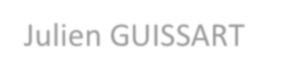
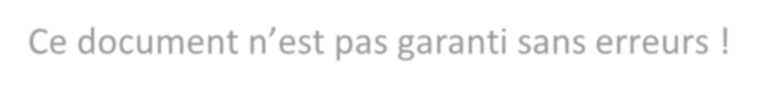
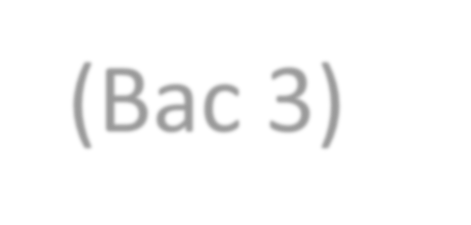
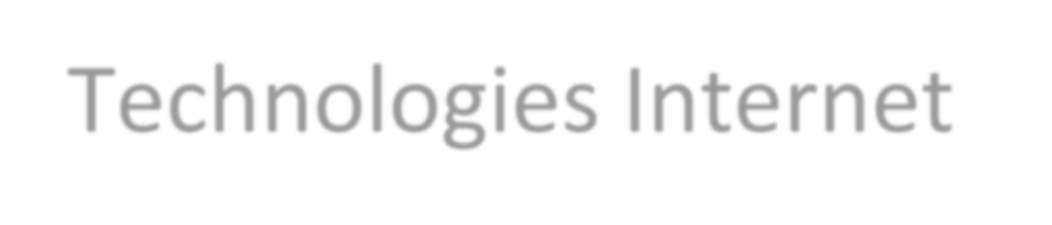
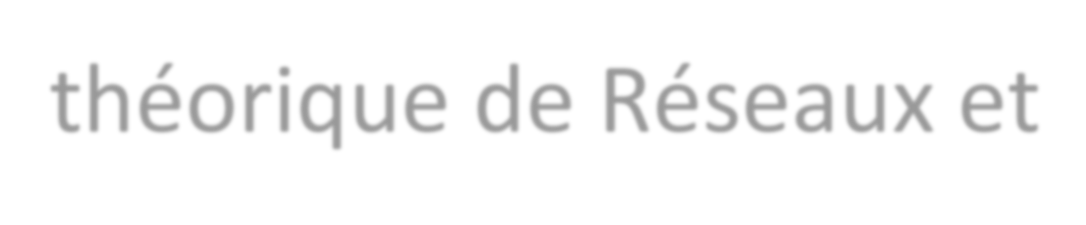
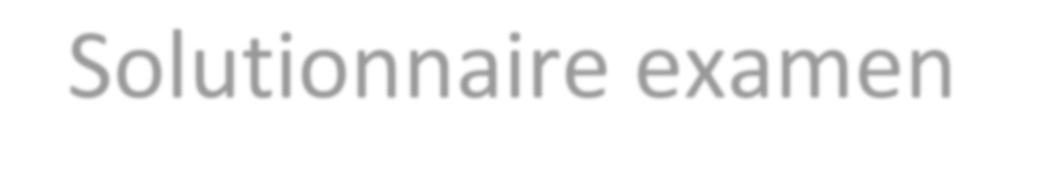
2015



Solutionnaire examen théorique de Réseaux et Technologies Internet (Bac 3)

Ce document n’est pas garanti sans erreurs !

Julien GUISSART

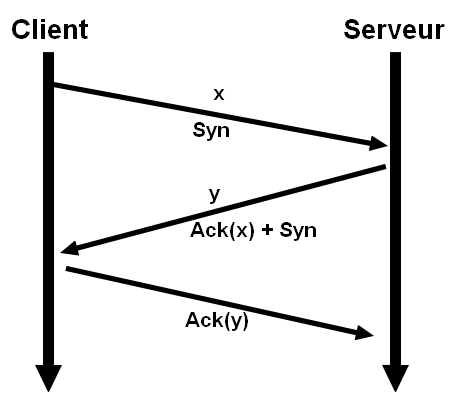
**Questionnaire Réseaux et Technologies Internet – Janvier/Septembre 2016**

# 1. Questions

## Expliquer, en vous aidant d’un schéma, le fonctionnement des primitives C socket(), bind(), listen(), connect() et accept() lors de la connexion d'un client TCP/IP à un serveur. En particulier, expliquer le rôle des queues de la structure socket, des flags SYN et ACK, de la "sliding window", du TTL. Que peut montrer la commande netstat et quel rapport avec les états de la machine à états de TCP ? Que se passe-t-il lors de la déconnexion de ce même client (états, flags, SYN et ACK) ? Expliquer en particulier les temps de latence qui sont explicitement prévus dans certaines transitions d’états.

### Phase de Handshake

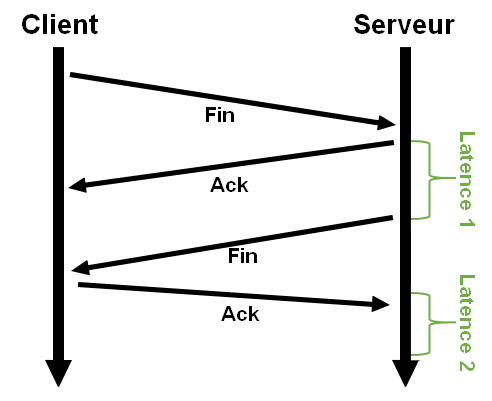
Avant toute possibilité de connexion, le serveur et le client doivent passer par une phase de reconnaissance : le handshake.Le handshake comprend l’échange de 3 segments (3-ways handshake).



* Établissement d'une connexion
* Le serveur se met à l'écoute sur un socket d'écoute (correspond à la fonction listen() mais il faut auparavant utiliser la fonction socket() pour initialiser la socket et un bind() qui permet de remplir l'IP et le port de la socket), on parle encore d'ouverture passive de la connexion.
* Le client se connecte (fonction connect() côté client), on parle encore d'ouverture active de la connexion. Il envoie alors un premier segment TCP de type SYN, ce qui permet de communiquer au serveur le premier numéro de séquence qui sera utilisée pour les données émises par le client (connexion dans l'état SYN\_SENT).
* Le serveur envoie au client un ACK et son propre SYN qui initialise les numéros de séquences des données qu'il enverra, la connexion est alors dans l'état SYN\_RCVD.
* Le client envoie un ACK au serveur, la connexion passe à l'état ESTABLISHED (cet état correspond en fait au transfert de données entre le serveur et le client). Une fois la connexion établie la fonction accept() crée une nouvelle socket (une copie de la socket d'écoute) qui permet de dialoguer avec le client (socket de service).

### Terminaison d’une connexion

Lors de la fermeture de la connexion, quatre segments sont échangés.



* Déconnexion du client
* Le client ferme la connexion (fonction close()), on parle encore de fermeture active de la connexion; il envoie alors un premier segment TCP de type FIN, la connexion est alors dans l'état FIN\_WAIT\_1.
* Le serveur reçoit le segment de fin et réalise une fermeture passive, la connexion est dans l'état CLOSE\_WAIT (connexion est à demi-fermée car des données peuvent encore transiter entre le client et le serveur).
* Le serveur envoie alors un ACK que le client doit recevoir, la connexion est dans l'état FIN\_WAIT\_2.
* Après un certain temps de temps de latence, la connexion est définitivement fermée par le serveur, il envoie alors un segment de TCP de type FIN, la connexion passe à l'état LAST\_ACK. Lorsque le client reçoit ce segment, il envoie un ACK, la connexion est dans l'état TIME\_WAIT (pour attendre que tous les paquets aient bien disparu).
* La connexion passe donc finalement dans l'état CLOSED.

### Sliding Windows

TCP utilise des « fenêtres glissantes » ou sliding windows, qui doivent leur nom au fait que des segments sont en fait envoyés alors que l’on n’a pas encore reçu l’ACK de certains segments précédents.

On peut distinguer :

* La fenêtre d’émission
  + Elle comporte les numéros de séquences déjà transmises mais non encore acquittées. Évidemment, sa taille est le nombre de buffers d’émission disponibles, puisqu’il faudra éventuellement réémettre ces segments.
* La fenêtre de réception
  + La taille de cette fenêtre est en fait le nombre de segments que le récepteur est capable de recevoir.

### Commande netstat

Cette commande permet d’observer l’état des connexions entrantes/sortantes au niveau du réseau. netstat –an | grep 3000

permet de voir les connexions qui touchent aux ports 3000x, avec les options :

* a : affiche toutes les connexions TCP actives et les ports TCP et UDP sur lesquels l’ordinateur écoute
* n : affiche les connexions TCP actives, cependant les adresses et les ports sont affichés au format

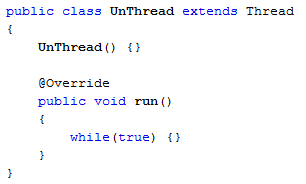
## Quelles sont les différentes manières de lancer et gérer un thread en Java ? Comment faire pour l'interrompre (de manière abrupte ou de manière plus contrôlée) ? Comment le problème des accès concurrents et celui de la synchronisation des threads sont-ils gérés en Java au moyen de monitors ? Quels sont les threads lancés automatiquement par une JVM Java ?

### Création d’un thread

Il existe 2 manières de créer un thread en Java :

#### Hériter de la classe Thread

Lorsqu’une classe hérite de la classe Thread (qui implémente l’interface Runnable), elle doit redéfinir la méthode run(). C’est dans cette méthode que se trouve en général une boucle qui sera l’objet du travail du thread.



Pour démarrer ce thread, on instanciera notre classe et on fera appel à la méthode start.



#### Créer un objet Runnable

Si une classe hérite déjà d’une autre classe, on ne peut évidemment pas la faire hériter à niveau de la classe Thread (pas d’héritage multiple en Java). La solution consiste à implémenter l’interface Runnable. Cette interface ne contient QU’UNE SEULE méthode : run().

On aura donc l’instanciation d’un objet Thread auquel on donnera à manger un objet de notre classe implémentant l’interface Runnable. Le thread exécutera alors, dans sa méthode run(), la méthode run() de notre classe.

### Synchronisation

Evidemment, qui dit thread dit concurrence, et qui dit concurrence dit ressources à protéger. En Java, il est possible de créer des sections critiques en déclarant des méthodes comme étant « synchronized ».

Lorsqu’un thread invoque une méthode synchronized d’un objet, l’objet est vérouillé : un autre thread qui invoque une méthode synchronized (la même ou une autre) sur le même objet restera bloqué jusqu’à ce que le verrou soit enlevé par la fin de la 1ere exécution de la méthode. On assure ainsi l’exclusion mutuelle.

En réalité, une méthode « synchronized » est ce que l’on appelle un « monitor ».

Les méthodes wait(), notify() et notifyAll() permettent de synchroniser différents threads. Ces méthodes sont définies dans la classe Object (car elles manipulent le verrou associé à un objet), mais ne doivent pas s’utiliser que dans des méthodes synchronized.

* Wait() : place le thread en attente d’une notification par un autre thread. Le moniteur est automatiquement libéré.
* Notify() : choisit un thread en attente sur le moniteur et le réveille, lui rendant automatiquement ce moniteur. S’il n’y a aucun thread en attente, la notification est perdue.

Un moniteur est réentrant, c’est-à-dire qu’il ne se bloque pas lui-même en appelant une seconde méthode synchronized.

On n’hérite pas de « synchronized ».

### Threads automatiques

Dans le cas d’une application utilisant un GUI, il existe un groupe de threads avec parmi eux le thread main (une simple instance de la classe Thread) et surtout un thread nommé AWT-EventQueue-0, instance de la classe EventDispatchThread (classe privée du package java.awt et qui hérite de Thread). Le rôle de ce thread est de gérer l’interface graphique et ceci de manière exclusive.

### Arrêt d’un thread

#### Avec une variable

Il est tout à fait possible d’arrêter un thread en boucle en sortant de cette boucle via un booléen. Problème : que faire si le thread est en attente sur une méthode bloquante ?

#### Avec un « interrupt »

Afin d’être plus propre, il est plus sage d’utiliser la méthode Thread.interrupt() ;

Si le thread est en attente, une exception (InterruptedException) est lancée, il suffit alors d’effectuer le traitement voulu pour arrêter le thread (mettre une variable à 0 par exemple).

Il existe aussi la méthode Thread.stop(), deprecated, qui est l’équivalent d’une utilisation d’un lance-roquette sur un lapin. A éviter donc.

## Quelles sont les caractéristiques des communications réseaux utilisant le protocole UDP ? Caractériser avec du code concret les grands points de la programmation UDP/IP en C/Unix et en Java. Expliquer avec du code concret pourquoi le paramétrage des sockets peut être nécessaire dans ce type de programmation UDP pour mettre en place un time-out ou pour utiliser une adresse multicast.

### Caractéristiques

UDP est un protocole non fiable et orienté sans connexion.

UDP envoie donc des datagrammes d’une machine à l’autre sans garantie de réception.

Conséquences :

* Complexité moindre
* Trafic réseau réduit
* Ne garantit plus l’ordre de réception des données ni leur longueur
* À n’utiliser que si la perte de paquet n’est pas impardonnable (par exemple, lors d’un chat avec quelqu’un d’inintéressant, comme Desart)
* Le fait d’être orienté sans connexion permet évidemment le multicast, c’est-à-dire le fait de viser tout un groupe d’adresses destinataires.

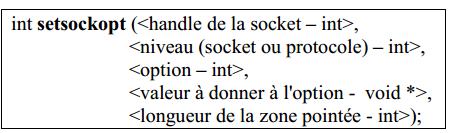
UDP simplifie aussi la programmation, puisque les applications désirant échanger des informations doivent simplement chacune :

* Créer une socket
* La faire reconnaitre par le système, du moins dans le cas du serveur
* Échanger les données (avec les précautions évoquées ci-dessus)
* Fermer leur socket
* Le code de la programmation UDP (C / Java)

|  |  |
| --- | --- |
| C | JAVA |
|  |  |

* Paramétrer une socket

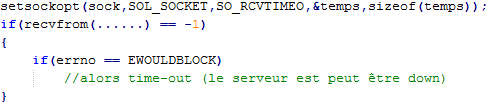
Il faut savoir que comme UDP n'est pas fiable, il se peut très bien que le serveur soit arrêté sans que le client puisse s'en rendre compte. Et donc le client restera bloqué sur un recvfrom() (cet erreur est appelé : erreur asynchrone). Ou même un datagramme peut être perdu, provoquant aussi un blocage sur le recvfrom().

* L'intérêt de paramétrer une socket est d'éviter se genre de blocage et on peut aussi faire du multicast grasse au paramétrage de socket.

La fonction en C qui permet de paramétrer une socket\_est :

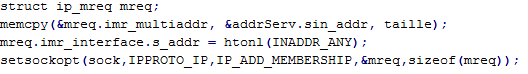
1. Time-out:

On suppose que la création de la socket et tout ce qui s'en suit à déjà été fait.



