# **AA: Réseaux et technologies Internet Questions à préparer pour l’examen de théorie (2ème session 2022-2023)**

TCP/IP :

1. Expliquer la succession des appels de fonctions de la librairie des sockets en C à utiliser pour mettre en place une application serveur et une application client communiquant par TCP/IP. Décrire ce qui se passe à chaque appel (notamment en termes de machine à états et des handshakes de TCP, des files de demande de connexions, du pipe TCP, etc) depuis le moment où le client veut se connecter et que le serveur correspondant établit un canal de communication correspondant.

Voici comment fonctionnent les appels de sockets en C pour établir une communication entre un client et un serveur via TCP/IP :

1. Le client envoie une demande de connexion au serveur en utilisant la fonction connect().

2. Le serveur écoute les demandes de connexion entrantes en utilisant la fonction bind() pour lier une adresse IP et un numéro de port à sa socket, puis en utilisant la fonction listen() pour mettre sa socket en mode d'écoute.

Lorsqu'une demande de connexion est reçue, le serveur accepte la connexion en utilisant la fonction accept(). Cette fonction crée une nouvelle socket pour la communication avec le client et retourne un descripteur de fichier associé à cette socket.

Le client et le serveur échangent des données en utilisant les fonctions **send()** et **recv()**.

Au niveau de la structure socket, chaque socket a un état qui détermine son comportement et ses options de configuration. Lorsque le client envoie une demande de connexion au serveur, le serveur passe de l'état "écoute" à l'état "connecté".

Le protocole TCP utilise un "handshake" pour établir une connexion de bout en bout entre le client et le serveur. Le handshake consiste en trois échanges de messages :

1. Le client envoie un segment de SYN (synchronisation) au serveur, indiquant sa volonté de démarrer une connexion.
2. Le serveur envoie un segment de SYN-ACK (synchronisation-acquittement) au client, indiquant qu'il a reçu la demande de connexion et qu'il est prêt à démarrer la communication.
3. Le client envoie un segment de ACK (acquittement) au serveur, indiquant qu'il a reçu le segment de SYN-ACK et qu'il est prêt à démarrer la communication.

Les headers IP et TCP sont utilisés pour encapsuler les données envoyées entre le client et le serveur. Le header IP contient des informations sur l'adresse IP de l'expéditeur et du destinataire, ainsi que sur le protocole utilisé (TCP dans ce cas). Le header TCP contient des informations sur le numéro de séquence et d'acquittement, ainsi que sur les options de configuration du segment de données.

Le serveur utilise une file de demande de connexion pour gérer les demandes de connexion entrantes lorsqu'il est en mode d'écoute. Lorsqu'une demande de connexion est acceptée, un pipe TCP est créé entre le client et le serveur pour la communication de données.

2. Expliquer les caractéristiques des fonctions recv() et send() du C. Qu'est-ce que le MTU et quel rapport avec la fonction recv() ? Décrire les boucles de lecture qui tiennent compte de ce MTU.

Les fonctions **recv()** et **send()** sont utilisées pour la communication de données via le protocole TCP.

Elles permettent au client et au serveur de s'échanger des données de manière fiable et ordonnée.

La fonction **recv()** permet de recevoir des données envoyées par l'autre extrémité de la connexion TCP. Elle prend en entrée un descripteur de fichier associé à la socket, un tampon où stocker les données reçues, la longueur du tampon et un indicateur de fin de transmission. La fonction retourne le nombre de octets reçus, ou une valeur négative en cas d'erreur.

La fonction **send()** permet d'envoyer des données à l'autre extrémité de la connexion TCP. Elle prend en entrée un descripteur de fichier associé à la socket, un tampon contenant les données à envoyer, la longueur des données et un indicateur de fin de transmission. La fonction retourne le nombre de octets envoyés, ou une valeur négative en cas d'erreur.

Le MTU (Maximum Transmission Unit) désigne la taille maximale des segments de données pouvant être envoyés sur un réseau sans être fragmentés.

La fonction **recv()** prend en compte le MTU en limitant la taille des tampons pouvant être utilisés pour stocker les données reçues. Si le segment de données reçu est plus grand que le tampon, seule une partie des données sera stockée dans le tampon et il faudra effectuer plusieurs appels à la fonction **recv()** pour récupérer l'intégralité des données.

1. Voici comment cela peut être mis en œuvre dans le code :

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

2. Il est également possible de mettre en place une boucle de lecture qui continue à appeler la fonction recv() jusqu'à ce que l'intégralité des données attendues soit reçue. Pour ce faire, il faut connaître la taille totale des données à recevoir et utiliser un compteur pour suivre l'avancée de la réception :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Il est important de noter que le MTU peut varier selon le type de réseau utilisé et il peut être nécessaire de configurer les sockets en conséquence pour éviter les problèmes de fragmentation de données.

