, que està controlat per ServerSocket Java.

Un cop el ServerSocket.accept() rep una petició de connexió des del Socket d'un ***Client***, es genera un ***ThreadServidor*** que proporciona una connexió única al ***Client*** a Socket concret, que serà el PORT d'entrada utilitzat pels dos per mantenir el contacte de missatges (de sistema).

Realment, el que es posa en contacte amb ***ThreadServidor*** és ***ThreadClient.*** Ambdós entren en un bucle infinit de missatges, iniciat pel missatge "Start_Client", fins que es rebi el missatge concret de desconnexió. Aquests missatges son de llista d'usuaris connectats ("LIST") i d'establiment de connexió entre clients per a realitzar una conversa ("Peti_Conn").

Un cop generat el Socket, s'obre la part gràfica de ***Client*** , qui ha llençat el ***ThreadClient***.

Cal esmentar que ***ThreadServidor*** utilitza la LlistaDeThreads per controlar els Threads actius en cada moment.

Quan un usuari decideix iniciar la conversa amb un altre usuari, el ***Client*** llença un Thread nou per ***SesionChat*** que es la classe que genera la pantalla de conversa. Aquesta classe generarà també un Thread nou anomenat ***Sesion***, que és el Thread de conversa.

Aquesta ***Sesion*** envía diferents missatges al servidor, per tal d'establir la connexió amb un altre ***Client***, que en rebre la "Peti_Conn" al seu ***ThreadClient***, genera un nou Thread que executarà la seva ***SesionChat*** i aquesta la seva pròpia ***Sesion.***

Un cop en aquet punt, es genera l'intercanvi de claus Diffie-Hellman, utilitzant la classe ***DecoderDH*** i es xifren els missatges amb la classe ***Cipher***.
Els ***ThreadServidor*** associats a ***Sesion*** es mantenen a una llista de ***Conversa***(s), que manté els ***Usuaris*** connectats a aquesta ***Conversa***.

Un cop arribats a aquest punt, es pot procedir a la conversa.

Un cop presentat el resum de l'aplicació, passem a introduir-nos de ple al funcionament real de l'aplicació:

4.1 SERVIDORXAT.CLASS

La classe ServidorChat és l'encarregada d'anar generant Threads que es lliguen als usuaris que es connecten a l'aplicació.

Per a cada usuari nou que s'iniciï es genera un Thread nou que li dona servei.

Aquesta classe escolta des del PORT 2001 per tal de rebre les peticions.

Es generen tres variables de classe; PORT, AUTO_DISCONNECTION i socketServidor.

PORT es una variable Static i private que indica el nombre del port on escoltarà el socketServidor. En el nostre cas és el 2001, com ja s'ha comentat diverses vegades en aquest document.

AUTO_DISCONNECTION és una variable que serveix per mesurar el temps de inactivitat d'un usuari. Inicialment la inactivitat serà de 20 minuts.
Aquesta variable és utilitzada a cada ThreadServidor nou que es vagi generant, doncs es tancarà la connexió amb l'usuari en funció d'ella.

socketServidor és la variable ServerSocket que escoltarà constantment des del port PORT i llençarà ThreadServidor(s) per cada petició dels usuaris que es connecten.
El constructor de la classe mostra un missatge d'inicialització de la aplicació, i posteriorment llença els mètodes arrancarServidor () i porcessarClients().

Aquesta classe és main, i per tant des de la main s'executa un new ServidorChat().

El constructor de la classe mostra un missatge d'arrencada de servei i llença els dos mètodes de la classe, que son:

arrancarServidor() genera la instància de new ServerSocket (PORT) a la variable de classe socketServidor, i posteriorment mostra un missatge de servei arrencat.

Aquest mètode genera llença tres Exception : BindException, SecurityException i IOException.

BindException : es llença si el port està ocupat.
SecurityException: Si hi ha restriccions de seguretat al port.
IOException: Per qualsevol error d'E/S.

processarClients() es el mètode que resta escoltant qualsevol petició al PORT.

El procés entra a un bucle infinit de manera que amb el mètode socketServidor.accept() va rebent totes les entrades.

El mètode accept() de la classe ServerSocket retorna un objecte de tipus *socket*.

Aquest objecte es recupera i es passa com a paràmetre al ThreadServidor que es genera un cop hem rebut aquesta entrada.

D'aquesta manera estem passant la connexió assignada al ThreadServidor generat, per tal que es posi en contacte amb el client que ha generat la connexió.

Si es produeix un error s'intenta tancar el Socket de client per si aquest hagués estat inicialitzat.

El mètode genera les Exception : IOException i SecurityException.

IOException: es genera per cada possible error d'E/S, molt probablement per que l'usuari no estigui actiu (ha tancat la connexió o s'ha produït algun error).
SecurityException: Problemes de seguretat.

errorFatal () és el mètode que gestiona els missatges d'error de l'aplicació.

Amb el mètode showMessageDialog de la classe JOptionPane es mostren missatges emergents amb els possibles errors que es vagin generant.

4.2 THREADSERVIDOR.CLASS

ThreadServidor es genera com a resultat d'una acceptació de connexió per part de ServidorChat. Aquesta classe extends de Thread, de manera que es converteix en un Thread nou en el moment de cridar el seu mètode run() amb la crida start ().

Aquesta possiblement sigui la classe més complexa i complicada de totes, doncs dona diferents serveis, tant a classes de tipus ThreadClient (clients passius) com a les Session (clients actius o en conversa).

Per tant, per entendre el funcionament d'aquesta classe caldrà diferenciar entre el que son **clients actius** i **clients passius**.