1. Le multicast:

Idem, On suppose que la création de la socket et tout ce qui s'en suit à déjà été fait.



Ici on joint le groupe multicast.

## Expliquer avec des éléments de code concret comment (principes, adresses et ports, code de base) on peut réaliser un multicast de type chat avec des participants Java et C. En particulier, quel est le rôle (implicite ou explicite) du paramétrage des sockets ?

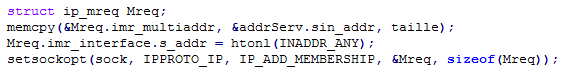
### Principe

Avec UDP, il est tout à fait possible de joindre tout un groupe de destinataire grâce à une adresse de groupe multicast. Lorsqu’on envoie un message à une telle adresse, les personnes intéressées (faisant partie du groupe multicast) seront notifiées.

Une adresse de multicast (classe D) est comprise entre 224.0.0.0 et 239.255.255.255. Afin de rejoindre un groupe multicast, une socket provenant du C doit être paramétrée :

* IP\_ADD\_MEMBERSHIP : rejoindre un groupe multicast
* IP\_DROP\_MEMBERSHIP : quitter un groupe multicast
* IP\_MULTICAST\_TTL : spécifier le TTL d’un message multicast sortant

### Joindre un groupe en C



* 1. **Joindre un groupe en Java**

Le package java.net fournit une classe MulticastSocket.

Le constructeur



réalise le bind sur le port spécifié.

Pour se joindre à un groupe multicast, on utilisera la méthode



dont le paramètre permet de préciser l’adresse (de classe D) du groupe multicast considéré. Exemple concret :



Pour recevoir un message, on réalisera (dans un thread) :



## Quelle est la démarche à suivre pour atteindre une base de données au moyen de JDBC et y pratiquer des mises à jour (dynamiques ou non) ? En quoi peut-on affirmer que l'on atteint un certain niveau de généricité et d'abstraction ? Comment peut-on réaliser les opérations de parcours aléatoire, d'ajout ou de suppression, de mise à jour ? Expliquer globalement ce que JPA apporte de plus en termes de renforcement de l'abstraction.

### Principe

JDBC est un ensemble d’APIs permettant d’accéder, au moyen d’instructions SQL, aux informations stockées dans une base de données.

La plupart des SGBD courants sont gérés, si bien que l’on peut considérer que le code écrit avec JDBC est indépendant de ceux-ci (généricité).

Le driver JDBC fera le pont entre Java et la BD visée. Ces drivers sont catégorisés selon 4 types : le type 1 étant écrit en C et le 4 en full Java.

#### Chargement en mémoire

Le chargement de la classe du pilote se fait en utilisant une méthode de la classe Class. Cette classe sert à représenter l’ensemble des renseignements sur une classe, comme son nom ou ses différentes méthodes. C’est la méthode forName(String className) qui va nous intéresser. La méthode, au moment de l’exécution, va localiser la déclaration de la classe, en charger la définition (donc aussi ses blocs statiques qui constituent le driver) et ajouter cette définition à l’environnement d’exécution de l’application courante = introspection.

On installe donc en mémoire le décor nécessaire au bon déroulement des opérations. Attention : pas d’instanciation !

On aura donc



#### Connexion à la base de données

La classe DriverManager et sa méthode getConnection(‘urlBD’, ‘login’, ‘password’) va aller chercher le driver approprié et, en cas de succès, fournir un objet qui implémente l’interface Connection, matérialisant la connexion demandée.

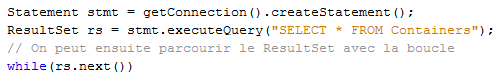
Ce DriverManager est un bel exemple de généricité puisqu’il réalise son choix parmi les drivers disponibles.

#### Requêtes SQL

On se retrouve donc avec un objet implémentant l’interface Connection qui va nous permettre de créer des requêtes SQL.

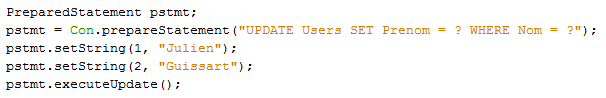
Pour cela, on s’offre un objet Statement qui possède les méthodes executeQuery(), execute() et executeUpdate().

Requête statique :



Requête dynamique :

On utilise alors un objet PreparedStatement :



### Curseurs

Il est possible d’obtenir des curseurs plus souples, en ce sens que l’on peut demander :

* Un type de parcours quelconque (donc pas forcément séquentiel)
* La possibilité de modifier un tuple traité (donc, le curseur n’est plus en read-only)

Ceci peut se préciser en utilisant une version polymorphe de la méthode createStatement() que possède tout objet implémentant l’interface Connection :

Le 1er paramètre détermine le type de déplacement autorisé :

* TYPE\_FORWARD\_ONLY : séquentiel uniquement
* TYPE\_SCROLL\_SENSITIVE : quelconque (les changements apportés par d’autres opérations sont pris en compte)

Le 2e détermine si le curseur est read-only ou read-write :

* CONCUR\_READ\_ONLY : curseur read-only
* CONCUR\_UPDATABLE : curseur read-write

### Apports de JPA

* Les entités sont créées comme de simples classes
* Les techniques d’interrogations, de concurrences et de transactions sont disponibles
* Les techniques classiques de la POO sont disponibles (héritage, polymorphisme, …)

Il permet donc un accès simplifié aux données (il n’y a plus de SQL) en élevant le niveau d’abstraction dans le code (injections, entités, …)

## Expliquer la structure et le fonctionnement d'un serveur multithread générique utilisant un pool de threads et des objets Runnable. Comment procéder pour qu'il puisse gérer des conversations prolongées entre le serveur et les clients ? Comment faire pour que ce serveur générique ne risque pas d'exécuter n'importe quelle commande, quelles procédures d'authentification peut-on envisager ?

Le serveur multithread générique lancera un thread (ThreadServeur) chargé de gérer les connexions réseaux (afin de ne pas bloquer le GUI). Ce thread :

* Se mettra à l’écoute sur un port donné
* Lancera au préalable les threads (ThreadClient) chargés de gérer les requêtes d’un client
* À chaque connexion, lira la requête sous forme d’objet sérialisé
* La requête sera mise dans une liste de tâches consultable par les threads

Afin de rendre ce serveur le plus générique possible, il ne sera pas « lié » à des requêtes de façon immuable. Les requêtes seront sauvegardées dans une liste de tâches (liste d’attente) mais seront sauvegardées en tant que Runnable. On sauvegarde donc uniquement ce que le thread « doit faire ». le thread exécutera alors la méthode run() de ce Runnable, il n’a ainsi pas besoin de savoir ce qu’il va faire.

### Quelques interfaces utiles :

#### SourceTaches

Interface comprenant les méthodes Runnable getTache() et void recordTache(), un tel objet contiendra les tâches à exécuter sous forme d’objets Runnable et permettra de mémoriser/récupérer ces tâches.

#### Requete

Interface ne comprenant que la méthode public Runnable createRunnable(Socket, ConsoleServeur). Selon notre logique de généricité, une requête doit pouvoir générer l’objet Runnable qui la traite. Les requêtes acceptées par notre serveur devront implémenter cette interface.

#### Reponse

Interface comprenant l’unique méthode getCode() qui retourne le code correspondant à la réponse qui sera contenue dans l’objet implémentant cette interface et donc le résultat.

### Les threads

#### ThreadServeur

Ce thread lancé au démarrage de l’application, il crée la socket serveur et lance les différents threads clients de son pool puis se met en attente d’une connexion client. Une fois celle-ci reçue, il attend un objet Requete. Dés réception, il crée un objet Runnable sur base de celui-ci en appelant sa fonction createRunnable() et enfile celui-ci dans sa LinkedList.

#### ThreadClient

Chaque thread client du pool du serveur est en attente d’une tâche à exécuter. Le thread client est celui qui communiquera réellement avec l’application de l’autre côté de la connexion.

### Le protocole

Le protocole est la seule chose qui devra être modifiée de serveur générique en serveur générique. Il définira des constantes pour chacune de ses requêtes, il devra également prévoir un accès aux data (ex : un bean d’accès aux BD), il implémentera évidemment l’interface Requete afin de créer des objets Runnable à la volée. Un protocole RequeteMonProtocole implique évidemment une classe ReponseMonProtocole. Ce protocole devra également implémenter l’interface Serializable afin de pouvoir communiquer via réseau.

Afin de pouvoir garder une conversation prolongée entre un client et un serveur, il faut absolument éviter de passer, pour CHAQUE requête, par la liste de tâches. Le but étant de garder une connexion ouverte entre un ThreadClient et une application. On laisse alors le ThreadServeur réaliser uniquement un travail lors de la création du premier Runnable.

Une solution est de créer une boucle dans le premier Runnable envoyé. On peut en effet imaginer une requête de type « Connexion » qui réaliserait un premier « démarrage » de Runnable pour ensuite se mettre en attendre sur d’autres Runnable.

## Qu'est-ce qu'un "moteur à servlets" et un "compilateur de pages" dans le contexte de la programmation Web en Java ? Décrire, à partir des répertoires et des différents tags des fichiers de configuration server.xml et web.xml de Tomcat, comment celui-ci réagit quand le client Web lui demande l'exécution d'une servlet (identification de la classe servlet, instanciation éventuelle, threads, etc). Même question mais pour l'exécution d'un JSP.

Un **moteur à servlet** est une application permettant le test et le déploiement des servlets.

On parle encore aussi de container de servlets. Son rôle est de fournir l'environnement d'exécution des servlets

* en réalisant la communication entre le serveur Web et les servlets,
* en gérant le chargement et l'instanciation des servlets ainsi que les invocations de leurs méthodes.

Le **compilateur de pages** du moteur JSP va transformer la page JSP en une servlet Java servlet.

Les éventuelles méthodes définies dans les scriptlets deviennent alors des méthodes de cette servlet.

Tomcat est un moteur à servlets et JSPs développé par Sun. Il peut fonctionner :

* De manière autonome (« standalone »), de cette façon, il joue le rôle d’un serveur HTTP
* De manière plus classique, en étant ajouté à un serveur Web frontal comme Apache

Tomcat est en réalité constitué d’un ensemble de composants dont le rôle est de traiter les requêtes envoyées par des clients (via le protocole HTTP). Les composants principaux (containers) vont en contenir d’autres. Leurs noms sont : Server, Service, Engine, Host et Context. Ces composants sont décrits très précisément dans un fichier de configuration nommé server.xml situé dans le répertoire « conf » de Tomcat.

### Architecture de server.xml

La balise la plus extérieure est <Server>, qui précise essentiellement sur quel port Tomcat attend un shutdown. Cette balise peut contenir un ou plusieurs « services », identifiés par une balise <Service>, qui comporte chacune un ou plusieurs connecteurs (balise <Connector>) qui utilisent le même moteur (engine).

Par « connecteur », on entend un point d’entrée, caractérisée par un port réseau, sur lequel une requête peut être reçue et par lequel une réponse peut être envoyée.

En réalité, la requête sera transmise à une servlet tournant dans le container associé au connecteur.

On peut définir des « engines », qui constituent des points d’entrée pour les requêtes HTTP (arrivant par les connecteurs) déterminant l’hôte concerné par cette requête. Ainsi, le moteur à servlet Catalina de Tomcat correspond à l’engine :



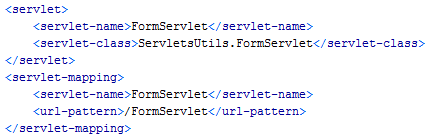
Le point essentiel, du point de vue développement, est cependant la balise <Context> que l’on trouve au sein d’une balise <Host>. Cette balise <Context> représente en effet tout simplement la description d’une application Web.

### Descripteur de déploiement web.xml

Le déploiement d’une servlet au sein d’une application Web est décrit par un fichier web.xml, encore appelé le

« descripteur de déploiement » de la servlet. Plus précisément, les éléments XML d’un tel descripteur sont définit en détails dans le document de spécification des servlets. Ils doivent se succéder, quand ils sont présents, dans un ordre bien précis :

Le point fondamental, c’est-à-dire les diverses servlets et leur chemin d’accès :



Le tag <servlet> : il existe donc dans l’application une servlet instance de la classe FormServlet (du package ServletsUtils). Elle sera désignée sous le nom de FormServlet.

Le tag <servlet-mapping> : la servlet connue sous le nom FormServlet sera démarrée par l’intermédiaire d’un browser utilisant le protocole HTTP au moyen de l’URL du serveur complétée par « /FormServlet ».

## Quelles sont les principales caractéristiques du protocole HTTP ? A quoi ressemble classiquement un dialogue client-serveur en termes de trames (commandes, headers divers) ? Décrire, comme le ferait un sniffer, la trame HTTP classique encapsulant une page HTML demandée par un user-agent; en particulier, pourquoi parle-t-on de « négociation » et de « facteur q » ? Expliquer comment le problème de l'authentification est-il pris en charge par HTTP (Basic et Digest).

* 1. **Protocole**

Le protocole HTTP (HyperText Transfer Protocol) est un protocole de la couche application. Il est fortement orienté TYPAGE et NEGOCIATION.

Typage = « se tracasser de la nature de ce qu’il transporte ». Il peut y avoir beaucoup de types différents. Grâce à MIME, il est possible d’identifier les types dans une trame HTTP (M de MIME = mail. HTTP s’est inspiré des protocoles de mails).

Négociation = le client contact un serveur mais il donne ses préférences. Exemples : « je veux telle ou telle forme pour mon renseignement ». Le serveur lui répond en fonction de la demande.

Par défaut, HTTP n’a pas d’état. Il est évidement basé TCP. HTTP est un protocole sans session.

Un client (ou « user-agent » qui peut être un browser, un éditeur, …) prend donc l’initiative de la communication en envoyant une requête au serveur.

Cette requête comporte

* La méthode utilisée pour cette requête (GET ou POST)
* L’URI de la ressource demandée
* La version du protocole utilisé
* Un message MIME contenant diverse informations

Le serveur répondra par un message précisant

* La version retenue du protocole
* Un code de succès ou d’erreur
* Un message MIME donnant des informations sur le serveur

### Forme générale des messages

Un message HTTP sera toujours une requête du client au serveur ou une réponse du serveur au client. Le header comporte un certain nombre de champs (éventuellement aucun).

<message HTTP> = <ligne de démarrage ou « start-line »>

<header général du message> CR LF

[<corps du message>] Avec CR = carriage return et LF = Line feed

### Forme d’une requête

<requête> = <méthode> SP <URI de la ressource considérée> SP <version HTTP>

<header général> | <header de la requête> | <header d’entité> CR LF

[<corps du message>]

La méthode peut être POST, GET, OPTIONS, PUT, …

GET : cette méthode permet de retrouver une information de nature quelconque identifiée par son URL. Dans le cas particulier où la ressource est en fait un service, ce sont les données produites par l’exécution de ce service qui constitueront la réponse.