LES SERVEURS ET THREADS

3. Expliquer la structure programmatique C et le fonctionnement d'un serveur multi-client à la demande utilisant des threads. Quelles sont les modifications à apporter pour passer à un serveur multithread en pool de threads ?

Un serveur multi-client à la demande peut être implémenté de deux manières différentes en C : en utilisant des sous-processus ou en utilisant des threads.

La structure programmatique d'un serveur multi-client à la demande en utilisant des sous-processus ressemble à ceci :

1. Le serveur crée une socket et la met en mode d'écoute en utilisant les fonctions **bind()** et **listen()**.
2. Le serveur entre dans une boucle infinie pour accepter de nouvelles connexions.
3. Lorsqu'une connexion est acceptée, le serveur crée un nouveau processus en utilisant la fonction **fork()**.
4. Le processus enfant gère la communication avec le client en utilisant les fonctions **send()** et **recv()**, puis termine son exécution.
5. Le processus parent retourne à l'étape 2 pour accepter de nouvelles connexions.

La structure programmatique d'un serveur multi-client à la demande en utilisant des threads ressemble à ceci :

1. Le serveur crée une socket et la met en mode d'écoute en utilisant les fonctions **bind()** et **listen()**.
2. Le serveur entre dans une boucle infinie pour accepter de nouvelles connexions.
3. Lorsqu'une connexion est acceptée, le serveur crée un nouveau thread en utilisant la fonction **pthread\_create()**.
4. Le thread gère la communication avec le client en utilisant les fonctions **send()** et **recv()**, puis termine son exécution.
5. Le thread principal retourne à l'étape 2 pour accepter de nouvelles connexions.

La principale différence entre les deux approches réside dans la manière dont les tâches sont gérées. Dans le cas des sous-processus, chaque tâche est exécutée dans un processus séparé, ce qui nécessite la création de nouveaux processus et l'allocation de mémoire pour chaque tâche. Dans le cas des threads, chaque tâche est exécutée dans un thread séparé au sein du même processus, ce qui est moins gourmand en ressources système. Cependant, les threads partagent la mémoire et les ressources du processus d'origine, ce qui peut entraîner des problèmes de synchronisation et de concurrence si des ressources critiques doivent être accédées par plusieurs threads simultanément.

Pour passer d'un serveur multithread en pool de threads à un serveur multi-client à la demande, il faut ajouter une étape supplémentaire avant la création du thread : le serveur doit vérifier s'il y a des threads disponibles dans le pool avant de créer un nouveau thread. Si un thread est disponible, le serveur peut l'utiliser pour gérer la communication avec le client. Si aucun thread n'est disponible, le serveur peut soit attendre qu'un thread se libère, soit rejeter la connexion du client.

Il est important de noter que la gestion du pool de threads nécessite une synchronisation pour éviter les conflits d'accès aux threads disponibles. Cela peut être réalisé en utilisant des verrous ou des sémaphores pour protéger l'accès au pool de threads.

4. Expliquer en utilisant le code source de la classe Thread, les deux manières de "threader l'action principale" d'une classe. De quelle méthode dispose-t-on pour arrêter proprement le thread finalement lancé ?

En Java, vous pouvez utiliser la classe **Thread** pour créer et gérer des threads. Voici comment vous pouvez utiliser cette classe pour lancer l'action principale d'une classe à l'aide de deux méthodes différentes, ainsi que comment arrêter proprement un thread.

**Méthode 1 : En sous-classant Thread**

Une image contenant texte, capture d’écran

Description générée automatiquement

**Méthode 2 : En implémentant Runnable**

Une image contenant texte, capture d’écran

Description générée automatiquement

Dans la méthode 1, vous créez une sous-classe de **Thread** et redéfinissez la méthode **run()** avec le code que vous souhaitez exécuter dans le thread. Dans la méthode 2, vous implémentez l'interface **Runnable** et créez un objet **Thread** en passant une instance de votre classe implémentant **Runnable** au constructeur.

Pour arrêter proprement un thread en Java, il est généralement recommandé d'utiliser un mécanisme de signalisation pour indiquer au thread qu'il doit se terminer. Par exemple, vous pouvez utiliser un booléen partagé pour contrôler la boucle d'exécution du thread.