- Clients actius: son aquells Threads de client que estan en plena conversa o inicialitzant-la, amb la qual cosa tenim missatges de tot tipus, com per exemple d'acord de claus, de conversa, de paràmetres...
- Clients passius: son aquells Threads de client que mostren al mateix la llista d'usuaris connectats i reben connexions en cas que un altre client demani una conversa amb ells. Aquests clients pràcticament només reben informació i n'envien molt poca.

La classe ThreadServidor dona servei tant a uns com als altres, fent-la relativament complicada.

Aquesta classe genera un nombrós grup de variables:

- Identificadors d'usuari : Variables que identifiquen els noms d'usuari amb que es connecta i de l'usuari destí d'una conversa, si es un client actiu.
  Es generen dos noms per cada client, el nom real que es correspon amb el nick, i el nom que es genera en cas de ser una conversa, client actiu, que es el nick més el port. En cas de tractar-se d'un client passiu només tindrà nom real.
  Pels usuaris destí de conversa també tindrem dos noms, el nick i el nick+port (només clients actius).
- Variable static d'històric de connexions. Fitxer de log de connexions.
- Llistes estàtiques de clients actius
  - Llista de clients actius: Usuaris connectats a l'aplicació (tant actius com passius).
  - Llista de clients en conversa: usuari amb el que s'està en conversa. Es manté a una llista per permetre l'adaptació a converses de multi-usuari.

    - Llista de clients actius reals: Usuaris connectats a l'aplicació (només clients passius).
- Objecte connexió: Els clients actius generen un objecte connexió que manté la relació amb els usuaris de la conversa, així com els noms real i figurat (nick+port) del mateixos.
- Socket de la connexió, rebut per paràmetre.
- Embolcalls dels canals d'entrada i sortida.

El constructor de la classe activa el socket de connexió, i embolcalla els canals d'entrada i sortida del mateix amb les variables adients per tal de realitzar la comunicació.

Escriu al historial el inici del programa i llença l'start() del Thread, executant el mètode run () que inicia el procés en paral·lel.

Aquest procés llegeix a la variable String textoUsuario les entrades realitzades des del socket, és a dir, els missatges enviats pel client.

El primer valor rebut és un tag que identifica un dels tres tipus d'usuari possibles:

- @START_CLIENT@
  És l'entrada dels clients passius. És el inici de connexió d'un client.
  En aquest cas se li envien dos missatges per tal que doni el nick que vulgui.
  Estava pensat de guardar aquest nick per que fos etern, permetent modificar-lo, però no era una funcionalitat imprescindible i per tant s'ha optat per no desenvolupar-la, per un obvi motiu de temps.
  Es rep el nom client, en cas que sigui null, se li assigna el nick "Convidat".

- @CONNECTION@
  Un client actiu inicia una conversa amb un usuari. Rebem la inicialització d'aquest nou thread del client, el qual ens envia el seu nom real (nick), que assignem a la variable nomReal.

- @REQUEST_CONNECTION@
  Un client actiu rep una conversa des d'un usuari. Rebem la inicialització d'aquest nou thread del client, el qual ens envia el seu nom real (nick), que assignem a la variable nomReal.

Cerquem a la llista d'usuaris actius si ja existeix un usuari amb aquest nom, i en cas de trobar-lo li afegim el port al darrera. En realitat això es farà sempre per clients actius, doncs el seu nomClient és igual al nomReal, i això no pot ser. El nomReal serà el nick i el nomClient serà el nomReal+Port.

En cas dels clients passius realment només es farà això si dos clients es diuen de la mateixa manera exactament.

S'afegeix el Thread a la llista de clients actius amb el mètode synchronized, addConexion(). Aquest mètode és sincronitzat doncs treballem amb una llista static, per tal que tots els usuaris accediran a la mateixa.

Si el client és passiu, s'envia el missatge que informa a l'usuari del seu nick assignat.

S'assigna el timeout al socket amb el mètode setSoTimeout().

Arribats a aquest punt el procés entra a un bucle infinit de lectura de missatges del qual no sortirà a menys que es tanqui la connexió, ja sigui per part del client, per Timeout o per un error en algun moment del procés.

Entrem aquí a tot un seguit de TAGs que identifiquen les accions que s'estan realitzant a cada moment;

- CLOSE: El procés es tanca, doncs l'usuari així ho ha demanat, o per que s'ha assolit el temps de TIMEOUT.
  En cas que estiguem tancant una conversa, es tancarà únicament aquesta conversa, com ha de ser, però en cas d'estar tancant la part passiva del client, es considera que s'apaga tot el programa, i per tant es tanquen totes les converses que aquest client tingui arrencades.
  Realitza un apunt a el historial.

- LIST: Envia a l'usuari tots els usuaris connectats, és a dir, tots els usuaris passius (els seus nicks), de la llista de clients actius reals.

- [RECEIBED]: Tag que indica al client que inicia una conversa que ha rebut la publicKey correctament. S'envia al destinatari directament.

- @~CONVERSA~@: Enviament normal de missatges xifrats d'un client a l'altre. Estem a plena conversa entre dos usuaris de l'aplicació.
  S'envia el següent:

  ```
  escribirACliente ("@~CONVERSA~@"+ nomRealClient + "> @CONVERSA@" + textoUsuario.substring(12));
  ```

  El Tag s'envia tal qual el rebem, després enviem el nostre nom real (nick) >, per que a l'usuari destí se li mostri qui li està enviant el missatge, i després un altre tag semblant @CONVERSA@.
  Aquesta nova marca s'envia per identificar al client que ho rep, que després d'aquesta marca arriba el missatge que ha de desxifrar.

- @CONNECTION@: Aquesta entrada és el segon enviament que ens farà un client actiu quan inicia una conversa. El primer enviament l'hem capturat a l'inici del Thread i ens enviava el seu nom. A aquest nou enviament ens envia el nom del client amb el que vol iniciar la conversa.