POST : il s’agit de demander que le serveur accepte les données contenues dans le message comme constituants d’une requête pour la ressource-service dont l’URI est précisée (un usage classique : envoi de données d’un formulaire pour effectuer une recherche dans une base de données).

Le header de requête

C’est dans le header de requête que le client peut transmettre des informations le concernant ou concernant sa requête.

Un exemple classique est le champ « Accept ». Ce champ de header de requête permet de spécifier les types de média qui pourront être utilisés dans la réponse, évitant ainsi une réponse incompréhensible par le client qui a formulé la requête.

L’accept peut préciser un paramètre d’acceptation q (comme « quality ») qui permet de spécifier ses préférences sur une échelle allant de 0 (= « s’il n’y a vraiment rien d’autre ») à 1 (= « mon préféré entre tous »).

Exemple :

Accept : text/plain ; q=0.5, text/html, text/x-dvi ; q=0.8

Signifie « je préfère la version HTML, mais si c’est impossible, alors je préfère la version en x-dvi ». A remarquer que l’absence de clause q signifie implicitement que q vaut 1.

### Forme d’une réponse

<réponse> = <version HTTP> SP <Code> SP <texte information>

<header général> | <header de réponse> | <header d’entité> CR LF

[<corps du message – par exemple la page HTML demandée>] Avec SP un espace.

Les headers de réponse sont moins importants que les headers de requêtes.

Le code envoyé par le serveur est un entier à 3 chiffres lui permettant de signifier dans quelle mesure il peut donner suite à la requête reçue. Ces codes sont répartis en 5 groupes :

1XX : Simple information (le traitement de la requête se poursuit) 2XX : Succès (la requête a été comprise et acceptée)

3XX : Redirection (une autre action doit être entreprise par le user-agent pour compléter la réponse) 4XX : Erreur côté client (soit la syntaxe, soit dans la nature même de la demande)

5XX : Erreur côté serveur (la requête ne peut pas être satisfaite)

### Vue du trafic

Supposons vouloir charger une page HTML qui se trouve sur un serveur quelconque. Si on place un sniffer pour visualiser les paquets, on peut directement remarquer :

* Qu’il y a une phase de connexion (normal, on utilise TCP comme protocole de transport)
* Que la transaction HTTP réclame 3 paquets TCP
  + Le 1er comporte les données (donc la requête proprement dite)
  + Le 2nd est la réponse du serveur tout en acquittant la requête
* Que la connexion TCP est ensuite fermée (four-way handshake)

### Authentification

#### Basique

Cette méthode est la plus simple, mais également la moins sécurisée car elle transmet le mot de passe en clair (ou presque). Elle est donc recommandée avec une connexion chiffrée (HTTPS).

Le serveur ne recevant pas d’en-tête d’identification correcte envoie ce genre d’en-tête à HTTP : WWW-Authenticate : Basic realm="MonRoyaume"

Le serveur indique la méthode requise (Basic), suivie des paramètres (la méthode Basic ne requiert que le paramètre realm identifiant le domaine de protection).

Le client peut alors réessayer la requête en spécifiant l’en-tête HTTP « Authorization ». Celui-ci doit contenir la méthode (Basic) suivi de la représentation en Base64 du nom et du mot de passe, par exemple :

Authorization : Basic ADRIENADMIS

#### Via Digest

Message digest = « Donnez-moi un message de longueur variable, et j’y applique un algorithme pour ne pas dépasser une longueur fixe ». (+- semblable à la définition d’un hashage).

Un digest ne crypte pas (on ne sait pas revenir dans le sens inverse) et si nous avons 2 messages différents, il faudra un hashage différent !

Le but étant de comparer les résultats des 2 hashages (de chaque côté de la transaction). S’ils sont les mêmes, tout vas bien.

##### ATTENTION : Digest = CONTRÔLE D’INTEGRITE

Le digest n’est PAS un outil d’authentification SAUF si le digest est construit sur un champ d’authentification (comme par exemple un mot de passe).

## Dans le contexte d'un serveur WEB, qu'est-ce qu'un CGI et qu'est-ce qu'une servlet ? Comparer ces deux technologies par rapport au traitement des formulaires classiques. A quoi ressemble le code d'un CGI basique qui répondrait à une simple sollicitation au moyen d'une page formulaire ? Expliquer à quoi ressemble le code de la servlet classique correspondante à ce CGI (packages, cycle de vie, principales variables et fonctions membres des classes usuelles, paramètres de ces dernières).

### CGI

Un CGI (Common Gateway Interface) est un programme qui tourne sur le serveur Web et qui a été lancé à partir d’une requête en provenance d’une interface de navigation client. Les CGIs sont habituellement écrit en C/C++ et sont compilés pour fournir un exécutable non-portable.

### Servlet

Les servlets sont des unités de code (en gros, des classes) Java qui étendent les possibilités des serveurs Web. Ceux-ci ont donc la même fonctionnalité que les CGIs mais permettent une meilleure solution en apportant des avantages :

#### Portabilité

Étant écrite en Java, elles sont exécutées sur une machine virtuelle se trouvant sur le serveur. Son exécution est donc indépendante de l’environnement du serveur.

#### Instanciation unique

L’objet servlet n’est instancié qu’une seule fois (au démarrage de celle-ci), et non à chaque demande d’un client (contrairement à une CGI). Elle peut donc conserver ses données.

Chaque demande d’un client génère un thread qui s’occupera de cette requête et tous ces threads travaillent sur le même objet (servlet). L’objet servlet devient donc un objet monitor (attention aux synchronized donc).

#### Sécurité

Etant dans un mécanisme Java, les sécurités Java peuvent être mise en place.

### Comparaison

Une fois l’utilisateur ayant validé les données introduites dans un formulaire, celle-ci sont véhiculées, dans un cas simple, dans l’URL.

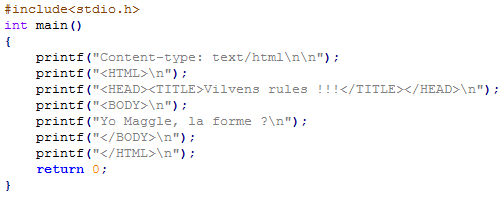
Dans le cas d’une servlet, celles-ci seront récupérées simplement grâce à la méthode getParameter(String nomParametre) de la classe HttpServletRequest qui a pour but de nous donner accès au header de la requête client mais surtout à ses paramètres. Cette méthode nous renvoie donc le paramètre indiqué en paramètre en NULL si celui-ci n’existe pas. La servlet réalise alors un traitement X et retourne une réponse au client sous forme de page HTML par exemple. Cette page sera donc générée dynamiquement par la servlet.

Dans le cas d’un CGI, le serveur, une fois une requête reçue, analysera tout d’abord l’en-tête HTTP pour essayer de lancer le programme approprié (script, …). Ce programme récupèrera les valeurs du formulaire sous forme de variables d’environnement. Il réalisera un traitement et écrira dans un premier temps le type de réponse dans l’en-tête HTTP tout comme la servlet ainsi que la page HTML (tag) proprement dite.

Ici, la comparaison est au niveau du passage des informations. Étant donné que le CGI est un programme

« externe », le seul moyen qui a été adopté pour lui faire prendre connaissance de ces informations est l’utilisation des variables d’environnement.

Code d’un CGI :



### Arborescence d’une servlet

Les classes/interfaces nécessaires se trouvent dans les JDK récents de J2SE/J2EE (package javax.servlet). Il existe :

#### Des interfaces

Servlet déclarant les méthodes de gestion des servlets (+ communication avec le client).

ServletRequest et ServletResponse correspondant respectivement au traitement d’une requête et à l’envoi d’une réponse.

#### Des classes abstraites

GenericServlet implémente l’interface Servlet et est indépendante du protocole de communication (nous dériverons donc de cette classe pour les servlets spécifiques telle que HttpServlet).

ServletInputStream et ServletOutputStream fournissent les flux de communication avec le client.

### Cycle de vie d’un servlet

#### L’initialisation

Le byte code de la servlet doit être chargé en mémoire à l’initiative du serveur, ce dernier exécute alors la méthode void init(ServletConfig config). C’est dans cette méthode que les connexions JDBC vers des bases de données seront mises en place, par exemple. En effet, ce genre de connexion ne se fait qu’une seule fois, et l’appel de cette méthode ne sera fait qu’une fois. Le serveur ne pourra recharger une servlet que si cette dernière a été au préalable déchargée au moyen de la méthode destroy().

#### Traitement des requêtes des clients

La servlet utilise la méthode void service(ServletRequest req, ServletResponse res) qui a pour but de matérialiser le mécanisme de requête/réponse en acceptant deux objets représentant la requête entrante du client et la réponse sortante qui lui sera renvoyée. Dans le cadre d’une servlet dédiée à un protocole de communication, ce sera cette méthode qui dirigera vers les méthodes appropriées (dans le cas HTTP, vers des méthodes dédicacées aux requêtes POST, ou GET).

#### Terminaison d’une servlet

Une servlet sera déchargée par le serveur par la méthode void destroy() qui réalise le travail inverse de la méthode init(), donc, par exemple, fermer la connexion JDBC vers la base de données utilisées par la servlet. À remarquer que le serveur devrait, sauf cas d’urgence impérative, s’assurer que les services en cours sont terminés ou, du moins, admettre un temps de latence suffisant. Il peut être utile, pour cela, de compter les requêtes en cours de traitement et d’en tenir compte dans la méthode destroy() redéfinie.

## Comment le problème que HTTP est un protocole "sans état" est-il résolu par les applications Web Java basées sur les servlets, notamment dans un contexte d'e-commerce où plusieurs clients réalisent chacun plusieurs requêtes ? En particulier, en quoi ces éléments suivants interviennent-ils: synchronized, cookies, réécriture d'URL, threads implicites, session ?

Le protocole HTTP est un protocole sans état. Une fois la transaction terminée, aucune information ne permet d’identifier le client lors des transactions futures.

Il est du coup impossible de parler de « session », ce qui est très embêtant pour une application e-commerce avec un panier par exemple. Avec plusieurs clients, cela devient le bordel total puisqu’on ne sait plus qui est qui …

### Différents moyens

#### Les cookies

Un cookie est une information (plus précisément un fichier texte) envoyée à un browser par un server qui vient d’être accédé afin que cette information soit sauvegardée sur le client.

Chaque fois que le browser s’adressera à ce serveur, il lui enverra le contenu du cookie. De cette manière, un serveur peut, dans le cas qui nous occupe ici, identifier de manière unique un client. Il existe une classe Cookie dont le constructeur est Cookie(String name, String value).

Les paramètres HttpServletRequest et HttpServletResponse (les cookies sont supportés par HTTP) possèdent des méthodes de lecture et d’écriture des cookies :



La méthode de récupération envoie un tableau que l’on peut ensuite parcourir. On peut ainsi récupérer, pour chaque cookie, son nom et sa valeur



#### La réécriture d’URL

Cette technique a pour principe de compléter l’URL de la servlet d’une information complémentaire que l’on désire transmettre.

Par exemple, dans ce cas-ci, un identificateur de session. Cette information se trouve entre le path et la query- string, c’est-à-dire entre l’URL proprement dite et la chaîne des paramètres du formulaire commençant par

« ? ». Elle est récupérable au moyen de la méthode de HttpServletRequest :



Si on ne récupère rien, c’est qu’il faut créer un nouveau numéro de session. Sinon, on se contente de celui récupéré.

#### Le suivi de session

Le JDK fournit un « API Session Tracking » qui permet de gérer une « session HTTP » au moyen d’une interface HttpSession. Puisque HttpSession n’est qu’une interface, l’objet qui l’implémente provient de HttpServletRequest, on y trouve donc la méthode



Le rôle de l’objet Session est de conserver des informations concernant l’interaction entre le serveur et le client. Ces informations peuvent être placées dans l’objet Session par les méthodes de HttpSession :



On récupère ces informations via :



Le cycle de vie d’une session :

Une session peut expirer

* Automatiquement après un certain temps d’inactivité dont la valeur est fixée par le serveur
* Explicitement par l’utilisation de la méthode void invalidate()

## Expliquer l'objectif, les diverses syntaxes (en les illustrant par un exemple) et le mécanisme des Java Server Pages (que se passe-t-il exactement quand un client accède à un JSP au moyen de son browser ?). Pourquoi le mécanisme des balises à Java Beans est-il vite ressenti comme nécessaire ? Expliquer comment fonctionne ce mécanisme des balises pour Java Beans à partir d'un exemple complet. Comment utiliser de telles balises pour réaliser une internationalisation de l'application ?

### Présentation

Les JSP (Java Server Page) permettent de créer dynamique une partie de page HTML stockée sur le serveur, le reste de la page étant statique.

La zone dynamique est codée en Java et générée du côté serveur (contrairement à un langage de script qui est généré du côté client). Nous nous rapprochons donc du fonctionnement du PHP avec en plus, grâce au Java, la portabilité, la sécurité et la possibilité d’y insérer des JavaBeans. Les JSP sont de plus multithread.

Au final, une JSP est donc une page HTML avec des éléments de Java intégrés. Notons qu’une JSP n’est qu’une servlet déguisée (et on fait tourner les servlets … Humour drôle). Le compilateur de page transformera notre JSP en servlet (le compilateur étant une servlet LUI-AUSSI, faisant partie de Tomcat).

Ces éléments de Java s’intègrent dans la page HTML via :

#### Des scriptlets

Ces zones sont déterminées par les tags <% et %>

#### Des directives

Elles définissent des attributs de la future page, attributs qui resteront constants pour toutes les requêtes. Elles sont déterminées par les tags <%@ et %>

Ex : <%@page contentType=’text/html’ %> précise le type MIME de la réponse qui va être générée et envoyée par la JSP.

#### Des déclarations de variables (ou de méthodes)

Membre de la future servlet résultat de la compilation de la page JSP, elles sont déterminées par les tags <% ! et %>

Ex : <%i int cpt = 0 ; %>

#### Des actions

Elles sont délimitées par des tags spécifiques comme <%jsp : …> pour gérer par exemple les beans et les plugins.

### Objectif

Etant donné que l’on retrouve le système de balise, il sera plus simple de coupler le travail entre graphistes et programmeurs. En effet, plus de dégueulasseries réalisées à coup de System.out.println().

### Mécanisme

Le code JSP va être compilé par le moteur à servlets en une servlet et c’est cette servlet qui sera, en définitive (on ne sait pas ce qu’elle devient, en définitif), compilée puis exécutée : la page résultante sera envoyée au client, comme d’habitude, comme une page statique normale. Les méthodes et variables déclarées dans un bloc de déclaration deviendront alors des méthodes de cette servlet.

### Les balises pour JavaBeans

L’objectif des JSP est bel et bien de séparer la présentation de la logique fonctionnelle … donc pourquoi ne pas confier cette logique à des éléments comme les JavaBeans ?