Une image contenant texte, capture d’écran

Description générée automatiquement

Dans cet exemple, le thread vérifie périodiquement la valeur du drapeau **shouldStop** dans sa boucle d'exécution. Lorsque la méthode **stopThread()** est appelée, le thread s'arrêtera proprement lorsqu'il vérifiera la condition **shouldStop**. Utiliser le modificateur **volatile** avec le booléen **shouldStop** garantit que les modifications de cette variable sont visibles entre les threads

5. Expliquer comment les mutex et variables de condition de la librairie C Posix des threads ont été transformés en Java pour aboutir à des paradigmes de programmation similaires (mais pas identiques). En particulier, expliquer la notion de « monitor ».

Les concepts de mutex (verrou) et de variables de condition trouvent leurs équivalents dans la programmation multithread en Java grâce à la notion de "monitor" et à l'utilisation de méthodes synchronisées et de mots-clés comme **synchronized**. Cependant, bien que les concepts soient similaires, il existe des différences subtiles dans la façon dont ils sont implémentés et utilisés entre les librairies POSIX des threads en C et la programmation multithread en Java.

**Mutex et verrous en C POSIX Threads :**

En C avec la librairie POSIX des threads (pthread), les mutex sont utilisés pour assurer l'accès exclusif à une ressource partagée par plusieurs threads. Un mutex est un verrou qui permet à un seul thread à la fois d'accéder à une ressource critique. La librairie pthread fournit des fonctions pour initialiser, verrouiller et déverrouiller les mutex.

Les variables de condition sont utilisées en conjonction avec les mutex pour permettre aux threads de se synchroniser et de communiquer. Une variable de condition permet à un thread d'attendre qu'une condition particulière soit satisfaite avant de continuer son exécution.

**Moniteurs et synchronisation en Java :**

En Java, le concept de "monitor" est utilisé pour gérer l'accès concurrentiel aux méthodes et aux blocs de code critiques. Un moniteur est essentiellement une façon de garantir que seule une seule tâche (thread) à la fois peut accéder à un bloc de code synchronisé. Java utilise le mot-clé **synchronized** pour indiquer qu'un bloc de code doit être exécuté de manière atomique par un seul thread à la fois.

En plus des moniteurs, Java fournit également des méthodes de synchronisation (**wait()**, **notify()**, **notifyAll()**) pour la gestion des variables de condition. Ces méthodes permettent aux threads de se synchroniser et de communiquer entre eux.

**Différences clés entre C POSIX Threads et Java :**

1. **Syntaxe et API :** La syntaxe et les fonctions pour travailler avec les mutex et les variables de condition en C POSIX Threads sont différentes de celles en Java. En C, vous avez des fonctions comme **pthread\_mutex\_lock()**, **pthread\_mutex\_unlock()**, **pthread\_cond\_wait()**, etc. En Java, vous utilisez **synchronized** et les méthodes **wait()**, **notify()**, **notifyAll()**.
2. **Exception handling :** En Java, les exceptions liées à la synchronisation sont gérées plus facilement grâce à la structure de gestion d'exceptions, alors qu'en C, vous devez gérer explicitement les erreurs de retour de fonctions.
3. **Niveaux d'abstraction :** Java abstrait davantage les détails de bas niveau en fournissant des mots-clés et des mécanismes plus simples pour la synchronisation, tandis que C POSIX Threads peut nécessiter une gestion plus manuelle des détails de bas niveau.
4. **Portabilité :** Java offre une meilleure portabilité grâce à sa machine virtuelle, tandis que les librairies POSIX des threads en C dépendent du système d'exploitation.

En résumé, bien que les concepts de mutex, de variables de condition et de moniteurs visent à résoudre les mêmes problèmes de concurrence et de synchronisation dans les programmes multithread, les détails de leur implémentation et de leur utilisation diffèrent entre les librairies C POSIX Threads et la programmation multithread en Java.

JDBC :

6. Qu'est-ce qu'un driver JDBC ? Pourquoi distingue-t-on 4 types ? Expliquer par quels mécanismes l'appel de Class.forName() permet d'instancier et de retrouver les drivers JDBC. Comment la connexion vers la base de données visée est obtenue au moyen de la chaîne de connexion ? Donnez un exemple de code d’accès (connexion + requête) à une base de données MySQL.

Tout d’abord, JDBC (Java DataBase Connectivity) est un ensemble d’APIs créé par SUN, permettant aux applications Java d’accéder par le biais d’une interface commune à des sources de données pour lesquelles il existe des drivers JDBC.   
Quel que soit la cible visée, nous communiquerons avec elle en SQL.   
Un driver JDBC permet de faire le lien entre notre application et une source de données.

Il existe en tout 4 types de driver :

1. Le Driver Java s’attaque à un autre Driver qui lui est Natif (c’est le cas pour ODBC par exemple), c’est-à-dire écrit en C.