  Cal tenir amb compte que el nom que estem rebent és el nom del client passiu, i no del client actiu que es generarà quan enviem els primers missatges de connexió. Un cop enviem els primers missatges de connexió a l'altre client, aquest generarà un thread que serà amb el que tractarem realment.

  Això ens planteja el dilema que primer enviem un missatge al client passiu, i després mantindrem la conversa amb el client actiu.

  Per aquest motiu jugarem amb el nom del client.

  Tenim el nom real, que és el nick o el nom del client passiu i el nom destí de la conversa que és el nick+port o el nom del client actiu.

  Utilitzarem la classe connexió, que guarda duples (nomCoversa, nomReal), i crearem a aquest punt un nou objecte connexió, enviant per constructor el nostre nomConversa (nick+port) que ja el tenim assignat, i el nostre nomReal (nick) que el tenim guardat. Aquest serà el primer element de la llista de converses de conversa.

  Tot seguit afegim la dupla (nomReal, “”). Aquí estem introduint el nom de l'usuari passiu destinatari de la nostra petició de connexió al lloc del nom de client actiu, i on està el nom de client passiu un blanc.

Això servirà per quan l'usuari destinatari de la conversa es connecti a la conversa, identifiqui la seva posició, modifiqui aquests valors pels correctes (nomConversa, nomReal) i quedi establerta la connexió, i ambdues parts es puguin identificar correctament.

Finalment afegim la conversa a la llista de converses actives per aquest ThreadServidor (o usuari actiu).

- @~DH_DECIDE~@: El client actiu que està inicialitzant la conversa utilitza aquest punt per tal d'enviar els valors del protocol Diffie-Hellman al client destí de la conversa.

  És des d'aquest punt que s'envia el primer missatge de petició de connexió al client passiu amb que volem iniciar la conversa.

  Per tal de fer-ho s'executa el mètode syncrhronized findContacte (String as_NomDesti, String as_nomClient, String as_nomRealClient).

  Aquest mètode realitza un cerca per Iterator de totes les connexions que hi ha a la llista de connexions, per tal de cercar la nostre, i mirar quina connexió te el nom client origen i el nom client destí de la conversa, segons els paràmetres de la funció.

  Quan la troba, retorna la cadena següent:

  "@PETI_CONN@"+as_nomClient+"@USER@";

  Que adverteix al client passiu que la rebrà, que es tracta d'una connexió entrant, que realitza as_nomClient (que identificarà per la cadena @USER@)

  Al client passiu destinatari d'una conversa s'envia aquesta cadena amb els paràmetres Diffie-Hellman darrera.

- @~DH_CONNECT~@: Quan es rep el dh_connect, estem rebent la clau pública per part de l'usuari destí de la conversa.

  Això vol dir que aquest TAG el rep el ThreadServidor associat al client actiu destí.

  Com Servidor li diem al client que ja estem disposats a rebre el binary de la publicKey, enviant un missatge de [WAITTING], i procedim a rebre aquesta clau i l'enviem al client actiu que ha iniciat la conversa.

- @REQUEST_CONN:@ de la mateixa manera que amb CONNECTION, el client destí de la conversa envia un nou TAG de connexió al seu Thread, per tal d'informar-lo quin és l'usuari que li està sol·licitant la conversa.

  A més executa el mètode iniciaConversa (String nomOrigen, Strin nomClient, String nomReal), que és l'encarregat de modificar els noms que han quedat pendents de modificació al punt @CONNECTION@, quan encara no s'havia executat el ThreadClient destí.

  Ara ja estem al ThreadCLient destí, i per tant ja sabem quin és el nomConversa, i per tant haurem de modificar l'Objecte conversa assignat a nosaltres, i posar (nomConversa, nomReal) a la dupla identificativa de la nostra connexió.

  Amb això, l'altre(s) usuari(s) de la conversa ja ens pot identificar com a connectats i enviar-nos missatges.

- [ACK_PUBLIC_KEY]: el client actiu que inicia la conversa, utilitza aquest punt per tal d'enviar la clau pública al client actiu destí.

  S'envia un TAG [WAITTING] de conformitat a l'enviament, i el client envia la cadena de bytes de la clau pública, que el ThreadServidor envia al client destí.

- [ACK_TRANSMISSION]: TAG utilitzat a mode de semàfor per indicar que un client està disposat a rebre dades.

La classe te diferents mètodes per a realitzar l'enviament de dades, tant Strings com binaries als diferents usuaris de la conversa, realitzant passades sistemàtiques pels clients actius i clients de conversa.

4.3 CLIENT.CLASS

La classe client genera la interfície gràfica bàsica per tal d'oferir a l'usuari la capacitat d'introduir el seu nick, i escollir de la llista d'usuaris amb qui vol iniciar una conversa.
Aquesta classe es connecta al socket del servidor utilitzant el mètode constructor de socket (host, port), de manera que es retorna un objecte socket que conté la connexió i els canals d'entrada i sortida.

La classe executa un Thread que es connecta amb el ThreadServidor generat per la connexió al servidor.

A més, te els mètodes adients per tal d'executar un thread nou en cas que l'usuari decideixi establir una conversa amb un altre usuari de la seva llista, o per llençar el mateix Thread en cas que rebi una petició de connexió per part d'un usuari connectat.

Amb el mètode següent es realitza una petició a un usuari per selecció a la llista:

```
list.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent e) {
        if (e.getClickCount() == 2) {
            int index = list.locationToIndex(e.getPoint());
            LlençaFinestraChat ((String)list.getSelectedValue(), ls_Client, "", null);
        }
    }
});
```

El mètode LlençaFinestraChat és el que solicita aquesta connexió.

Es passa el propi nom, el nom del destinatari, un paràmetre que representa el missatge, pel nostre cas null, i un altre paràmetre que finalment no ha calgut utilitzar.