Les JSP fournissent des balises particulières qui sont des « actions » permettant de spécifier l’utilisation d’un élément extérieur (un bean, un plugin, une autre JSP) et/ou d’effectuer une tâche selon des informations obtenues au moment où le client réalise un accès à la JSP.

Lorsqu’on voudra utiliser un bean, on utilisera cette balise :

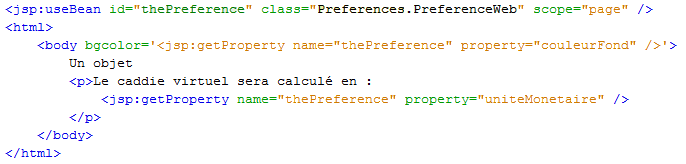


La portée étant la durée de vie du composant (par défaut = la page).

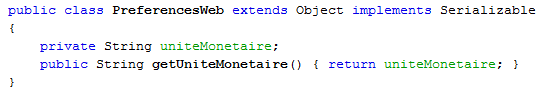
Les valeurs possibles de cette portée sont : page, request, session, application.

#### Exemple

JSP :



Bean :



1. **Décrire le modèle MVC dans le contexte des servlets et des Java Server Pages. Quels sont les principes et méthodes (paramètres, codes, tags, …) à mettre en œuvre pour l'implémenter ? Décrire le protocole applicatif et son implémentation dans le cas classique d'un caddie virtuel. Donner une structure de base du code (principales variables membres, logique de base, transfert de pages, ...).**
   1. **Modèle-Vue-Contrôleur**

Modèle : ce rôle est assuré par les beans d’accès aux données. Ces beans font le pont entre le contrôleur (les servlets) et le SGBD.

Vue : le rôle est assuré par les JSPs. En effet, c’est grâce aux pages créées par les JSP que la magie opèrera face au client.

Contrôleur : rôle assuré par les servlets. La servlet va donc constituer le point de passage obligé entre deux JSPs successifs. Autrement dit, le client ne passera jamais directement d’un JSP à un autre.

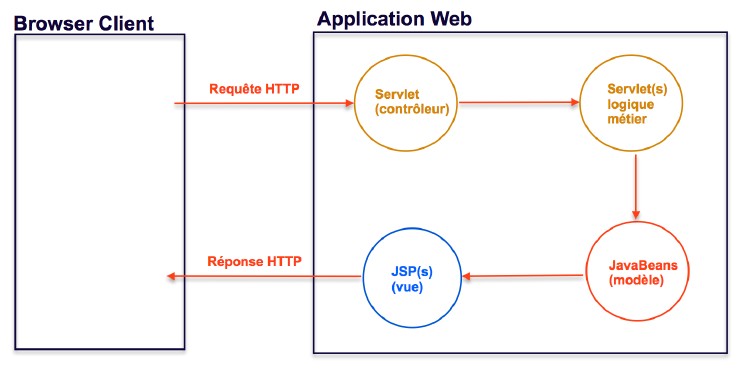
### Fonctionnement

La servlet est un point de passage obligé entre plusieurs JSPs. Ainsi un client ne pourra pas passer d’une JSP à l’autre, ce sera une servlet (le contrôleur) qui redirigera le client.

Pour ce faire, les JSPs invoqueront une servlet avec un tag permettant de faire comprendre à la servlet le travail demandé. La servlet pourra alors rediriger le client vers la JSP concernée grâce à la méthode void sendRedirect(String location) de la classe HttpServletResponse. Cette méthode permet de renvoyer le client vers une ressource dont le chemin correspond au paramètre « location ».

Une autre façon de faire est d’utiliser la méthode RequestDispatcher getRequestDispatcher(String path) de la classe ServletContext (obtenue par la méthode getServletContext sur l’objet Servlet). L’objet RequestDispatcher obtenu n’est pas la ressource en elle-même mais une enveloppe de celle-ci contenant sa référence. La méthode void forward(ServletRequest request, ServletResponse response) charge la ressource dans le navigateur internet du client.

### Schéma du modèle MVC



1. **Expliquer l'évolution historique de la notion de clé de chiffrement. En particulier, expliquer les termes de « substitution de caractères »,**

**« transposition », « chiffrements poly-alphabétiques », « grille de Vigenère »,**

**« substitution à répertoire », « chiffrement de blocs », « CBC », « padding »,**

**« clé publique », « clé secrète », « clé privée », « protocole de Diffie-**

**Hellman », « DES », « RSA ». Que penser de l'affirmation : « les systèmes de chiffrement actuels sont forcément vulnérables, puisque les algorithmes qu'ils utilisent sont publics » ? Quels sont, parmi ces termes, ceux que l'on retrouve dans la programmation des questions de chiffrement et d'authentification ?**

* 1. **Historique**

#### Codage de César et principe de substitution

La cryptographie voit le jour vers 1900 ACN en Egypte avec l’utilisation de hiéroglyphes non-standards. On cite cependant le code de César (-50 ACN) pour débuter l’histoire de la cryptographie.

Il est basé sur le principe de substitution de caractères : remplacer chaque lettre du message par celle qui se trouve x positions plus loin dans l’alphabet. On dit encore que la clé de cryptage est le nombre de position. Une telle clé doit évidemment rester secrète. Cet algorithme de cryptage est public, c’est la connaissance de la clé qui est fondamentale, pas celle de l’algorithme.

L’algorithme de substitution a également vu le jour durant même période. Il consiste à effectuer une translation de certaines parties du message considéré.

Par exemple : transposer 2 lettres successives et laisser la 3e intacte, ce qui nous donne : Mot d’origine = guissart  ugissatr

La clé sera donc 2/3 par exemple, pour indiquer que les lettres ont été transposées 2 par 2 et que la 3e reste intacte. Pour compliquer ces systèmes, la transposition et la substitution peuvent être combinées.

Évidemment, ces algorithmes sont extrêmement sujets au cassage grâce à l’analyse fréquentielle.

#### Changement d’alphabet en cours de cryptage

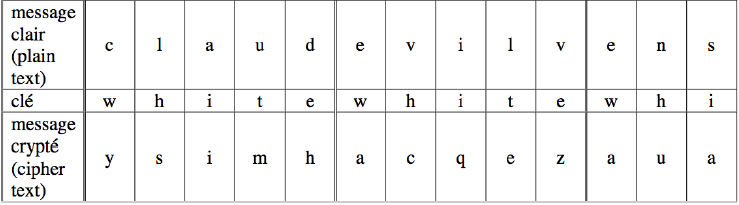
On parle aussi de chiffrement polyalphabétique. L’idée est de changer l’alphabet crypté en cours de cryptage d’un message. Pour la mise en œuvre, Alberti proposa le cadran chiffreur, qui met en correspondance l’alphabet clair (disque extérieur du cadran) et l’alphabet crypté (disque intérieur). Pour l’utiliser, il suffit de connaître une correspondance lettre claire/lettre cryptée pour disposer d’un système de cryptage.

Mais l’idée d’Alberti est de changer de « calage » à intervalles répétés. La clé est alors composée d’un mot dont les lettres donnent les calages successifs et d’un nombre indiquant le nombre de caractères que l’on peut coder avant de passer au calage suivant.

L’algorithme de cryptage est toujours public et une lettre claire n’est pas forcément cryptée de la même manière à chaque occurrence : le cryptage d’un caractère dépend de sa position. Et l’analyse fréquentielle n’est plus d’aucun secours.

#### La grille de Blaise de Vigenère

Pour réaliser un cryptage, on utilise une simple clé, par exemple « white ». On recopie alors cette clé le nombre de fois nécessaire en dessous du texte clair. Alors, la lettre de la clé fixe une ligne dans la grille, la lettre du message fixe la colonne : la lettre trouvée à l’intersection est la lettre cryptée.



L’algorithme de cryptage est toujours public et une lettre claire n’est pas forcément cryptée de la même manière à chaque occurrence : le cryptage d’un caractère dépend de sa position. Et, une fois de plus, l’analyse fréquentielle ne sert à rien !

#### L’utilisation combinée de la substitution et de la transposition

Le principe de transposition appliquée aux caractères n’est pas très riche en possibilités. Cependant, ce même principe peut devenir plus intéressant si on l’applique à des blocs de caractères. Ainsi, on peut imaginer découper le message clair en blocs de longueur fixe, puis de transposer les blocs selon une certaine règle, laquelle utilise éventuellement une clé.

Cependant, l’utilisation de blocs apporte un nouveau problème : que faire quand le dernier bloc est incomplet ? On pourrait penser à le compléter avec un caractère comme « & » mais ce serait donner une information au cryptoanalyste (ou le connard de hacker, au choix) qui tenterait de décoder. Il faudra donc un algorithme de remplissage (padding) qui génère des caractères de remplissage qui ne soient pas détectables trop facilement.

#### Les algorithmes symétriques ou à clé secrète

La même clé (secrète) est utilisée au chiffrement et au déchiffrement (exemple : remplacer chaque lettre du message par celle qui se trouve x positions plus loin dans l’alphabet).

#### Les algorithmes asymétriques ou à clé publique

Ici, on utilise une paire de clés (publique et privée), l’une servant à chiffrer et l’autre à déchiffrer, sans que la même clé puisse jouer les 2 rôles. Ces algos sont utilisés pour assurer la confidentialité (donc le chiffrement) et l’authentification (c’est-à-dire les signatures électroniques) : on parle encore de PKI (Public Key Infrastructure).

#### Diffie-Hellman – la clé partagée

* Alice et Bob se mettent d’accord publiquement sur 2 nombres n et p, n étant inférieur à p. il s’agit en fait des 2 paramètres d’une fonction puissance en arithmétique modulaire : nx % p
* Bob choisit un nombre aléatoire (A), nombre qu’il gardera secret. Alice choisit aussi un nombre (B) qu’elle gardera tout aussi secret.
* Bob calcule sa clé publique et l’envoie à Alice : a = nA % p

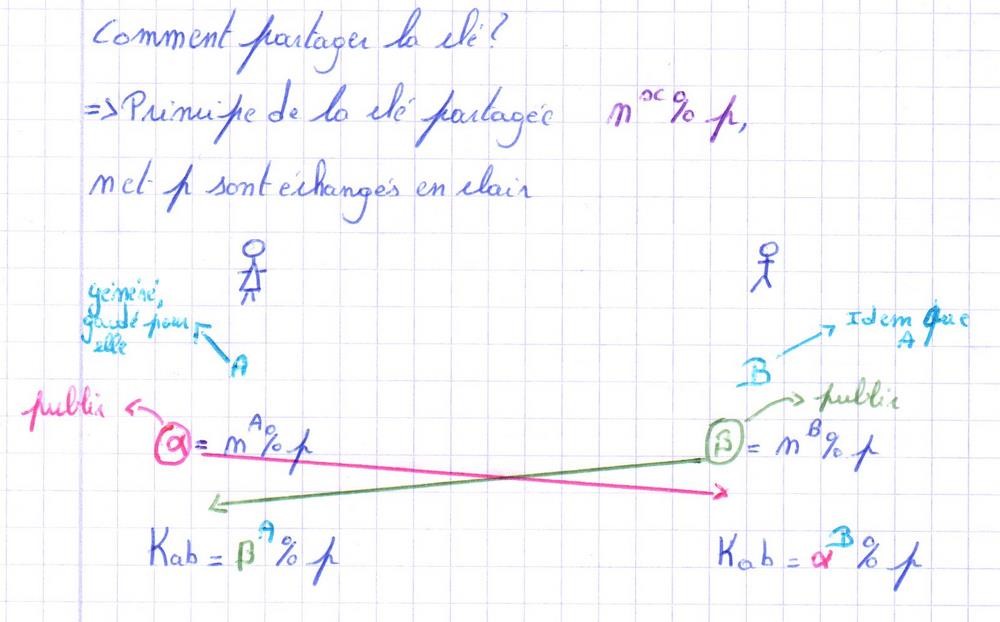
De même, Alice calcule sa clé publique et l’envoie à Bob : b = nB % p

Les 2 correspondants disposent donc à présent chacun des 2 clés publiques.

* Bob utilise la clé publique d’Alice pour construire une clé secrète : Kbob = bA % p

De manière similaire, Alice construit sa clé secrète grâce à la clé publique de Bob : Kalice = aB % p

La clé secrète sera donc commune aux deux parties. Pour qu’un hacker puisse construire la même clé, il faudrait qu’il connaisse A ou B, qui sont restés secrets ou qu’il essaie de les deviner, ce qui est très difficile car la fonction modulaire est une fonction non-réversible.



### Mode de chiffrement

On peut encore, selon l’un ou l’autre algorithme, distinguer la manière dont les blocs d’un texte clair correspondent à des blocs de texte chiffré : on parle alors de « mode de chiffrement ».

Le chiffrement de bloc :

Transforme un bloc de données claires d’une taille préalablement fixée en un bloc de données chiffrées de même taille. Pour chiffrer un message, il faut donc :

* Découper celui-ci en blocs de taille acceptée par l’algorithme utilisé ;
* Chiffrer chaque bloc : ceci se fait en utilisant un mode chainage qui peut être l’un des modes de chiffrement courants :
  + ECB (Electronic Code Book) : un bloc de texte clair se chiffre en un bloc de texte chiffré, indépendamment des autres blocs ; il faut remarquer que 2 blocs identiques produisent les mêmes blocs chiffrés et qu’il y a peu de protection sur l’intégrité du message, puisque l’indépendance des blocs chiffrés ne permet pas de détecter des permutations, duplications ou suppressions de blocs.
  + CBC (Cipher Bloc Chaining) : chaque bloc de texte clair est combiné par un XOR avec le bloc de texte chiffré précédent. Cette fois, 2 blocs identiques ont peu de chance de produire les mêmes blocs chiffrés, et il y a protection sur l’intégrité du message, puisque des permutations, duplications ou suppressions de blocs auront des implications sur les blocs résultants.

### Les systèmes de chiffrement actuels sont forcément vulnérables, puisque les

**algorithmes qu’ils utilisent sont publics**

Faux ! Les systèmes de chiffrement actuels ne sont pas cassables dans des temps « raisonnables ». En effet, ceux-ci se basent sur des algorithmes modulaires, donc sur la non-reversibilité des calculs. La force des chiffrements actuels n’est pas l’algorithme utilisé mais bien la complexité de la clé.