1. Le driver java commence à toucher au SGBD, il s’occupe par exemple du concept de transactions mais le principal du travail est encore réalisé avec une librairie native. La différence avec le premier étant que le driver natif n’a pas une existence indépendante ici, il est calé sur l’autre. Ces deux premiers types ne sont pas portables contrairement aux deux types qui suivent. 
2. Le driver Java (driver full java) s’attaque à un SGBD qui offre les services d’un middleware, autrement dit il offre un front qui possède des normes, et ce front normalisé est utilisé par d’autres SGBD. Java n’a donc qu’à causer au middleware en question (c’est assez peu courant mais ça peut arriver de le croiser lorsqu’on utilise des BDD orientée objet par exemple)



1. Le type 4 est la version la plus courante, c’est un également un driver écrit en pure java mais il sait parfaitement à qui il s’adresse et n’a pas besoin de l’intermédiaire d’un Middleware pour communiquer avec un SGBD. C’est un driver dédié. (Perd toute généricité vu qu’il est au service d’un unique SGBD).

Aussi étonnant que cela puisse paraître, nous n’instancierons jamais directement de classe driver. Pour obtenir notre connexion, nous commencerons par utiliser une méthode statique de la classe « Class ». Cette méthode statique c’est « forName() », exemple d’utilisation :

  
 Cette méthode reçoit un driver comme paramètre mais elle n’instancie pas le driver en tant que tel, ce n’est pas une méthode factory. forName() charge la définition de la classe en mémoire.  
 Le bytecode qui contient la définition de la classe est mis en mémoire. Et lorsqu’on charge du code en mémoire, tout ce qui est statique est automatiquement exécuté/initialisé -> Les drivers contiennent une série de données et de blocs statiques.

C’est là qu’on voit apparaitre la seule classe de java.SQL, la classe DriverManager :

Cette classe est là pour gérer le fait que nous pourrions décider de jouer avec plusieurs drivers différents en même temps. Le DriverManager contient un container, une sorte de ArrayList où il place les références des drivers au fur et à mesure qu’ils sont chargés.

Dans DriverManager il existe également une méthode getConnection() qui prend en paramètre un URL (la chaine de connexion ici), et éventuellement un username et un password (ou NULL). Cette méthode va nous renvoyer un objet Connection.

L’objet Connection va nous permettre d’agir sur notre base de données, on peut par exemple :

-Refermer la connexion avec close().  
-Changer le mode de commit des transactions automatiques avec setAutoCommit(boolean) ;  
-Commit nous même avec commit()  
-Rollback avec la méthode rollback()  
-Et surtout, le plus important, créer un objet statement avec createStatement().  
C’est cet objet statement qui permet de créer et d’exécuter des instructions sur la base de données.

7. Expliquer l'architecture classique d'un serveur multithread basé sur le principe de la source de tâches et écrit en Java. Plus particulièrement, décrire les interfaces et les classes qui permettent de définir la structure classique de la partie réseaux d'une application multithreads. Selon ce schéma, comment une requête de connexion d'un client est-elle traitée ?

L'architecture classique d'un serveur multithread basé sur le principe de la source de tâches consiste à utiliser un modèle où chaque connexion entrante est gérée par un thread distinct. Cela permet au serveur de traiter simultanément plusieurs clients sans bloquer le traitement d'autres connexions. En Java, cela peut être réalisé en utilisant des classes telles que ServerSocket, Socket, et ThreadPoolExecutor.

Voici un aperçu des interfaces et classes impliquées dans la partie réseau d'une telle application multithread en Java :

1. **ServerSocket**: Cette classe permet au serveur d'écouter sur un port spécifique pour les connexions entrantes. Lorsqu'une connexion est établie, le ServerSocket accepte la connexion et retourne un objet Socket représentant la connexion.
2. **Socket**: Un objet Socket représente une connexion réseau entre le serveur et un client. Il fournit des flux d'entrée et de sortie pour envoyer et recevoir des données.
3. **Runnable**: L'interface Runnable doit être implémentée par toute classe dont les instances sont destinées à être exécutées par un thread. Elle définit une méthode run() qui contient le code exécuté par le thread.
4. **ThreadPoolExecutor**: Cette classe permet de gérer un pool de threads pour exécuter les tâches. Elle gère automatiquement la création et la réutilisation des threads, ce qui peut réduire les coûts associés à la création fréquente de nouveaux threads.