4.4 THREADCLIENT.CLASS (client passiu)

Un cop un client es posa en marxa, i es connecta am el servidor, s'executa la classe ThreadClient extends Thread, que és un procés paral·lel que està en connexió amb el servidor mitjançant socket.

Aquest Thread te una feina molt senzilla; primer de tot informa al Thread que és un client passiu. Per tant el servidor li demana el nick per pantalla, i l'usuari l'ha d'enviar.

Un cop realitzat aquest senzill procés, el procés resta escoltant missatges per part del servidor pel seu canal d'entrada, que s'hereta de la classe client, a on s'han generat, i tant sols escolta dos tipus de peticions:

- LIST: quan el servidor envia la llista d'usuaris connectats, es recuperen aquests obviant el propi usuari, i es torna a carregar la llista de la pantall d'usuari amb els nous valors. Això es fa cada segon

- @PETI_CONN@: Rep una petició de connexió per part d'algun usuari, recupera el nomUsuari i el missatge, utilitzant el TAG separado @USER@, i llença la finestra Xat utilitzant el mètode de client.class de la manera següent:

```
cliente.LlençaFinestraChat(ls_nomClient, cliente.ls_Client, ls_missatge, null);
```

El procés envia cada cop el missatge LIST al servidor per anar rebent les successives llistes d'usuaris connectats.
A més, comprova que les dades rebudes no siguin un missatge de tancament, cas que voldria dir que l'usuari o el timeout han tancat la sessió.

Ho identifica com a tal, avisa a l'usuari i tanca la sessió.

4.5 SESSIONXAT.CLASS

La classe SesionChat és l'encarregada de mostrar el front-end d'usuari del Xat pròpiament dit.

Aquesta classe ofereix les eines gràfiques que permeten a l'usuari escriure missatges per enviar-los a l'altre usuari de la conversa.

El constructor de la classe rep per paràmetre els valors del client que ha iniciat la conversa, el client destí de la conversa, el missatge (que si és informat vol dir que el client és client actiu destí) i una cadena de bytes que era utilitzada per passar la clau pública, però que ara mateix no s'utilitza.

Aquesta classe inicialitza els components gràfics utilitzats, i posteriorment es connecta amb el socket adient del servidor, proporcionat per la classe Socket, embolcalla els canals de comunicació amb els buffers de caràcters i bytes necessaris, i llença una Sesion, que és la part del client que porta el major pes específic de tot el programa Client.

Els diferents events dels diferents components de la classe;

Funció que tenca la connexió en cas que es tanqui el frame.

```
protected void processWindowEvent (WindowEvent e){
        super.processWindowEvent(e);
        // Si es tanca el frame enviem l'ordre de desconnectar al servidor.
        if (e.getID()==WindowEvent.WINDOW_CLOSING){
                sortida.println ("CLOSE");
        }
}
```

S'envien els missatges pel flux de sortida. Com es pot observar, és a aquest punt on es xifren els missatges que son enviats als altres usuaris.

```
void txtMessage_actionPerformed (ActionEvent e){
System.out.println (txtMessage.getText());
        sortida.println ("@~CONVERSA~@" + Encriptador.encrypt(txtMessage.getText()));
        txtMessage.setText ("");
```

```
}
```

Botó tancar, que tanca el frame.

```
void btnCerrar_actionPerformed (ActionEvent e){
        sortida.println("CLOSE");
}
```

4.6 SESSION.CLASS (client actiu)

Aquesta és sens dubte la classe més important de la part client.

Es tracta de la classe que extends Thread, per tant és un Thread, que realitza la comunicació amb el servidor a nivell de conversa, és a dir, és la classe encarregada d'iniciar una conversa, o rebre la inicialització d'una conversa, entrar en contacte amb la sesion del client destí o origen, depenent si es tracta d'un o l'altre, establir els protocols de seguretat Diffie-Hellman i enviar els missatges pròpiament dits, així com desxifrar-los, i és qui inicialitza la classe Cypher a la variable Encriptador, que es declarada a SesionChat.

Es generen tres variables privades de classe, una SesionChat, per rebre la SesionChat que la crida com a paràmetre, secretKey, per tal d'emmagatzemar la clau que utilitzarem pel xifratge de dades, i la binaria clauPublica, que manté les successives passades de la clau entre diferents usuaris (en principi dos, però és adaptable).

Al constructor de la classe es rep com a paràmetre la SesionChat que la crida, com ja ha estat comentat, i s'assigna a la variable SesionChat creada a l'efecte.

Posteriorment es llença el Thread amb l'start() que llença el mètode run().

Dins de run () cal destacar la instancia d'una variable tipus DeclderDH, que és el decodificador Diffie-Hellman del que parlarem posteriorment.

Com és lògic pel que s'ha comentat fins ara, aquesta classe te dues maneres de funcionar:

- Si és classe iniciadora de conversa
- Si és classe que rep una conversa

S'intentarà anar explicant el funcionament des dels dos punts de vista alhora, identificant els missatges segons siguin **CAI** (Client Actiu Inicial) o **CAD** (Client Actiu Destí).

La classe SesionChat ha rebut per paràmetre el nom del client, el nom del destí i un *missatge*, com s'ha comentat, provinents del ThreadClient.

En cas que estiguem iniciant un CAD aquest missatge vindrà informat, informant-nos que un usuari ens està fent una sol·licitud de conversa.
Anàlogament, el missatge serà un caràcter buit en cas que siguem un CAI.

Primer de tot el CAI envia al ThreadServidor el nom del propi client i el nom del client amb qui està iniciant la conversa.
Ambdós amb el TAG @CONNECTION@.