#### Quels sont, parmi ces termes, ceux que l’on retrouve dans la programmation des questions

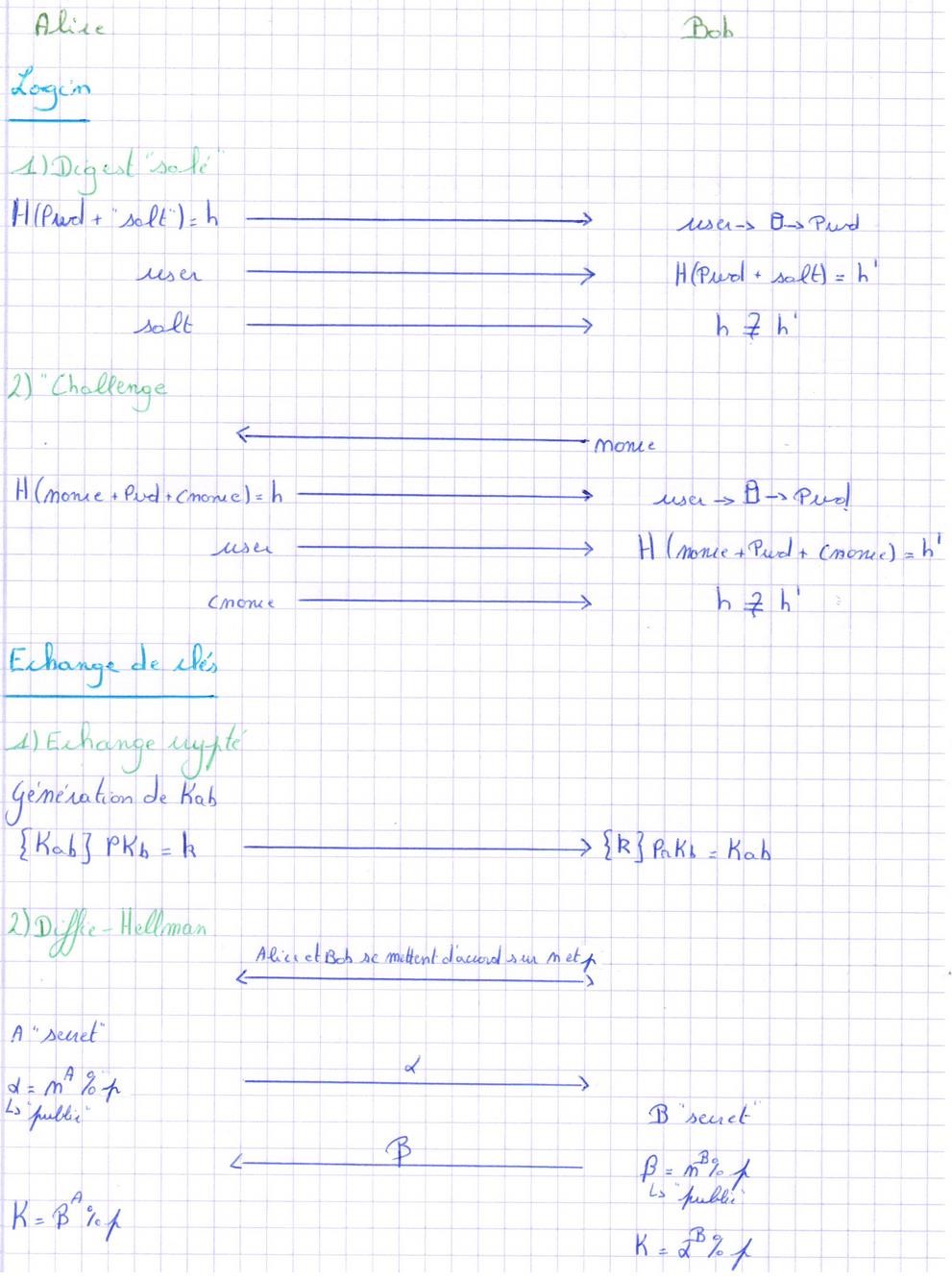
**de chiffrement et d’authentification ?**

« chiffrement de blocs », « padding », « clé publique », « clé secrète », « clé privée », « Diffie- Hellman », « AES », « RSA ».

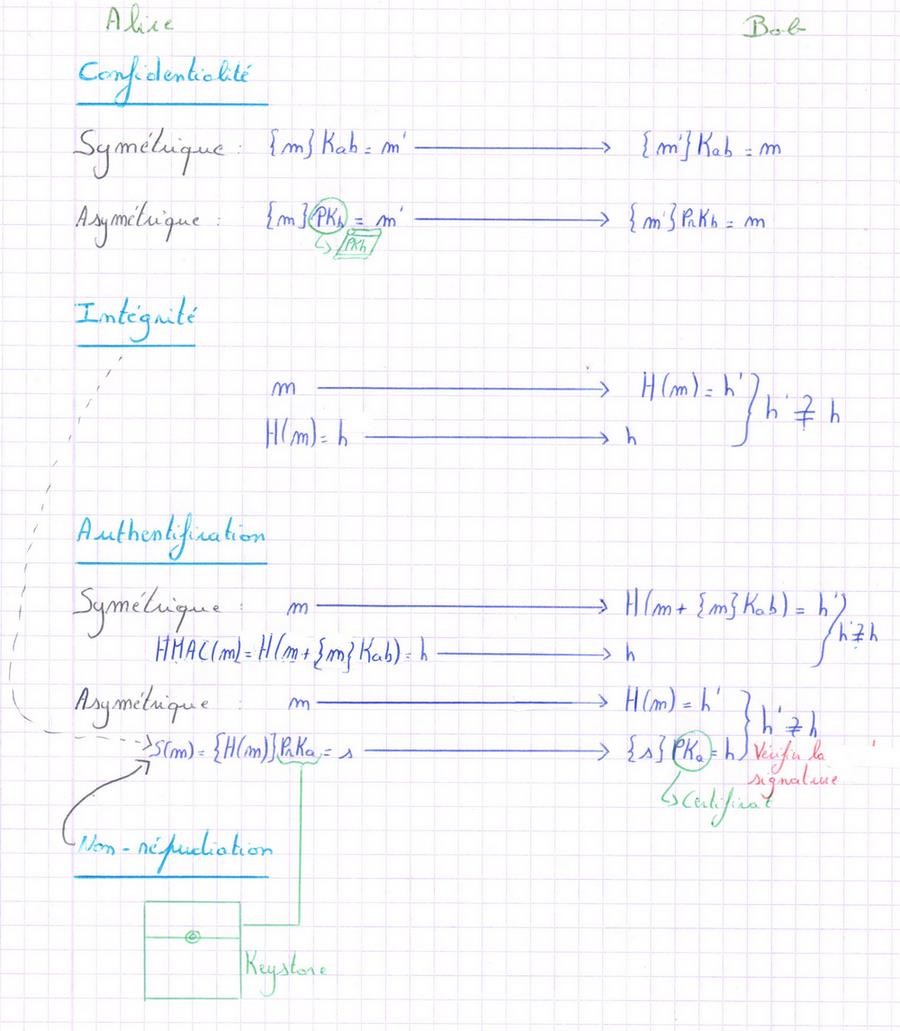
1. **Donner et expliquer une synthèse sous forme de tableau les différentes possibilités offertes par la cryptologie moderne en termes de confidentialité, intégrité, authentification, non répudiation, échange de clés (utiliser les notations K, PrK, PK, H, S, …). Pourquoi peut-on souvent distinguer une phase**

**« handshake » et une phase « communication » ? Quel est habituellement le rôle de la cryptographie symétrique et celui de la cryptographie asymétrique ?**

1. **Phase de handshake**



1. **Phase de communication**



* 1. **La synthèse**

Indispensable pour l’authentification du client et puis la génération des clés.

En effet, il se peut que n’importe qui ne puisse pas avoir accès au service voulu, nous nous retrouvons donc avec un login avant que le mécanisme de chiffrement se mette en place. Pour ce faire, un simple digest est utilisé. La génération des clés doit être également faite avant la partie communication  Pas de clé, pas de chiffrement ! Ensuite, il ne sert à rien de chiffrer les messages si cette clé passe en claire (en risquant d’être interceptée). Plusieurs échanges sont possibles, comme par exemple l’échange public (Diffie-Hellman).

Ensuite, nous constatons la partie communication, qui, elle, est la partie qui échange les informations d’un côté à l’autre avec différents moyens pour vérifier l’intégrité, la confidentialité ou l’authentification. Mais à ce stade, toute configuration préalable est terminée.

### Cryptographie symétrique vs asymétrique

Le cryptage symétrique utilise la même clé pour chiffrer et déchiffrer un message. Le danger réside évidement dans la protection de la clé … il ne faut pas qu’elle soit interceptée.

Vu que cette clé doit rester secrète, on parle d’algorithme à clé secrète (on parle aussi de clé de session, cf plus loin). Ce type de chiffrement est dit « léger ».

Étant donné que les personnes qui interviennent dans un mécanisme de chiffrement peuvent être séparées de plusieurs millions de milliards de kilomètres, il peut être très difficile de protéger la clé secrète. C’est là qu’intervient le chiffrement asymétrique.

Dans ce type de cryptage, chaque utilisateur possède 2 clés :

* Une clé publique, qui peut donc être connue de tous. La seule contrainte est qu’elle soit accompagnée d’un certificat provenant d’une autorité en laquelle l’émetteur et le récepteur ont confiance.
* Une clé privée que l’utilisateur peut générer sans devoir la faire circuler sur le réseau (c’est évidemment ici qu’apparaît le caractère sécurisé de la chose)

L’idée est donc que n’importe qui peut envoyer un message à JMW, il suffit de crypter un message avec sa clé publique, mais seul JMW pourra le décrypter avec sa clé privée.

En pratique, on n’utilise pas de cryptage asymétrique DIRECTEMENT pour sécuriser des échanges. Cela serait beaucoup trop gourmand en ressources. La solution réside en un couplage des 2 techniques (c’est là qu’intervient le terme de clé de session)

#### L’expéditeur du message

* + - * Fabrique une clé secrète (de session)
      * Code son message avec cette clé (de façon symétrique donc)
      * Code cette clé secrète avec la clé publique du destinataire (de façon asymétrique)
      * Envoie le message crypté symétriquement et la clé de session cryptée asymétriquement

#### Le destinataire

* + - * Décrypte la clé de session grâce à sa propre clé privée
      * Utilise cette clé de session pour décrypter le message

## Décrire les acteurs de la cryptographie symétrique : chiffrement, intégrité, authentification (utiliser les notations K, PrK, PK, H, S, …). Quel problème principal apporte l'utilisation d'une seule clé (clé de ssion) et comment peut-on le résoudre (deux techniques) ? Comment procède-t-on en programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ? Expliquer les différences apportées par le chiffrement symétrique Rijndael/AES en CBC.

### Présentation

Le cryptage symétrique utilise la même clé pour chiffrer et déchiffrer un message. Le danger réside évidement dans la protection de la clé … il ne faut pas qu’elle soit interceptée.

Vu que cette clé doit rester secrète, on parle d’algorithme à clé secrète (on parle aussi de clé de session, cf plus loin). Ce type de chiffrement est dit « léger ».

Étant donné que les personnes qui interviennent dans un mécanisme de chiffrement peuvent être séparées de plusieurs millions de milliards de kilomètres, il peut être très difficile de protéger la clé secrète. C’est là qu’intervient le chiffrement asymétrique.

(nb : c’est tout à fait normal que des parties soient identiques à la question précédente …)

On a donc un souci lorsque l’on utilise cette clé et uniquement cette clé. Comme on l’a dit, on va alors coupler le cryptage symétrique avec le cryptage asymétrique.

### 1ere solution : coupler 2 cryptages

Dans ce type de cryptage (asymétrique), chaque utilisateur possède 2 clés :

* Une clé publique, qui peut donc être connue de tous. La seule contrainte est qu’elle soit accompagnée d’un certificat provenant d’une autorité en laquelle l’émetteur et le récepteur ont confiance.
* Une clé privée que l’utilisateur peut générer sans devoir la faire circuler sur le réseau (c’est évidemment ici qu’apparaît le caractère sécurisé de la chose)

L’idée est donc que n’importe qui peut envoyer un message à JMW, il suffit de crypter un message avec sa clé publique, mais seul JMW pourra le décrypter avec sa clé privée.

En pratique, on n’utilise pas de cryptage asymétrique DIRECTEMENT pour sécuriser des échanges. Cela serait beaucoup trop gourmand en ressources. La solution réside en un couplage des 2 techniques (c’est là qu’intervient le terme de clé de session)

#### L’expéditeur du message

* + - * Fabrique une clé secrète (de session)
      * Code son message avec cette clé (de façon symétrique donc)
      * Code cette clé secrète avec la clé publique du destinataire (de façon asymétrique)
      * Envoie le message crypté symétriquement et la clé de session cryptée asymétriquement

#### Le destinataire

* + - * Décrypte la clé de session grâce à sa propre clé privée
      * Utilise cette clé de session pour décrypter le message

### 2e solution : Diffie-Hellman

Une variante, qui évite de transférer la clé de session sur le réseau, consiste à permettre aux 2 parties de générer localement la même clé de session sur base de renseignements échangés par le réseau (que celui-ci soit préalablement sécurisé ou non). C’est ce que permet l’algorithme de Diffie-Hellman qui se base sur un échange de nombre premiers et aléatoires. Le danger est cependant qu’un intrus peut se glisser dans cette conversation préalable et se faire passer pour l’expéditeur quand il converse avec le destinataire et vice-versa.

### Rijndael vs DES

Rijndeal, ou AES, utilise des clés de 128, 196 ou 256 bits. DES utilise des blocs de 64 bits et une clé secrète de 56 bits.

La principale différence entre les 2 types de chiffrement est que DES utilise un système de blocs chaînés (EBC) (ou chaque bloc dépend donc de son prédécesseur) alors qu’AES utilise un système où les blocs sont indépendants (CBC).

Puisque CBC se base sur le fait que chaque bloc dépend de son prédécesseur, comment faire pour le 1er bloc ‘sieur ?

On a besoin d’un vecteur d’initialisation.

La méthode d’initialisation de l’objet de chiffrement a besoin de ce paramètre :



Ou en pratique



### En Java

Il nous faut évidemment, pour débuter, nous procurer une clé de cryptage. C’est le travail d’un générateur de clé, objet instanciant une classe de type KeyGenerator. Nous utilisons pour ce faire une méthode factory



Qui produit une valeur de départ totalement imprévisible.

SecretKey est une interface qui dérive de l’interface Key et qui n’apporte rien de plus que l’algorithme qui lui correspond. Nous obtenons la clé par SecretKey cle = cleGen.generateKey() ;

On veut ensuite obtenir un moyen de chiffrer. Cet « outil » est matérialisé par la classe « Cipher ». la factory à utiliser pour obtenir une instance de cette classe est là même que toutes les autres.

Nous pouvons enfin crypter notre message, on initialise d’abord l’objet chiffrement avec la clé obtenue auparavant : Void init(int opmode, Key key)

Le 1er paramètre prenant l’une des valeurs

* Public static final int ENCRYPT\_MODE = 1
* Public static final int DECRYPT\_MODE = 2

Il n’y a plus qu’à réaliser maintenant le cryptage sur notre tableau de bytes : byte[] doFinal(byte[] message) ;

## Expliquer le principe du chiffrement asymétrique selon RSA (utiliser les notations K, PrK, PK, H, S, …). En particulier, à quoi ressemblent les clés publiques et privées de cet algorithme ? Un tel chiffrement est-il réellement utilisé pour crypter l’intégralité d’une communication réseau sécurisée ?