Voici comment le traitement d'une requête de connexion d'un client est généralement effectué dans cette architecture :

1. Le serveur crée un ServerSocket et le met en attente sur un port donné.
2. Lorsqu'une requête de connexion d'un client est reçue, le ServerSocket accepte la connexion et renvoie un objet Socket représentant cette connexion.
3. Le serveur crée une instance d'une classe qui implémente Runnable, en lui passant le Socket de la connexion. Cette instance encapsule la logique de traitement du client.
4. Le serveur soumet cette instance Runnable à un ThreadPoolExecutor pour qu'elle soit exécutée par l'un des threads du pool.
5. Le thread exécute la méthode run() de l'instance Runnable, ce qui effectue le traitement spécifique au client à travers le Socket.
6. Une fois que le traitement est terminé, le thread retourne dans le pool pour être réutilisé pour d'autres connexions entrantes.

En utilisant ce schéma, le serveur peut gérer plusieurs clients en même temps en les traitant dans des threads distincts, ce qui améliore les performances et la réactivité du serveur. Cependant, il est important de prendre en compte les problèmes liés à la concurrence et à la synchronisation lors de l'accès aux ressources partagées entre les threads.

8. Décrire le format des requêtes et des réponses HTTP. En particulier, expliquer cette phrase : "HTTP est un protocole sans état" ? Comment peut-on remédier à ce problème ?

HTTP (HyperText Transfer Protocol) est un protocole de communication utilisé pour échanger des informations sur le World Wide Web. Il permet de transférer du contenu HTML, du texte, des images et d'autres types de fichiers entre des ordinateurs connectés à Internet.

Dans HTTP, les requêtes et les réponses sont structurées selon un format standard qui comprend un ensemble de champs de métadonnées et un corps de message.

Les requêtes HTTP sont envoyées par le client (par exemple, un navigateur web) au serveur pour demander un certain contenu ou une action. Elles comprennent un verbe (par exemple, GET, POST, PUT, DELETE) qui indique l'action à effectuer, une URL (Uniform Resource Locator) qui indique l'emplacement du contenu ou de l'action, et des champs de métadonnées qui fournissent des informations sur la requête (par exemple, le type de contenu accepté par le client, le navigateur utilisé, etc.).

Les réponses HTTP sont envoyées par le serveur au client pour répondre à une requête. Elles comprennent un code de statut qui indique si la requête a réussi ou échoué, des champs de métadonnées qui fournissent des informations sur la réponse (par exemple, le type de contenu retourné, la date de dernière modification, etc.), et un corps de message qui contient le contenu demandé ou une erreur.

HTTP est un protocole orienté typage et négociation car il permet de spécifier le type de contenu attendu dans une requête et de négocier avec le serveur pour trouver un type de contenu qui convient à la fois au client et au serveur. Cela se fait en utilisant les champs de métadonnées "Accept" et "Content-Type" dans les requêtes et les réponses HTTP.

HTTP est un protocole sans états, ce qui signifie qu'il n'enregistre pas d'informations sur les requêtes et les réponses précédentes. Cela signifie que chaque requête et chaque réponse sont traitées indépendamment, sans tenir compte de celles qui ont précédé ou qui suivront. Cela peut poser des problèmes dans certaines situations, par exemple lorsque vous souhaitez maintenir l'état d'une session utilisateur ou lorsque vous voulez suivre l'historique de navigation d'un utilisateur.

Pour remédier à ce problème, il existe plusieurs solutions :

• Utiliser les cookies HTTP : les cookies sont des petits fichiers de données stockés sur l'ordinateur de l'utilisateur par le navigateur web. Ils permettent de stocker des informations sur les requêtes et les réponses HTTP et de les réutiliser lors de requêtes ultérieures.

• Utiliser les sessions HTTP : les sessions permettent de maintenir l'état d'une session utilisateur en utilisant une variable de session stockée sur le serveur. Lorsqu'un utilisateur accède au site web, le serveur génère une clé de session unique qui est envoyée au client sous forme de cookie HTTP. Toutes les requêtes ultérieures de l'utilisateur sont accompagnées de ce cookie, permettant au serveur de reconnaître l'utilisateur et de maintenir l'état de sa session.

• Utiliser des URL rewrite : les URL rewrite permettent de modifier l'URL de la requête de manière à inclure des informations sur l'état de la session ou de la requête. Par exemple, on peut ajouter un identifiant de session à l'URL pour maintenir l'état de la session. Cette technique est utilisée lorsque le navigateur de l'utilisateur ne supporte pas les cookies ou lorsqu'on souhaite masquer les informations d'état de la session.

9. Expliquer ce qu'est un digest (ou hachage), les propriétés qu'il doit posséder et comment il est construit. Qu’a de plus un digest « salé » ? Décrire les diverses situations informatiques où il est utilisé.