Posteriorment el CAI calcula els paràmetres del protocol Diffie-Hellman amb la classe DecoderDH, per enviar-los al CAD:

```
ls_VariablesDH = decoDH.getDHPublicKey(clienteChat.ls_Contacte);
```

Per enviar-los, el CAI utilitza el TAG @~DH_DECIDE~@, que el servidor interpretarà com el inici del intercanvi de protocols de seguretat entre CAI i CAD.

Arribats a aquest punt, el CAI resta esperant el missatge [ACK_TRANSMISSION] que ha d'enviar el CAD, conforme ja està preparat per rebre la clau pública en binari.

Degut al missatge @~DH_DECIDE~@ el ThreadServidor del CAI ha enviat un missatge al Client passiu destí de la conversa, i aquest client passiu ha llençat un nou Thread SesionChat, amb els paràmetres de nom d'usuari (nick+port), nom d'usuari que inicia la conversa i un missatge informat.

Arribat a Sesion el CAD està preparat per començar a funcionar.
Envia dos missatges @REQUEST_CONN@ amb el seu nom real (nick+port) i el nom de l'usuari que l'ha cridat.

Aquests missatges generen al ThreadServidor la indicació que la conversa ja és activa.

El CAD envia al CAI el missatge de [ACK_TRANSMISSION] per tal que aquest li enviï la clau pública.

El CAI rep el missatge, i es disposa a enviar la clau pública al servidor. Envia un missatge [ACK_PUBLIC_KEY] i espera el [WAITTING] que el servidor li enviarà quan pugui rebre la clau.

Un cop rebut el missatge, el CAI envia la clau pública en binari al servidor, que avisarà al CAD amb un [ACK_PUBLIC_KEY] que li envia la clau i el CAD la rebrà.

El CAD informarà amb un [RECEIBED] que ja te la clau pública, calcularà la seva pròpia clau pública i la secreta, i retornarà la pública al CAI.

Finalment el CAD envia un @~DH_CONNECT~@ per tal d'enviar la clau al CAI.

S'activa el codificador d'informació;

```
clienteChat.Encriptador = new Cypher (secretKey);
```

I el programa entra a un bucle infinit de rebuda de missatges.

Aquest bucle no fa cas de les trames LIST i les elimina directament.

L'únic tipus de trames que rep el programa son trames tipus @~CONVERSA~@, a les quals realitza el desxifrat de la informació utilitzant el TAG @CONVERSA@ per separar el nom d'usuari que es mostra per pantalla (tipus "Juan >") de la resta de missatge:

```
String ls_nomClient = linia.substring(12, linia.indexOf("@CONVERSA@"));
String msg = clienteChat.Encriptador.decrypt(linia.substring(linia.indexOf("@CONVERSA@")+10)) +
System.getProperty("line.separator");
```

El programa es tanca en cas de rebre alguna de les trames de tancament (voluntari o per TIMEOUT).

4.7 CONNEXIO.CLASS

Classe que s'utilitza per mantenir els clients que pertanyen a una conversa.

Cal tenir identificats en tot moment els clients d'una conversa, i fent-ho d'aquesta manera es permeten modificacions futures per tal d'adaptar el programa per permetre converses entre més de dos usuaris.

Ara mateix es podria solucionar per programa a una llista comuna als ThreadServidor d'una conversa, però d'aquesta manera és més net i es permeten les mencionades modificacions.

La Classe és senzilla, consta d'una llista d'Usuaris, sent Usuari una classe que guarda la dupla NomConversa, NomReal.

La classe connexió dona les eines per tal de treballar amb la llisa d'usuaris connectats, donar-ne d'alta, modificar-los, cercar-los...

4.8 USUARI.CLASS

La classe usuari ens permet de mantenir la dupla nomConversa, nomReal, que ens servirà per identificar usuaris d'una conversa.

El nomConversa és el nom que el Servidor ha assignat al fil del Client per la conversa que està mantenint, que és nick+port assignat.

El nomReal, és el nom amb que l'usuari s'ha identificat al inici de la sessió (nick).

La classe ofereix mètodes pel tractament i modificació de les dues variables.

4.9 DECODERDH.CLASS

Aquesta classe és l'encarregada de gestionar els paràmetres Diffie-Hellman, recuperar les claus pública i privada i obtenir el SecretKey amb el que codificarem i descodificarem la informació transmesa per la xarxa.

Consta de dues variables de classe; una PrivateKey i una PublicKey.

Mètodes:

genDhParams
Aquest mètode s'encarrega de la generació de paràmetres Diffie-Hellman, per tal de fer el càlcul de les claus.
Utilitza una classe *AlgorithmParameterGenerator* instanciat com DH (Diffie-Hellman), amb una inicialització de 1024 bits i generem els paràmetres a una classe *AlgrorithmParameter*.
Aquesta classe *AlgorithmParameter* genera un *DHParameterSpec* que generarà els paràmetres necessaris.

```
AlgorithmParameterGenerator paramGen = AlgorithmParameterGenerator.getInstance("DH");
paramGen.init(1024);

// Genera els paràmetres
AlgorithmParameters params = paramGen.generateParameters();
// Classe que recupera els paràmetres, generant una classe Diffie-Hellman amb els valors necessaris.
DHParameterSpec dhSpec = (DHParameterSpec)params.getParameterSpec(DHParameterSpec.class);
```

getDHPublicKey
Mètode que recupera la clau pública per una crida per part del CAI.

Crida el mètode *genDhParams* per tal de generar els paràmetres, però primer mira si al fitxer XML d'agenda ja hi tenim definida una connexió per aquest usuari.

En cas de no tenir definida una connexió per aquest usuari, llençarà el mètode esmentat, i desarà els resultats al fitxer agenda.