**Comment procède-t-on en programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ?**

### Principe

Le chiffrement asymétrique permet de s’affranchir de la nécessité pour les 2 parties s’échangeant des messages de devoir disposer de la même clé secrète comme c’est le cas pour les algorithmes de chiffrement symétrique. En effet, l’échange de cette clé secrète peut être problématique (un hacker pourrait sniffer le réseau comme certains sniffent de la coke en allant retirer toutes les 5 minutes).

On se trouve donc en asymétrique avec une paire de clé, chaque utilisateur en possède une :

* Une clé publique, qui peut donc être connue de tous. La seule contrainte est qu’elle soit accompagnée d’un certificat provenant d’une autorité en laquelle l’émetteur et le récepteur ont confiance.
* Une clé privée que l’utilisateur peut générer sans devoir la faire circuler sur le réseau (c’est évidemment ici qu’apparaît le caractère sécurisé de la chose)

Lorsqu’Alice souhaite envoyer un message crypté à Bob, elle doit se procurer la clé publique de Bob, crypter le message avec cette clé publique et l’envoyer à Bob. Ce message ainsi crypté ne sera décryptable que par la clé privée de Bob. On est ainsi assuré que seul Bob peut décrypter le message.

Tout se base sur le fait que, s’il est facile de multiplier 2 nombres premiers l’un par l’autre, il est, par contre, extrêmement difficile de retrouver 2 tels nombres à partir d’un nombre donné.

### RSA

Le chiffrement asymétrique RSA (R. Rivest, A. Shamir, L. Adleman) est basé sur la difficulté de factoriser les grands nombres, c’est l’algorithme le plus utilisé au monde. Il est considéré comme efficace avec une clé de 1024 bits.

Algorithme :

Formule de cryptage avec la clé publique : <caractère chiffré> = (<caractère à chiffrer>e) % n Formule de décryptage avec la clé privée : <caractère déchiffré> = (<caractère à déchiffrer>d) % n

Les composants des couples (n, e) et (n, d), formant respectivement les clés publiques et privées, sont calculés au moyen de la théorie des nombres premiers.

En pratique :

* Clé publique (PK) : on choisit aléatoirement deux nombres premiers p et q relativement grands. Alors n est simplement donné par leur produit : n = p \* q

L’exposant public (e) est choisi comme un entier dans [3, n-1] (donc inférieur à n) et premier avec : z = (p - 1) \* (q – 1).

En pratique, e est souvent pris égal à 3 ou encore 65537 si la valeur de n le permet.

* Clé privée (PrK) : le module (n) est le même. L’exposant privé (d) est un nombre tel que (e \* d – 1) soit divisible par z.

### Un tel chiffrement est-il réellement utilisé pour crypter l’intégralité d’une communication réseau sécurisée ?

Nenni, ça coûterait bonbon aux ressources ! On va utiliser le chiffrement asymétrique pour crypter la clé de session, et ainsi garantir la garantie (…) de nos futurs messages envoyés avec un cryptage symétrique.

#### L’expéditeur du message

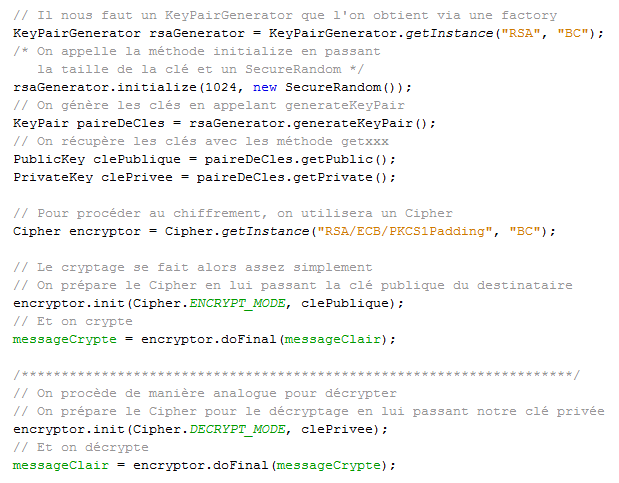
* + - * Fabrique une clé secrète (de session)
      * Code son message avec cette clé (de façon symétrique donc)
      * Code cette clé secrète avec la clé publique du destinataire (de façon asymétrique)
      * Envoie le message crypté symétriquement et la clé de session cryptée asymétriquement

#### Le destinataire

* + - * Décrypte la clé de session grâce à sa propre clé privée
      * Utilise cette clé de session pour décrypter le message

### En Java

Avant toute chose, il faut se procurer un provider, par exemple Cryptix ou Bouncy Castle.



## Expliquer comment les notions d'interfaces et de méthodes factory sont utilisées en programmation Java dans le contexte de la cryptographie (chiffrement et authentification). Comment une méthode comme getInstance() fonctionne-t-elle (en particulier, comment trouve-t-elle les classes providers et comment peut-elle les instancier) ?

### Principe des factory

En cryptographie (et en général), les factory sont nécessaires car elles permettent de choisir au moment de l’exécution une classe qui implémentera les algorithmes de chiffrements, de digest ou de remplissage que l’on spécifiera à l’aide de la méthode factory getInstance().

Plus particulièrement, JCA (Java Cryptography Architecture) fournit des éléments nécessaires (classes, interfaces) à l’utilisation de la crypto en Java. Ces éléments font partie intégrante du JDK.

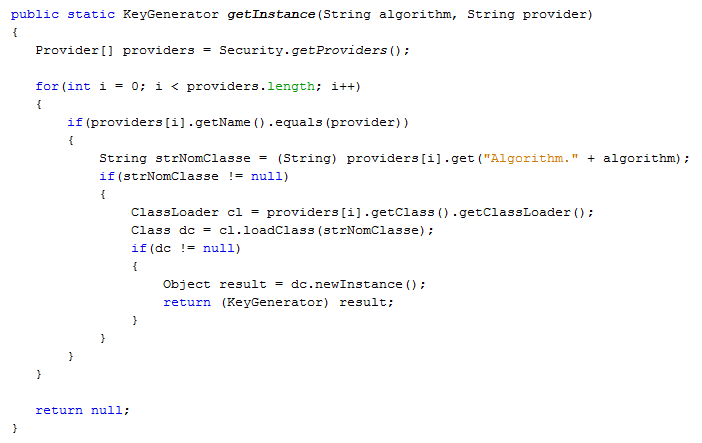
On retrouve dans les packages java.security et javax.crypto :

* Des APIs (Application Programming Interface) : il s’agit des méthodes, publiques, que l’on peut appeler directement dans une application. Le plus souvent, une API fait usage d’une SPI de manière encapsulée.
* Des SPIs (Service Provider Interface) : il s’agit des méthodes d’interfaces, donc en fait, des méthodes sans implémentation. Fournir l’implémentation d’une SPI permet donc de se servir de l’API correspondante et l’on disposera en fait d’autant de versions de l’API que l’on fournit de SPIs distinctes.

JCA propose la démarche suivante : les véritables classes nécessaires au travail cryptographique ne sont pas instanciées au moyen d’un constructeur mais plutôt en utilisant des méthodes factory, la plus répandue étant getInstance(). Cela permet de laisser au système le choix d’instancier pour nous une instance du provider le plus adéquat et qui répond au mieux à nos attentes, selon le paramètre passé et ce sans pour autant connaître le nom de la classe qui sera finalement instanciée.

### Programmation en Java

Du côté de la programmation Java, on utilise donc des méthodes telles que getInstance(à ou getConnection() auxquelles on passe des paramètres tels que l’algorithme souhaité ou encore le provider de la base de données. L’utilisation massive de méthodes factory permet de rendre le code plus compréhensible pour le programmeur, mais aussi entièrement modulaire et portable (en effet, un changement de provider ne devrait pas nécessiter de gros changement dans le code source).



Les classes qui fourniront l’implémentation seront fournies par des providers. Pour la cryptographie, le JCE (Java Cryptography Extension) se charge de fournir une implémentation à ces interfaces, mais le gouvernement américain en limite l’usage aux USA et au Canada. Étant européen, on utilisera les classes de sociétés tierces appelées CSP (Cryptography Service Provider). On peut citer Cryptix, qui est gratuit et utilisable librement, ou encore Bouncy Castle.

La classe Provider hérite elle-même de la classe Properties qui permet de fournir une liste de propriétés, ces propriétés serviront en cryptographie à renseigner sur les algorithmes effectivement implémentés.

Ainsi, la méthode getInstance(), qui est toujours une méthode de classe, va rechercher par le biais de la classe Security la liste des providers enregistrés (cette liste se trouve dans le fichier java.security). Selon un ordre de préférence défini par l’utilisateur, la méthode demandera à chaque provider un objet de la classe algorithme recherché. Si le provider visé la possède, il en fournira une instance, sinon, la recherche passe au provider suivant ou donnera une exception si la liste est épuisée.

## Expliquer la nature et la finalité d'un message digest. Comment on peut l'utiliser dans un contexte d'authentification se substituant au schéma classique login-password ? Comment procède-t-on en programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ? Quel rapport avec les HMACs et comment ces derniers assurent-ils une authentification (légère) fiable (utiliser les notations K, PrK, PK, H, S, …) ?

### Principe du message digest

Dans le cadre d’une communication réseaux, il est important de savoir si les données que l’on reçoit sont les MÊMES que celles qui ont été envoyées. En effet, il est possible qu’une personne mal intentionnée trifouille le message et le modifie.

Dans ce contexte, on utilise un message digest qui représente « la valeur de hashage » d’un document. Celle-ci s’obtient en utilisant une fonction de hashage qui, recevant un ensemble de données d’une taille quelconque, fournit une chaine de taille fixe (typiquement 128 bits) en sortie.

A note que les fonctions de hashage idéales ne peuvent être inversées (ou, du moins, il faut un temps quasiment infini pour y parvenir) : il est donc impossible de retrouver la donnée qui a produit un digest.

L’intégrité est donc vérifiée

* En calculant le message digest du texte obtenu
* En le comparant au message digest accompagnant le texte

En cas d’égalité, on peut penser que le texte original n’a pas été altéré.

### Utilisation dans un contexte d’authentification

#### Le client

1. Connait son mot de passer pour accéder au serveur
2. Crée un digest avec son mot de passe
3. Envoi son nom et le digest au lieu de son mot de passe en clair

#### Le serveur

1. Reçoit le nom et le digest
2. Recherche le nom dans son fichier ou sa BD et calcule le digest correspondant
3. Compare le digest reçu avec le digest calculé. S’ils correspondent, l’utilisateur est authentifié.

Pour plus de sécurité, le digest sera calculé avec le mot de passe complété d’un nombre aléatoire et de l’heure. Le digest devient alors « salé », attention donc, cela peut donner soif … (humour drôle).

Ces 2 informations seront envoyées en clair au serveur afin qu’il puisse lui aussi calculer le digest pour le comparer à celui qu’il a reçu.

### Programmation en Java

Tout d’abord, il faut obtenir une instance de digest qui utilise l’algorithme SHA-1 (par exemple) et qui est fournie par le provider Bouncy Castle (encore par exemple) au moyen de la méthode factory



Ce qui donne pour l’exemple



Il faut ensuite préparer le digest en lui passant nos « ingrédients » (mot de passe, nombre aléatoire, heure) en utilisant la méthode void update(byte[] input).

Si on ne peut pas utiliser getBytes(si on a donc autre chose que des String), on peut penser à utiliser la classe ByteArrayOutputStream couplé à un DataOutputStream pour pouvoir utiliser les habituelles méthodes writeXXX.

On utilisera finalement byte[] digest(byte[] input) qui applique effectivement l’algorithme sélectionné pour fournir un digest sous forme d’un tableau de bytes.

Afin de comparer 2 digest, on utilisera la méthode boolean isEqual(byte[] digest1, byte[] digest2).

### HMAC

En généralisant la notion de digest, on définit un MAC (Message Authentication Code) comme un petit bloc de taille fixe qui a également pour objectif d’assurer l’authentification et l’intégrité d’un message.

Techniquement, un MAC simple peut être un digest résultat d’un hashage (MAC simple hashé).

Mais il peut aussi utiliser une clé secrète (MAC simple à clé) : le résultat de l’application d’un chiffrement symétrique utilisant cette clé au message constitue alors le MAC. Celui-ci est envoyé au destinataire avec le message en clair. Le destinataire reçoit donc le message et le MAC.

Comme il connait la clé secrète, il peut calculer sur le message clair un second MAC.

Si les 2 MACs sont semblables

* Cela signifie que l’intégrité est vérifiée
* Cela assure aussi l’authentification puisqu’il faut partager le secret de la clé secrète pour vérifier cette intégrité

Un MAC peut aussi utiliser conjointement une clé et un digest (MAC complexe à clé et hashage) : celui-ci est construit avec la chaine d’entrée et cette même chaine préalablement cryptée au moyen d’une clé secrète en plus. Le représentant typique de cette famille est le HMAC (keyed-Hash Message Authentication Code).

HMAC(m) = H(m + {m}Kab) => (m + x) => x = ?= H(m + {m}Kab)

Kab = clé de session (symétrique)

## Qu'est-ce qu'une signature digitale (ou numérique ou électronique) et comment l'utiliser pour assurer l'intégrité et l'authentification (lourde) ? Qu'entend-on par "vérifier une signature" (utiliser les notations K, PrK, PK, H, S, …) ? Quelles sont les analogies et/ou les différences avec une signature manuscrite classique ? Comment procède-t-on en programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ? Pourquoi le catalogue des fonctionnalités cryptographiques offertes par un provider ne citent-elles pas apparemment les algorithmes de cryptage asymétrique ?

### Principe

Une signature électronique est un bloc de données qui a été créé en utilisant une clé privée et qui peut être vérifié par la clé publique correspondante sans la connaissance de la clé privée. Elle permet d’authentifier l’expéditeur des données. En effet, avec la confidentialité et l’intégrité, l’authentification est certainement le problème majeur de la transmission sécurisée d’information au niveau applicatif.

Signature numérique ou digitale ne sont que des synonymes.

Attention qu’une signature n’est pas nécessairement un certificat tel que nous les connaissons dans le monde de l’informatique.

### La construction d’une signature basée sur un digest

#### Du côté de l’expéditeur

* Construction d’un message digest du document à envoyer, selon un algorithme de digest (SHA-1 ou MD5)
* Chiffrer au moyen d’un algorithme approprié de chiffrement asymétrique (RSA) la chaine obtenue pour le digest en utilisant la clé privée du signataire (qui restera chez ce signataire). On obtient la signature électronique.
* L’expéditeur envoie alors le message et la signature.