Un digest, également appelé hachage, condensé ou empreinte, est un code de longueur fixe généré à partir d'un message ou d'une donnée. Le digest est généralement utilisé pour vérifier l'intégrité et l'authenticité de la donnée d'origine, c'est-à-dire pour s'assurer qu'elle n'a pas été altérée ou modifiée.

Pour posséder ces propriétés, un digest doit répondre à certaines conditions :

• Unicité : pour chaque donnée d'origine, le digest doit être unique. Cela signifie qu'il ne doit pas y avoir deux données différentes qui produisent le même digest.

• Irréversibilité : il doit être impossible de retrouver la donnée d'origine à partir du digest. Cela signifie qu'il ne doit pas y avoir de méthode qui permette de décoder le digest et de retrouver la donnée d'origine.

• Robustesse : le digest doit être robuste face aux altérations de la donnée d'origine. Cela signifie que même si la donnée d'origine est légèrement modifiée, le digest doit être très différent de celui de la donnée d'origine

Les digest "salé" sont utilisés pour améliorer la sécurité des mots de passe stockés dans les systèmes informatiques. Il contient en plus une variable aléatoire(sel) qui sera concaténée à des données sensibles avant d’être hashée. Augmentant la sécurité des données.

Ils sont utilisés dans de nombreuses situations informatiques, notamment :

* + Stockage de mots de passe
  + Authentification
  + Gestion des identifiants
  + Stockage de données sensibles
  + Protection des mots de passe
  + Renforcement contre les attaques

10. Expliquer les 4 composantes de la cryptologie. Décrire les différents algorithmes relevant de ces composantes qui interviennent à l'heure actuelle dans un système de communication sécurisée par cryptologie symétrique.

La cryptologie est l'étude de la création, de l'utilisation et de la défense de systèmes de cryptage de données.

Elle se compose de quatre composantes principales :

- La confidentialité : s’assurer que seul le destinataire puisse lire le message en le rendant illisible par d’autres.

- Le non-répudiation : L’impossibilité pour une personne de nier avoir reçu ou émis un message.

- L’intégrité : s’assurer que le message n’a pas été modifié depuis son envoi.

- L’authentification : s’assurer que le message provient bien de l’expéditeur par une signature vérifiable.

Exemples d'algorithmes de cryptographie symétrique qui interviennent à l'heure actuelle dans les systèmes de communication sécurisés :

Le premier type d’algo qui nous vient naturellement en tête lorsqu’on parle d’algorithme de chiffrement symétrique c’est les algorithmes de de chiffrement, ici par substitution de caractères.

Une image contenant ligne, nombre, Parallèle, Rectangle

Description générée automatiquement

Ce genre d’algorithmes de cryptage sont dit « symétriques » parce que la clé qui a servi à crypter sert également à décrypter. (c’est le cas du codage de césar par exemple, où la clé est le décalage dans l’alphabet)   
Dans ce genre d’algorithme on retrouve une clé « secrète » (pas privée, ça c’est dans les algos asymétrique) notée K (comme key) (Kab si clé partagée par alice et bob)  
 Le souci avec les algorithmes de substitution de caractères classiques est que la longueur de la chaine en entrée est exactement le même que celle de la chaine de sortie et qu’un caractère est toujours remplacé par un même autre caractère. C’est déjà une information qui fragilise énormément ces algorithmes, si en plus on connait la langue du message, alors le code est cassé, nous n’aurions qu’à faire une analyse fréquentielle en cherchant les caractères qui se présentent le plus souvent.  
Un autre type d’algorithme important dans la famille des algos symétriques c’est les algos par transposition des caractères. L’idée c’est par exemple de permuter des couples de caractères selon une certaine clé dont la valeur est à interpréter selon l’algorithme précis. (ici la clé est de 3)

**Une image contenant ligne, Rectangle, carré, nombre

Description générée automatiquement**

Pour se faire comprendre il faut que les deux parties génèrent la même clé au même moment, c’est ainsi que sont apparus les algorithmes de générations de clés, la clé est générée de manière dynamique.

Un bon algorithme de génération de clé est quelque chose de relativement imprévisible. (ces algorithmes sont publics tout comme les algorithmes de chiffrement)  
Pour générer la clé on fait intervenir le hasard, ce qui rend le travail de l’hackeur extrêmement compliqué. Donc qui dit algorithme de génération de clés dit algorithme de génération de nombres aléatoire sur le côté.

Ensuite, les choses se sont encore compliquées lorsque les algorithmes de substitution et de transposition de caractères ont vu naître la notion de blocs.  
 Ces blocs il faut bien dire comment on les forme et comment ils sont chainés les uns aux autres, donc dans notre collection d’algorithmes viennent se rajouter les algorithmes de génération de blocs. Les blocs générés accueilleront les messages à coder mais il se peut cependant qu’un message ne remplisse pas complètement le bloc et qu’il y ait donc des cases vides qu’il faudra remplir.