Utilitza un generador de claus *KeyPairGenerator* instanciat com DH, l'inicialitza amb el *DHParameterSpec* anterior, i genera les claus a una classe *KeyPair*, que manté el resultat de la privateKey i a publicKey.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DH");
DHParameterSpec dhSpec = new DHParameterSpec(p, g, l);
keyGen.initialize(dhSpec);
KeyPair keypair = keyGen.generateKeyPair();

// Get the generated public and private keys
privateKey = keypair.getPrivate();
publicKey  = keypair.getPublic();
```

Es retornen els paràmetres i el missatge, que ara ha quedat en desús.

getDHPublicSecretKey
És el mateix mètode que l'anterior, però per l'usuari CAD.

Rep els valors DH per paràmetre i després cerca la clau secreta.

getSecretKey
Mètode que genera la clau secreta amb que es codificaran i descodificaran els missatges.

Primer codifica la cadena binaria que rep a un objecte PublicKey. Cal recordar que el encoded() de la publicKey (que es com es passa a binari per tal d'enviar-la) codifica en format X509 la publicKey, i per tornar-la a passar a publicKey, cal utilitzar la classe X509EncodedKeySpec i el factory KeyFactory instanciat a DH:

```
X509EncodedKeySpec x509KeySpec = new X509EncodedKeySpec(publicKeyBytes);
KeyFactory keyFact = KeyFactory.getInstance("DH");
l_publicKey = keyFact.generatePublic(x509KeySpec);
```

Després es genera un *KeyAgreement* per tal d'inserir-li la clau pública (o les successives claus en cas de fer una multi sessió),.
S'instancia com DH, s'inicialitza amb la privateKey i es realitza el doPhase amb la o les diferents publicKeys dels diferents usuaris que intervinguin a la conversa.

Finalment es genera el secretKey amb l'algoritme TripleDES com algoritme amb que es codificarà.

```
// Prepara la generació del secretKey amb la privateKey i la publicKey de l'altre client implicat
KeyAgreement ka = KeyAgreement.getInstance("DH");
ka.init(privateKey);
ka.doPhase(l_publicKey, true);

// Tipus de clau a generar.
String algorithm = "TripleDES";

// Genera la clau secreta
secretKey = ka.generateSecret(algorithm);
```

Els mètodes getContacte () i addContacte () son els encarregats de tractar amb el fitxer XML d'agenda.

Per tal d'utilitzar-lo només cal instanciar la classe XML passant com a paràmetre la ruta on està o estarà el fitxer d'agenda i posteriorment utilitzar els mètodes de la classe:
Llegir_valors (String nomContacte) o inserir_node (String nomContacte, String cadenaValorsDH).

4.10 CYPHER.CLASS

Aquesta classe és l'encarregada d'utilitzar la potència del paquet javax.crypto que ofereix la classe Cipher.

Es generen dues instanciacions de la classe, una per codificar les dades i l'altre per descodificar-les.

L'algoritme de codificació ha de ser evidentment l'utilitzat per la generació de claus a la classe DecoderDh, el TripleDES al nostre cas.

Una instanciació de la classe s'utilitza per fer l'encrypt i l'altre pel decrypt.

```
ecipher = Cipher.getInstance("TripleDES");
dcipher = Cipher.getInstance("TripleDES");
ecipher.init(Cipher.ENCRYPT_MODE, key);
dcipher.init(Cipher.DECRYPT_MODE, key);
```

El mètode doFinal(byte[]) de la classe Cipher és el que realitza aquesta tasca de xifrar/desxifrar les dades, segons el MODE que assignem a l'init.

Per tant, i al nostre cas, ecipher.doFinal () codificarà o dcipher.doFinal () descodificarà la informació que li enviem.

Cal pensar que l'String que volem xifrar s'ha de passar a binari en format UTF8 , al desxifrar s'ha de passar a String amb el mateix format:

Codificació:

```
byte[] utf8 = str.getBytes("UTF8");

// Encrypt
byte[] enc = ecipher.doFinal(utf8);

// Encode bytes to base64 to get a string
return new sun.misc.BASE64Encoder().encode(enc);
```

Descodificació:

```
byte[] dec = new sun.misc.BASE64Decoder().decodeBuffer(str);

// Decrypt
byte[] utf8 = dcipher.doFinal(dec);

// Decode using utf-8
return new String(utf8, "UTF8");
```

4.11 XML.CLASS

Amb la classe XML podem accedir a un fitxer xml que ens permet de mantenir l'agenda de valors organitzada en format d'arbre amb els avantatges que això proporciona en quant a organització, fiabilitat i velocitat d'accés.

Per fer-ho s'ha utilitzat la llibreria JDOM que proporciona unes eines molt potents i senzilles d'utilitzar per tal de treballar amb XML.

Primer de tot es genera el parser *SaxBuilder*, i si el fitxer existeix, es genera el document parsejat amb que treballarem. En cas de no existir el crearem prèviament.

```
SAXBuilder builder=new SAXBuilder(false);
File fopen = new File (ruta+fitxer);
if (!fopen.exists()){
        creaFitxer ();
}

document = builder.build(ruta+fitxer);
llegir_arrel ();
```

Per tal de llegir els valors, el mètode llegir_valors (String ls_nomUsuari) realitza una iteració pels nodes que pengen de l'arrel (document.getRootElement()) i cerquem el node ls_nomUsuari.

Un cop obtingut recuperem els valors de les seves tres fulles, que es corresponen a cada un dels valors dels paràmetres de DH.