#### Du côté du destinataire

* Réception du message avec la signature et déchiffrement de cette signature au moyen de la clé publique du destinataire du message (car chiffrement avec la clé privée du destinataire). Il obtient ainsi le message digest calculé lors de l’envoi.
* Calculer un second message digest pour le message obtenu. Si les 2 digest correspondent, on peut être assuré que l’information n’a pas été falsifiée, ce qui assure donc l’intégrité ; dans ce cas aussi, la clé privée utilisée était bien celle du signataire, ce qui assure donc aussi l’authentification.
* On peut également convaincre un tiers que le document reçu a bien été envoyé par le propriétaire de la clé privée, pas par quelqu’un d’autre : on résout ainsi le problème de la non-répudiation.

### Vérification de la signature

La vérification de la signature consiste à « déchiffrer » la signature reçue à l’aide de la clé publique de l’expéditeur puis, à comparer le message clair obtenu avec le digest calculé à partir du message effectivement reçu. Si les 2 digest sont identiques, alors la signature est valide.

La vérification d’une signature assure la détection de la perte d’intégrité du message ainsi que l’authentification de l’origine du message. La vérification ne garantit l’authentification que si l’auteur est bien le seul détenteur du secret qu’est la clé privée de signature.

### Analogie avec la signature manuscrite

Par rapport à la notion de signature manuscrite, une nuance est d’importance et mérite d’être bien soulignée :

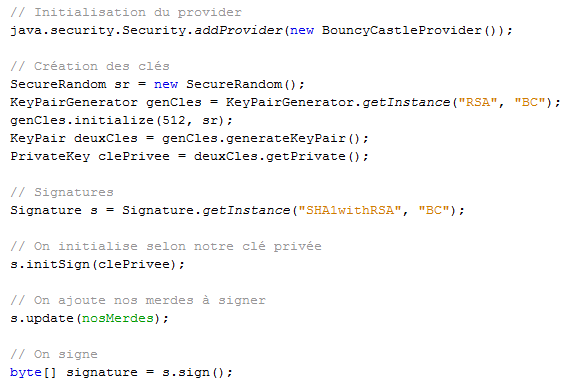
« *Une signature digitale est fonction, outre de la clé privée utilisée, du message qu’elle accompagne : elle est donc propre non seulement au propriétaire de la clé privée mais aussi à ce message* ».

Cela signifie que chaque utilisation d’une signature électronique, pour une même personne, sera différente pour chaque message alors que pour une signature manuscrite, la signature reste identique quel que soit le message. Cependant, le but reste le même : authentifier l’expéditeur du message.

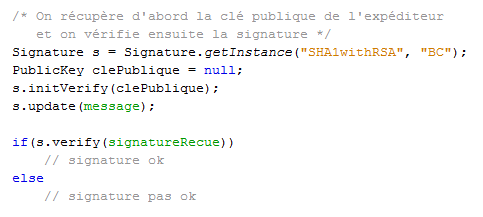
Cette différence entre les 2 types de signature est, en pratique, due au fait qu’un hacker qui aurait dérobé une signature digitale ne saurait s’en servir que pour signer un autre message.

### Exemple en Java

#### Coté expéditeur



* + 1. **Coté destinataire**



* 1. **Pourquoi le catalogue des fonctionnalités cryptographiques offertes par un provider ne citent-elles pas apparemment les algorithmes de cryptage asymétrique ?**

Car ceux-ci nécessitent des fonctions mathématiques coûteuses en ressources machines aussi bien en puissance qu’en temps, ce qui implique un temps d’exécution des programmes trop important. Si l’on souhaite crypter toute une communication, il faut utiliser un mélange de cryptage symétrique et asymétrique : la clé de session.

## Qu'entend-on par « certificat », « certificat X509 », « niveau (1-2-3,3+) » et

**« vérifier un certificat » ? A quoi correspondent les formats DER et PEM ? Qu'est-ce qu'un « keystore » ou « magasin à clés » ? Expliquer dans une liste des principales commandes comment les couples de clés et les certificats sont gérés au moyen de l'outil keytool. Comment procède-t-on en programmation Java (quelles classes, fichier et classe KeyStore, obtenues comment, quelles méthodes, …) ?**

En utilisant une signature digitale, on peut certifier qu’une clé publique appartient bien à une certaine personne en créant ainsi ce que l’on appelle un certificat, c’est-à-dire un message qui a valeur officielle et qui a la reconnaissance de tous.

**Concrètement on construit un message comportant :**

1. l’identification de son propriétaire
2. la clé publique

Ce message est ensuite signé au moyen de la clé d’un organisme en qui on a toute confiance que l’on appelle les certifications authority (CA) (par exemple : Verisign). Le message ainsi complété de l’identification du CA et de sa signature constitue ce que l’on appelle un certificat d’authentification. Il servira d’attestation officielle que la clé publique en question est bien celle d’une personne précise.

|  |
| --- |
| Certificat  Informations fondamentales |
| - Identification du propriétaire.  - Clé publique.  - Identification du CA  - Signature digitale du CA |

**La norme la plus utilisée est dénommée X.509 et stipule qu’un certificat comporte :**

|  |
| --- |
| Certificat X509 :  Informations plus complètes |
| - Numéro de version  - Numéro de série  - Identifiant de l’algorithme de signature  - Identification du propriétaire de la clé publique certifiée.  - Période de validité  - Identification du CA  - Information sur l’algorithme de la clé publique  - Clé publique.  - Extensions diverses  - Signature digitale du CA pour les champs ci-dessus |

**Formats de certificats**

|  |
| --- |
| DER (Definite Encoding Rule) -> encodage ASN.1 (.der, .cer, .crt, .cert) |
| PEM (Privacy Enhanced Mail) -> DER en base64 + en-tetes en ASCII |

* Comment savoir si le certificat que l’on vient de recevoir, et qui fournit la clé publique de quelqu’un, est le vrai ?

On va d’une part calculer le digest des informations contenues dans le certificat et d’autre part utiliser la clé publique du CA pour obtenir le digest qui est à la base de la signature : si les deux digests coïncident, la signature est donc déclarée valide.

Pour vérifier cette signature, il faut donc disposer de la clé publique du CA, deux possibilités:

* le CA est bien connu, sa clé publique aussi et la vérification peut se faire sans problème.
* le CA est seulement local, il faut donc se procurer un certificat pour ce CA local, certificat signé par un autre CA plus connu. L’opération est à recommencer pour ce nouveau certificat et ainsi de suite jusqu’à parvenir à un CA connu (on parle de chaîne de certificat).

La classe KeyStore du package java.security permet de réaliser la gestion des clés. En fait il s’agit d’une sorte de dictionnaire qui contient, de manière cryptée, deux types d’entrées :

* Key Entry : une clé privée et une liste de certificats concernant la clé publique correspondante. On peut ainsi prouver qui l’on est. Rien n’interdit de disposer de plusieurs couples clé publique/clé privée.
* Trusted Certificate Entry : un certificat d’une personne considérée comme sûre. On peut ainsi authentifier une autre partie.

Ces informations sont désignées, au sein de la structure, par des identifiants appelés des alias qui permettront de retrouver l’information demandée (comme une clé).  
De manière sémantique, un objet KeyStore ressemble à ceci :

|  |  |  |
| --- | --- | --- |
| Alias | Information | Usage |
| cleApplets | Clé privée+clé publique + certificat(s) | Signer des applets |
| cleMails | Clé privée+clé publique + certificat(s) | Signer des mails |
| Samu-elle | Certificat | Authentifier |
| Quevin | Certificat | Authentifier |
| TonyByzou | Certificat | Authentifier |

L’implémentation de l’objet KeyStore est laissée à la discrétion des providers. Une implémentation par défaut est cependant fournie par Sun : appelé « JKS », elle utilise un fichier au format propriétaire (nommé classiquement .keystore).

On peut spécifier un type de keystore dans la méthode de la classe :

* public static KeyStore getInstance(String type)

La classe comporte des méthodes permettant d’ajouter des entrées à l’objet crée :

* public final void setKeyEntry(String alias, byte[] key, Certificate[] chain)
* public final void setKeyEntry(String alias, byte[] key, char[] password, Certificate[] chain)
* public final void setCertificateEntry(String alias, Certificate cert)

On retrouve les informations au moyen des méthodes :

* public final Key getKey(String alias, char[] password)
* public final Certificate getCertificate(String alias)

Cette dernière méthode fournit donc le certificat associé à l’alias précis.

Concernant le certificat on retrouve la méthode :

* public abstract void verify(PublicKey key)

Qui permet de verifier que le certificate a été signé au moyen de la clé privée associée à la clé publique passée en argument.

On peut aussi gérer une keystore au moyen d’un outil en ligne de commande appelé keytool, outil qui est apparu avec le jdk1.2. On peut générer un certificat signé par l’utilisateur. La paire de clé produite (option –keygen), selon l’algorithme précisé (option –keyalg) correspondra à une entrée de type « key entry », la clé publique étant celle faisant l’objet du certificat. L’entrée en question peut être désignée par un alias (option –alias). Le nombre de bits utilisés peut être spécifié (option –keysize). On se souviendra enfin qu’un certificat contient une identification du propriétaire. L’option –dname permet de définir un distinguished name (DN), qui décrit une personne de manière normalisée.

Dans tout certificat on retrouve un “Certificate fingerprint”, qu’il faut comprendre comme un digest qui a été calculé sur le certificat entier.

Il existe 2 types de certificats :

* certificat auto-signé le signataire d’un tel certificat est le même que celui dont la clé publique fait l’objet de la certification.
* certificat réel atteste de sa clé publique, un utilisateur quelconque doit construire une demande que l’on appelle un CSR (Certificate Signing Request). Il s’agit d’un fichier qui contient :
* la clé publique du demandeur
* la signature construite au moyen de la clé privée du demandeur

Le CA, après réception, pourra alors vérifier le bien fondé de la demande et, dans l’affirmative, fournir le certificat demandé.

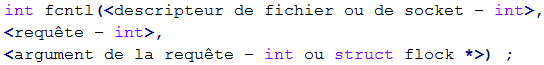
Dans le cas où l’on reçoit un certificat d’un tiers (qui n’a donc pas d’alias dans le Keystore), on créera une « Trusted Certificate Entry ».

## Expliquer les principes généraux de paramétrage des sockets (primitives, types de paramètres, …). Expliquer avec du code concret les deux situations classiques où il est nécessaire de paramétrer les sockets TCP pour le mode non bloquant et la réutilisation d'un couple adresse-port et les deux Expliquer avec du code concret les deux situations classiques où il est nécessaire de paramétrer les sockets les sockets UDP pour mettre en place un time-out et configurer un système multicast.

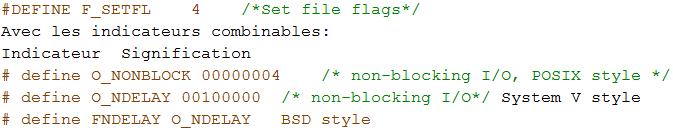
Même si dans la plupart des cas on utilise le comportement par défaut des sockets, on peut dans certains cas modifier ceux-ci afin de mieux maîtriser leurs comportements. On peut par exemple donner à une socket des caractéristiques non bloquantes en utilisant deux fonctions qui ne sont pas propres aux sockets mais concernent tout fichier au sens d’UNIX, il s’agit de **fcntl**() et **ioctl**().

**La fonction fcntl** : contrôle des descripteurs de sockets

En utilisant les headers fcntl.h, sys/types.h et unistd.h, la fonction fcntl() classique a pour prototype:



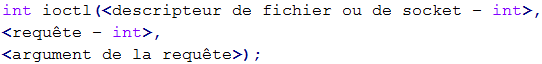
L’utilisation de cette fonction, au spectre plus large rappelons-le, est ici celle qui emploi comme requête:



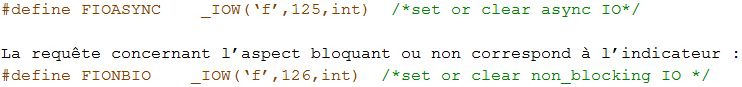
On peut donc passer en mode non bloquant mais selon des indicateurs variables selon le système hôte.

**La fonction ioctl** : contrôle des périphériques flux

Pour le contrôle des périphériques flux, il existe une fonction prototypée dans **stropts.h** et qui s’écrit :



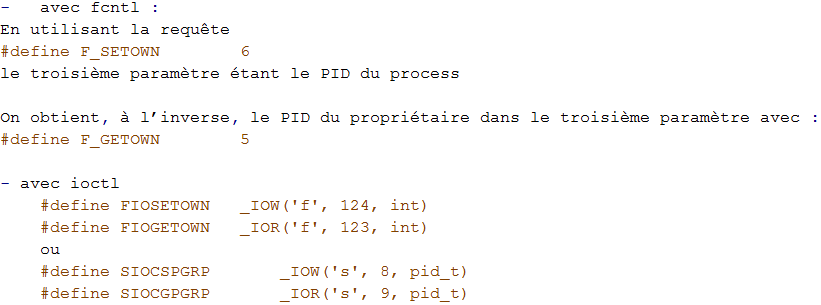
On passe en mode bloquant si le troisième paramètre est non nul. La requête de passage en mode asynchrone (permettant l’utilisation du signal SIGIO) est celle qui utilise :



* Différents paramétrages:

Etre le propriétaire d'une socket :

Il peut être utile qu'un processus se rende propriétaire d'une socket. Ceci lui permet ainsi de pouvoir être avisé de l'un ou l'autre signal (comme SIGURG ou SIGIO). L'opération peut se faire :



Obtenir des informations sur une socket :

La primitive à utiliser pour se faire une idée des caractéristiques d'une socket est prototypée dans **socket.h** et a pour forme :



Ainsi, dans le domaine **AF\_INET**, deux niveaux sont le plus fréquemment utilisés :

* le niveau socket : #define SOL\_SOCKET 0xffff
* le niveau protocole : les informations obtenues s'appliquent donc en correspondance avec le protocole spécifié au moyen des constantes définies dans netinet/in.h

Ex: IPPROTO\_IP, IPPROTO\_TCP ou IPROTO\_UDP

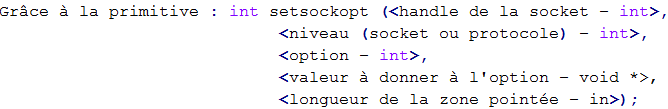
* booléenne : elle s'applique à la socket ou pas à l'instant considéré
* non booléenne : l'information reçue est donc cette fois une véritable valeur qui est celle d'une caractéristique de la communication dont la socket est chargée.



La nature du quatrième paramètre dépend de la nature du renseignement demandé tandis que le dernier paramètre contient la longueur de la zone recevant l’information: il est initialisé à la longueur attendue, mais rectifié éventuellement par la primitive sur la longueur de son résultat.

Cette primitive renvoie -1 en cas d’erreur et 0 en cas de succès.