Le truc c’est qu’on ne peut pas se contenter de remplir les cases vides d’un même caractère, ce serait donner une information précieuse aux hackeurs.

Il a donc fallut pondre un nouveau type d’algorithme, il s’agit de la famille des algorithmes de padding(remplissage) permettant de remplir la fin d’un blocs de caractères de remplissage qui ne soient pas détectables trop facilement.

Jusqu’ici on partait du principe que les deux parties de la communication cryptée était en possession de la clé mais en pratique cette hypothèse n’est pas forcément facile à satisfaire. Et le problème est que dans le cas de l’informatique par exemple, ce serait bien trop risqué de s’échanger la clé en clair sur le réseau.

C’est alors que son apparu les algorithmes d’échange de clés, ce n’est cela dit pas toujours un échange de clé qui est réalisé, en réalité c’est le plus souvent une génération simultanée de la clé des deux côtés qui est opéré (avec Diffie-Hellman).

11. Qu'est-ce qu'une clé de session symétrique ? Comment procéder pour que les deux parties puissent en disposer de manière sûre ?

Une clé de session symétrique est une clé utilisée pour chiffrer et déchiffrer des données de manière sécurisée lors d'une communication entre deux parties. Elle est généralement utilisée dans le cadre d'un échange de données à court terme, comme lors d'une session de chat en ligne ou d'une transaction bancaire en ligne.

Il existe plusieurs moyens de procéder pour que les deux parties puissent disposer de manière sûre d'une clé de session symétrique :

1. Échange de clé publique/privée : l'une des parties peut générer une paire de clés publique/privée et envoyer sa clé publique à l'autre partie. La seconde partie peut alors utiliser la clé publique pour chiffrer la clé de session symétrique et l'envoyer à la première partie, qui peut la déchiffrer avec sa clé privée. Cette méthode permet de s'assurer que seules les deux parties ont accès à la clé de session symétrique, mais nécessite l'utilisation de clés publiques/privées, qui peuvent être complexe à mettre en place.

2. Échange de mot de passe : les deux parties peuvent échanger un mot de passe et utiliser celui-ci pour générer la clé de session symétrique à l'aide d'une fonction de hachage. Cette méthode est plus simple que l'échange de clés publique/privée, mais elle nécessite que les deux parties échangent le mot de passe de manière sécurisée, ce qui peut être difficile à réaliser. Il est donc recommandé de ne pas utiliser cette méthode si des données sensibles sont en jeu.

Il est important de choisir une clé de session symétrique qui est assez longue et aléatoire pour éviter qu'elle ne soit devinée ou crackée par un tiers. Il est également recommandé de changer régulièrement la clé de session symétrique pour renforcer la sécurité de la communication.

12. Qu'est qu'un Cryptography Service Provider (csp) ? Comment est-il représenté en programmation Java ? Expliquer (avec le code Java) le fonctionnement d'une factory permettant d'obtenir un générateur de clés et un objet de chiffrement symétrique quel que soit l'environnement d'exécution.

Un Cryptography Service Provider (CSP) est un module logiciel qui fournit des services de chiffrement à d'autres programmes. Il peut être utilisé pour gérer les clés de chiffrement, effectuer des opérations de chiffrement et de déchiffrement, et fournir des fonctionnalités de signature numérique.

En Java, un CSP est représenté par l'interface javax.crypto.CryptoPrimitive et par l'interface javax.crypto.CipherSpi qui est une classe abstraite qui définit les méthodes de chiffrement et de déchiffrement.

Pour obtenir un générateur de clés et un objet de chiffrement symétrique quel que soit l'environnement d'exécution, vous pouvez utiliser une factory telle que javax.crypto.KeyGenerator et javax.crypto.Cipher.

Voici un exemple de code Java qui montre comment utiliser ces classes pour chiffrer et déchiffrer un message :

Une image contenant texte, capture d’écran

Description générée automatiquement

Dans ce code, nous utilisons la factory KeyGenerator pour générer une clé secrète de type AES à 128 bits. Nous utilisons la factory Cipher pour créer un objet de chiffrement symétrique qui utilise l'algorithme AES en mode ECB avec un padding PKCS5. Nous utilisons ensuite cet objet pour chiffrer et déchiffrer un message.

13. Expliquer le principe de l'échange d'un message chiffré symétriquement ou asymétriquement. Décrire le code Java permettant de chiffrer un message avec un algorithme de chiffrement symétrique ou asymétrique. Comparer les deux procédés : quelles similitudes, quelles différences ? En particulier, quelles sont les clés de chiffrement manipulées ?