```
nodos = arrel.getChildren(node);
if (nodos.size() != 0){
        Iterator i = nodos.iterator();
        Element contacte = (Element)i.next();
        ls_Return =
        contacte.getChild("p").getValue()+","+contacte.getChild("g").getValue()+","+contacte.getChild("l").getVal
        ue();
}
```

El mètode Inserir_node (String ls_NomContacte, String ls_Valors) insereix un node ls_NomContacte, i les seves tres fulles, recuperant els valors de la llista de valors ls_Valors.

Es genera un no Element ls_NomContacte, i dins de cada un s'afegeix un nou Element, cada un dels quals te el valor d'un dels paràmetres de la llista de valors ls_Valors.

Finalment aquest element s'afegeix a l'arrel i es regenera el fitxer XML amb el nou node:

```
Element contacte = new Element(ls_Cont);
contacte.addContent (new Element("p").setText(ls_p));
contacte.addContent (new Element("g").setText(ls_g));
contacte.addContent (new Element("l").setText(ls_l));
arrel.addContent(contacte);
document.removeContent();
document.addContent(arrel);

 try{
    XMLOutputter out=new XMLOutputter();
    FileOutputStream file=new FileOutputStream(gruta+gfitxer);
    out.output(document,file);
    file.flush();
    file.close();
 }catch(Exception e){
```

5 – MANUAL D'INSTAL·LACIÓ

S'adjunten dos fitxers instal·lables l'un pel Servidor i l'altre pel Client.

Els fitxer es troben al directori adjuntat a l'entrega : "XatSegur – Instal·ladors"

Cal tenir amb compte que ja que no s'ha creat un fitxer de paràmetres per tal de no HardCodejar la IP del Servidor i el Port del mateix, la IP utilitzada és localhost i el Port és el 2001, per tant **s'instal·laran tant el Servidor com el Client a la mateixa màquina**.

El programa instal·lador demana la ruta a la que s'han de copiar els programes, i pel mateix motiu que al punt anterior, el directori és fix **: c:\XatSegur**

Un cop c:\XatSegur te copiats els arxius extrets del programa d'instal·lació, es procedirà a executar primer el Servidor, per tal que ofereixi els serveis, i posteriorment tots aquells clients que es requereixin.

Cal tenir instal·lada correctament, com a mínim, la **versió 1.4.2.08 de la J2RE** (versió amb que s'han realitzat les proves) amb les APIs de criptografia i els Parsers inclosos**,** per tal que accepti el treball amb els Parsers d'XML. En cas contrari l'aplicació no funciona.

Problemes amb la instal·lació del Parser

En tot cas, podria ser que es treballés amb una versió del J2RE que no tingués les llibreries del PARSER de XML, amb la qual cosa caldrà afegir al CLASSPATH la llibrería **org.apache.crimson.jaxp.DocumentBuilderFactoryImpl,** continguda a diversos paquets de classes**.**

Caldrà instal·lar algun paquet de PARSERs.

En cas de no ser possible aquesta instal·lació, s'afegeix un ClientXat.jar al directori del projecte "XatSegur – Instal·ladors/Versio sense Parser", per tal de modificar-lo pel ClientXat.jar que es genera al directori C:\XatSegur, al moment de la instal·lació.
Aquest nou fitxer permet treballar sense fer us del Parser d'XML i per tant sense poder utilitzar l'Agenda.

En aquest cas, caldrà executar el JAR (auto executable) ClientXat.jar, en comptes del executable generat per la instal·lació.

6 – JOC DE PROVES

Per tal de comprovar el correcte funcionament de l'eina, s'han estat realitzant proves tant des del punt de vista del Servidor com des del punt de vista del Client:

Proves al Servidor:

- Intentar arrencar el Servidor:
  El servidor arrenca correctament.

- Arrencar un altre cop el Servidor
  Es produeix un error de port ocupat.

- Arrencar el Servidor amb un Firewall que talli accessos a localhost.

Després de bloquejar amb un FireWall (ZoneAlarm) les connexions a localhost, intentem executar l'aplicació per tal de veure que passa.
Es produeix un error de restricció de seguretat.

- Arrencar el Servidor i esperar una connexió per part d'un client:
  Funciona correctament.

- Rebre una nova petició de connexió
  Provar que es poden connectar dos o més usuaris alhora.
  Funciona correctament.

- Deixar transcorre el temps d'inactivitat d'un client
  El client s'atura degut al temps de TIMEOUT.

- Comprovar que els events que es produeixen al Servidor es guarden correctament al fitxer d'historial
  Funciona correctament.

- Comprovar que si un usuari es penja per algun motiu, es desconnecten i es tanquen tots els buffers, sockets i demés
  Per tal de realitzar aquesta comprovació cal debugar l'aplicació, doncs sinó no podem veure que és el que fa realment el programa.
  Funciona correctament

Proves al Client:

- Arrencar un client amb el servidor engegat
  Connecta correctament

- Arrencar un client amb el servidor apagat
  Es produeix un error de connexió.

- Arrencar el client posant una IP o un port incorrectes
  Retorna una excepció de servidor no localitzat.

- Arrencar dues converses alhora
  Comprovar que realment un usuari pot mantenir alhora dues converses actives sense que això doni cap tipus de problema.
  Funciona correctament.

- Tancar una de les converses
  Es tanca la pantalla de la conversa, però no tanca les altres converses. Correcte.

- Tancar la finestra principal del client.
  Tanca totes les finestres que pengen de la pantalla inicial. Correcte.

- Comprovar el funcionament del generador de claus DH.
  Després de diversos problemes amb el pas de la clau pública, funciona correctament.
  S'han tingut molts problemes per passar la clau doncs estava codificada en format X509 i el programa intentava codificar-la a publicKey des del format String que no te el format X509 com un encoder correcte.
  Un cop solucionat aquest problema tot ha funcionat correctament

- Comprovar que es guarden correctament els valors a l'agenda i que els recupera
  Funciona correctament.

- Comprovació del xifratge/desxifratge dels missatges
  Després d'alguns problemes deguts a un error a la descodificació, funciona correctament.