Modifier les options d'une socket:



Celle-ci renvoie -1 en cas d’erreur et 0 en cas de succès.

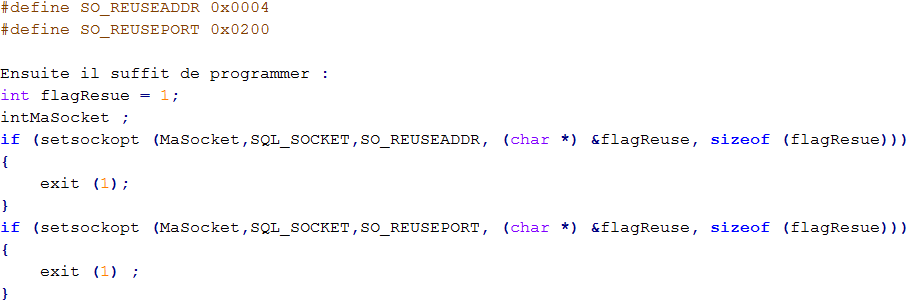
Exemple de paramétrage courant en UDP :

Pour pouvoir recevoir des datagrammes multicast, nous avons besoin d’une socket qui est liée à une adresse multicast définie (exemple : 234.5.5.9) et qui va se joindre au groupe multicast en utilisant à primitive setsockop :



Ici, on spécifie bien les paramètres nécessaires à savoir : la socket concernée, le protocole (UDP), la constante signifiant l’ajout au groupe multicast, mreq correspondant ici à l’adresse de multicast que l’on veut atteindre via une structure ip\_mreq qui contient dans son champ imr\_multiaddr le champ sin\_addr de la stuct sockaddr\_in paramétrée (adresse multicast, port) et enfin la taille de cette structure.

Néanmoins, dans l’exemple du multicast, la même adresse et le même port seront utilisés. Comme une adresse locale ne peut correspondre qu’à un seul socket, il faut contourner cela en, pour les sockets (ne marche que pour celle du domaine AF\_INET), positionnant l’option correspondante au moyen des constantes :



Exemple en TCP :

La constante #define

TCP\_MAXSEG 0x02 permet d’obtenir ou de diminuer la taille du segment. On ne peut utiliser que des sockets SOCK\_STREAM. Si la socket est connectée, on aura la valeur du MTU diminuée de la taille des en-têtes IP et TCP.

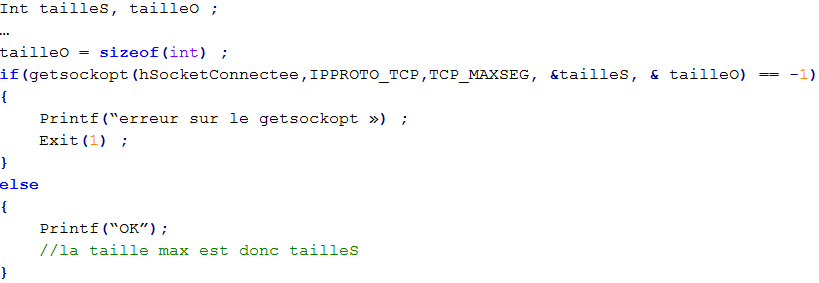


Table des matières

1. [Questions 1](#_bookmark0)
   1. [Expliquer, en vous aidant d’un schéma, les différentes étapes de la connexion d’un client TCP/IP à](#_bookmark1) [son serveur et comment les états de la machine à états de TCP évoluent. Quel rapport avec les primitives](#_bookmark1) [socket(), bind(), listen(), connect() et accept(), les queues de la structure socket, les flags SYN et ACK, la](#_bookmark1)

[« slidding windows », le TTL, la commande netstat ? Que se passe-t-il lors de la déconnexion de ce même](#_bookmark1) [client (états, flags, SYN et ACK) ? Expliquer en particulier les temps de latence qui sont explicitement](#_bookmark1) [prévus dans certaines transitions d’états. 1](#_bookmark1)

* 1. [Quelles sont les différentes manières de lancer et gérer un thread en Java ? Comment faire pour](#_bookmark2) [l'interrompre (de manière abrupte ou de manière plus contrôlée) ? Comment le problème des accès](#_bookmark2) [concurrents et celui de la synchronisation des threads sont-ils gérés en Java au moyen de monitors ?](#_bookmark2) [Quels sont les threads lancés automatiquement par une JVM Java ? 4](#_bookmark2)
  2. [Quelles sont les caractéristiques des communications réseaux utilisant le protocole UDP ?](#_bookmark3) [Caractériser avec du code concret les grands points de la programmation UDP/IP en C/Unix et en Java.](#_bookmark3) [Expliquer avec du code concret pourquoi le paramétrage des sockets peut être nécessaire dans ce type de](#_bookmark3) [programmation UDP (deux situations différentes au moins)? 6](#_bookmark3)
  3. [Expliquer avec des éléments de code concret comment (principes, adresses et ports, code de base)](#_bookmark4) [on peut réaliser un multicast de type chat avec des participants Java et C. En particulier, quel est le rôle](#_bookmark4) [(implicite ou explicite) du paramétrage des sockets ? 8](#_bookmark4)
  4. [Quelle est la démarche à suivre pour atteindre une base de données au moyen de JDBC et y pratiquer](#_bookmark5) [des mises à jour (dynamiques ou non) ? En quoi peut-on affirmer que l'on atteint un certain niveau de](#_bookmark5) [généricité et d'abstraction ? Comment peut-on réaliser les opérations de parcours aléatoire, d'ajout ou de](#_bookmark5) [suppression, de mise à jour ? Expliquer globalement ce que JPA apporte de plus en termes de](#_bookmark5) [renforcement de l'abstraction 9](#_bookmark5)
  5. [Expliquer la structure et le fonctionnement d'un serveur multithread générique utilisant un pool de](#_bookmark6) [threads et des objets Runnable. Comment procéder pour qu'il puisse gérer des conversations prolongées](#_bookmark6) [entre le serveur et les clients ? Comment faire pour que ce serveur générique ne risque pas d'exécuter](#_bookmark6) [n'importe quelle commande ? 11](#_bookmark6)
  6. [Décrire, à partir des répertoires et des différents tags des fichiers de configuration server.xml et](#_bookmark7) [web.xml de Tomcat, comment il réagit quand le client Web lui demande une page statique, l'exécution](#_bookmark7) [d'un CGI, l'exécution d'une servlet, l'exécution d'un JSP. Comment la réponse au client est-elle générée et](#_bookmark7) [envoyée selon les cas ? Expliquer en quoi il s'agit d'un serveur Web mais aussi d'un "moteur à servlets et à](#_bookmark7) [JSP". 13](#_bookmark7)
  7. [Quelles sont les principales caractéristiques du protocole HTTP ? A quoi ressemble classiquement un](#_bookmark8) [dialogue client-serveur en termes de trames (commandes, headers divers) ? Décrire, comme le ferait un](#_bookmark8) [sniffer, la trame HTTP classique encapsulant une page HTML demandée par un user-agent; en particulier,](#_bookmark8) [pourquoi parle-t-on de « négociation » et de « facteur q » ? Expliquer comment le problème de](#_bookmark8) [l'authentification est-il pris en charge par HTTP (Basic et Digest) 16](#_bookmark8)
  8. [Dans le contexte d'un serveur WEB, qu'est-ce qu'un CGI et qu'est-ce qu'une servlet ? Comparer ces](#_bookmark9) [deux technologies par rapport au traitement des formulaires classiques. A quoi ressemble le code d'un](#_bookmark9) [CGI basique qui répondrait à une simple sollicitation au moyen d'une page formulaire ? Expliquer à quoi](#_bookmark9) [ressemble le code de la servlet classique correspondante à ce CGI (packages, cycle de vie, principales](#_bookmark9) [variables et fonctions membres des classes usuelles, paramètres de ces dernières). 19](#_bookmark9)

##### [Quel est le problème principal d'un dialogue applet-servlet dans un contexte d'e-commerce où](#_bookmark10) [plusieurs clients réalisent chaun plusieurs requêtes ? Et comment est-il résolu explicitement](#_bookmark10) [(synchronized, cookies, réécriture d'URL, …) ou de manière plus abstraite (threads implicites, session, …) ?](#_bookmark10)

[22](#_bookmark10)

##### [Expliquer l'objectif, les diverses syntaxes (en les illustrant par un exemple) et le mécanisme des](#_bookmark11) [Java Server Pages (que se passe-t-il exactement quand un client accède à un JSP au moyen de son browser](#_bookmark11)

[**?). Pourquoi le mécanisme des balises à Java Beans est-il vite ressenti comme nécessaire ? Expliquer**](#_bookmark11)[**comment fonctionne ce mécanisme des balises pour Java Beans à partir d'un exemple complet.** 24](#_bookmark11)

* 1. [**Décrire le modèle MVC dans le contexte des servlets et des Java Server Pages. Quels sont les**](#_bookmark12)[**principes et méthodes (paramètres, codes, tags, …) à mettre en œuvre pour l'implémenter ? Décrire le**](#_bookmark12)[**protocole applicatif et son implémentation dans le cas classique d'un caddie virtuel. Donner une structure**](#_bookmark12)[**de base du code (principales variables membres, logique de base, transfert de pages, ...)** 26](#_bookmark12)
  2. [**Expliquer l'évolution historique de la notion de clé de chiffrement. En particulier, expliquer les**](#_bookmark13)[**termes de « substitution de caractères », « transposition », « chiffrements poly-alphabétiques », « grille**](#_bookmark13)[**de Vigenère », « substitution à répertoire », « chiffrement de blocs », « CBC », « padding », « clé**](#_bookmark13)[**publique », « clé secrète », « clé privée », « protocole de Diffie- Hellman », « DES », « RSA ». Que penser**](#_bookmark13)[**de l'affirmation : « les systèmes de chiffrement actuels sont forcément vulnérables, puisque les**](#_bookmark13)[**algorithmes qu'ils utilisent sont publics » ? Quels sont, parmi ces termes, ceux que l'on retrouve dans la**](#_bookmark13)

[**programmation des questions de chiffrement et d'authentification ?** 28](#_bookmark13)

* 1. [**Donner et expliquer une synthèse sous forme de tableau les différentes possibilités offertes par la**](#_bookmark14)[**cryptologie moderne en termes de confidentialité, intégrité, authentification, non répudiation, échange**](#_bookmark14)[**de clés (utiliser les notations K, PrK, PK, H, S, ). Pourquoi peut-on souvent distinguer une phase**](#_bookmark14)

[**« handshake » et une phase « communication » ? Quel est habituellement le rôle de la cryptographie**](#_bookmark14)[**symétrique et celui de la cryptographie asymétrique ?** 32](#_bookmark14)

* 1. [**Décrire les acteurs de la cryptographie symétrique : chiffrement, intégrité, authentification**](#_bookmark15)[**(utiliser les notations K, PrK, PK, H, S, …). Quel problème principal apporte l'utilisation d'une seule clé (clé**](#_bookmark15)[**de ssion) et comment peut-on le résoudre (deux techniques) ? Comment procède-t-on en programmation**](#_bookmark15)[**Java (quelles classes, obtenues comment, quelles méthodes, …) ? Expliquer les différences apportées par**](#_bookmark15)[**le chiffrement symétrique Rijndael/AES en CBC.** 35](#_bookmark15)
  2. [**Expliquer le principe du chiffrement asymétrique selon RSA (utiliser les notations K, PrK, PK, H, S,**](#_bookmark16)

[**…). En particulier, à quoi ressemblent les clés publiques et privées de cet algorithme ? Un tel chiffrement**](#_bookmark16)[**est-il réellement utilisé pour crypter l’intégralité d’une communication réseau sécurisée ? Comment**](#_bookmark16)[**procède-t-on en programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ?** 37](#_bookmark16)

* 1. [**Expliquer comment les notions d'interfaces et de méthodes factory sont utilisées en**](#_bookmark17)[**programmation Java dans le contexte de la cryptographie (chiffrement et authentification). Comment une**](#_bookmark17)[**méthode comme getInstance() fonctionne-t-elle (en particulier, comment trouve-t-elle les classes**](#_bookmark17)[**providers et comment peut-elle les instancier) ? Quel est le rôle du fichier java.security ?** 39](#_bookmark17)
  2. [**Expliquer la nature et la finalité d'un message digest. Comment on peut l'utiliser dans un contexte**](#_bookmark18)[**d'authentification se substituant au schéma classique login-password ? Comment procède-t-on en**](#_bookmark18)[**programmation Java (quelles classes, obtenues comment, quelles méthodes, …) ? Quel rapport avec les**](#_bookmark18)[**HMACs et comment ces derniers assurent-ils une authentification (légère) fiable (utiliser les notations K,**](#_bookmark18)[**PrK, PK, H, S, …) ?** 41](#_bookmark18)
  3. [**Qu'est-ce qu'une signature digitale (ou numérique ou électronique) et comment l'utiliser pour**](#_bookmark19)[**assurer l'intégrité et l'authentification (lourde) ? Qu'entend-on par "vérifier une signature" (utiliser les**](#_bookmark19)

##### [notations K, PrK, PK, H, S, …) ? Quelles sont les analogies et/ou les différences avec une signature](#_bookmark19) [manuscrite classique ? Comment procède-t-on en programmation Java (quelles classes, obtenues](#_bookmark19) [comment, quelles méthodes, …) ? Pourquoi le catalogue des fonctionnalités cryptographiques offertes par](#_bookmark19) [un provider ne citent-elles pas apparemment les algorithmes de cryptage asymétrique ? 43](#_bookmark19)

##### [Qu'entend-on par « certificat », « certificat X509 », « niveau (1-2-3,3+) » et « vérifier un certificat »](#_bookmark20)

##### [? A quoi correspondent les formats DER et PEM ? Qu'est-ce qu'un « keystore » ou « magasin à clés » ?](#_bookmark20) [Expliquer dans une liste des principales commandes comment les couples de clés et les certificats sont](#_bookmark20) [gérés au moyen de l'outil keytool. Comment procède-t-on en programmation Java (quelles classes, fichier](#_bookmark20) [et classe KeyStore, obtenues comment, quelles méthodes, …) ? 46](#_bookmark20)

##### [Expliquer les principes généraux de paramétrage des sockets (primitives, types de paramètres, …).](#_bookmark21) [Expliquer avec du code concret des situations courantes où il est nécessaire de paramétrer les sockets TCP](#_bookmark21) [et deux autres (différentes) pour les sockets UDP 47](#_bookmark21)

##### [Expliquer (utiliser les notations K, PrK, PK, H, S, …) ce que l'on entend par « authentification](#_bookmark22) [légère » (symétrique) et « authentification forte » (asymétrique). Comparer, notamment du point de vue](#_bookmark22)

##### [charge machine, fiabilité, difficultés d'échanges, programmation en Java 50](#_bookmark22)

1. [***Sources*** 52](#_bookmark23)