  Amb missatges que sortien per pantalla es comprova que realment la codificació viatja xifrada per la xarxa.

- Comprovar que sense el fitxer d'agenda, la classe XML funciona correctament.
  Es llença una prova per veure si el fitxer XML és capaç de reaccionar correctament en el cas que es trobi sense el fitxer agenda.xml creat.
  El programa genera el fitxer i permet la continuació del fil d'execució.
  Correcte.

Finalment, es creen uns fitxers .jar per tal de fer les classes executables per si soles, i en aquest moment totes les proves funcionen correctament, amb l'excepció de la classe XML que penja el programa.
S'investiga el motiu i pot ser donat a que la classe te problemes amb funcionament amb Threads. Sigui com sigui es modifica la classe per treballar amb els Parsers propis de java, que no son tant potents ni macos, però donen el servei que es buscava.

7 – COMENTARIS I CONCLUSIONS

El programa de Xat segur proporciona les eines necessàries per tal d'assolir els objectius marcats prèviament, que son els següents:

- Proporcionar una interfície gràfica a l'usuari per tal que pugui realitzar les gestions de la seva conversa a cop de ratolí.
- Permetre l'usuari de mantenir diverses converses alhora.
- Generar les claus utilitzant el protocol Diffie-Hellman, i es guarden els valors públics a una agenda, per tal d'optimitzar el funcionament de l'aplicació.

A més, l'aplicació ha estat pensada per permetre que un usuari mantingui una conversa amb més d'un client alhora, i de fet es pretenia realitzar la modificació, però malauradament l'etern enemic, el temps, ha tornat a causar estralls.
Sigui com sigui, l'aplicació ha estat pensada per fer-ho tenint amb compte les llistes de connexions static que s'utilitzen, els objectes Conexio i Usuari, que permet treballar amb multiusuari, així com els mètodes d'enviament de cadenes de text i de cadenes de byte, que basen el seu funcionament a les llistes i no als usuaris.

També es pretenia utilitzar l'excel·lent potència de treballar amb XML per tal de treure tots els elements de configuració hardcodejats a un fitxer de configuració.
Elements com la IP a la que atacar, el PORT amb que fer-ho, el directori del historial o el directori de l'agenda eren possibles destinataris d'aquest XML.

Sigui com sigui, l'aplicació assoleix tots els objectius necessaris segons l'especificació del projecte, i ofereix eines que el poden potenciar, així com alguns elements extres com son el petit historial i l'agenda XML.

La conclusió de la feina, és que realment Java proporciona unes eines força senzilles i no per això poc potents, de treball amb aplicacions per xarxa, gràcies als sockets i el paquet **java.net**, de seguretat, gràcies al paquet **javax.crypto** i **java.security** i tot plegat motiven a qui treballa amb elles a voler seguir investigant quines son les meravelles d'aquests paquets així com de les constants evolucions que s'hi fan.

Com sempre la manca de temps han impedit al que escriu a motivar-se de manera extra i cercar tot el potencial que es pot assolir treballant amb sockets, claus privades, públiques...

8 – BIBILIOGRAFIA I RECURSOS

La bibliografia utilitzada certament s'ha reduït a la meravellosa eina de la nostre realitat temporal que és el Google, autèntica font de coneixement i mànega imprescindible i inesgotable font de coneixements que atorga a qui vol un sense fi de coneixements i nous mons, començant per es.wikipedia.org, les planes de sun com http://java.sun.com/webservices/jaxp/dist/1.1/docs/api/index.html, els amics de javahispano : www.javahispano.org , http://www.exampledepot.com/egs/javax.crypto/KeyAgree.html
O la ja clàssica www.programacion.com
I com oblidar-me de www.jdom.org.

A tots ells i moltes webs per les que hagi pogut passar a algun o altre moment:

Moltes gràcies.

9 – APÈNDIX 1

Tal i com es comenta al punt 6 - Joc de proves, al moment d'executar el programa independentment de l'eina amb que s'ha desenvolupat, l'Eclipse, l'accés a la classe XML per tal de recuperar les dades de l'agenda deixa penjat el programa.

Primer es pensava que podia ser donat a un tema d'execució amb Threads, però pensant-ho millor, quan s'executa el programa des de l'Eclipse funciona correctament, i l'execució és idèntica; mitjançant multi-Threadding.

Per tant es pensa que pot ser donat per un problema amb els Paths que Eclipse proporciona automàticament com a entorn de programació, per la qual cosa es procedeix a generar els .jar amb les llibreries JDOM incrustades i els PATHS afegits al fitxers MANIFEST.MF ... però ni així.

Tot resistint-me a acceptar no treballar amb la meravellosa llibreria del JDOM, s'intenten una sèrie de proves sense èxit, per la qual cosa finalment es genera una nova classe XML que utilitza els rudimentaris Parsers que Java porta incorporats, per poder utilitzar el fitxer d'agenda.

Les interfícies de crida son les mateixes per ser transparent a la classe que crida a XML (DecoderDH.class).

La nova classe XML.class funciona de la següent manera:

S'utilitzen DocumentBuilderFactory i DocumentBuilder per tal de parsejar el fitxer XML:

```
try {  DocumentBuilderFactory dbFactory = DocumentBuilderFactoryImpl.newInstance ( );
                DocumentBuilder docBuilder = dbFactory.newDocumentBuilder ( );
                xmlDoc = docBuilder.parse ( NOMBRE_ARCHIVO_XML );
                arrel = xmlDoc.getDocumentElement ();
```

Si el fitxer no existeix, és creat.

El mètode llegir_valors () s'encarrega de llegir els nodes del fitxer, utilitzant tant les classes NodeList com Node per anar accedint a cada Node/Fulla de l'XML, combinat amb funcions de cerca de la informació sobre els Strings que aquestes classes proporcionen a Java, realitzant un recorregut dels ítems.

Per tal d'inserir un node nou, es recrearà el fitxer XML prenent el anterior fitxer com un Element, i afegint-li al final el nou Node i les noves fulles, recreant de nou el fitxer XML.