Voici comment fonctionne l'échange d'un message chiffré symétriquement ou asymétriquement : Pour chiffrer un message symétriquement :

1. L'expéditeur et le destinataire échangent une clé secrète K de manière sécurisée (par exemple, en utilisant un échange de clés asymétrique).

2. L'expéditeur utilise la clé K pour chiffrer le message M en utilisant un algorithme de chiffrement symétrique, obtenant ainsi le message chiffré C.

3. L'expéditeur envoie le message chiffré C au destinataire. 4. Le destinataire utilise la clé K pour déchiffrer le message C en utilisant le même algorithme de chiffrement symétrique, obtenant ainsi le message M déchiffré.

Voici un exemple de code Java qui montre comment chiffrer un message symétriquement :

Une image contenant texte, capture d’écran

Description générée automatiquement

Pour chiffrer un message asymétriquement :

1. L'expéditeur génère une paire de clés publique-privée PK, PrK.

2. L'expéditeur envoie sa clé publique PK au destinataire.

3. Le destinataire utilise la clé publique PK de l'expéditeur pour chiffrer le message M en utilisant un algorithme de chiffrement asymétrique, obtenant ainsi le message chiffré C.

4. Le destinataire envoie le message chiffré C à l'expéditeur.

5. L'expéditeur utilise sa clé privée PrK pour déchiffrer le message C en utilisant le même algorithme de chiffrement asymétrique, obtenant ainsi le message M déchiffré.

Voici un exemple de code Java qui montre comment chiffrer un message asymétriquement :

Une image contenant texte, capture d’écran

Description générée automatiquement

Dans ce code, nous utilisons la factory KeyPairGenerator pour générer une paire de clés publiqueprivée de type RSA à 2048 bits. Nous récupérons la clé publique et la clé privée de l'expéditeur. Nous utilisons la factory Cipher pour créer un objet de chiffrement asymétrique qui utilise l'algorithme RSA en mode ECB avec un padding PKCS1. Nous utilisons ensuite cet objet pour chiffrer le message en utilisant la clé publique de l'expéditeur.

Il y a plusieurs différences entre le chiffrement symétrique et asymétrique :

• Dans le chiffrement symétrique, la même clé est utilisée pour chiffrer et déchiffrer le message. Dans le chiffrement asymétrique, une clé publique et une clé privée différentes sont utilisées pour chiffrer et déchiffrer le message.

• Le chiffrement symétrique est généralement plus rapide que le chiffrement asymétrique, mais il est moins sécurisé car la clé doit être partagée entre les deux parties. Le chiffrement asymétrique est plus lent, mais il est plus sécurisé car la clé privée n'a pas besoin d'être partagée.

Un vecteur d'initialisation (VI) est un paramètre utilisé dans certains algorithmes de chiffrement pour initialiser l'état du chiffrement. Il est généralement utilisé dans les algorithmes de chiffrement en mode de chiffrement à bloc (par exemple, AES en mode ECB ou CBC).

Le VI est généralement choisi aléatoirement et doit être partagé entre l'expéditeur et le destinataire afin que le message puisse être déchiffré correctement. Il est utilisé pour rendre le chiffrement plus sécurisé en empêchant les attaques de type "rejouabilité" ou "modification de données".

14. Comment le modèle MVC peut-il être implémenté dans une application WEB Java basée sur les servlets, les JSPs et les Java beans ? En particulier, expliquer les rôles du ServletContext. Illustrer ces explications sur un caddie virtuel exemple élémentaire.

Le modèle MVC (Model-View-Controller) est un modèle de conception couramment utilisé dans les applications web pour séparer les différentes couches de l'application en vue de faciliter le développement et la maintenance du code.

Dans le modèle MVC, le modèle (Model) représente les données de l'application et les règles de gestion de ces données, la vue (View) est ce que l'utilisateur voit et interagit avec, et le contrôleur (Controller) gère la communication entre le modèle et la vue et traite les actions de l'utilisateur.

Une image contenant texte, capture d’écran, diagramme, conception

Description générée automatiquement

Pour implémenter le modèle MVC dans une application web Java basée sur les servlets, les JSPs (Java Server Pages) et les Java beans, on peut utiliser les servlets comme contrôleurs, les JSPs comme vues et les Java beans comme modèles. Voici comment cela peut être mis en œuvre dans le contexte d'un caddie virtuel élémentaire :

1. Le modèle (Java bean) représente les données du caddie virtuel, comme la liste des produits et le montant total.

2. La vue (JSP) affiche l'interface utilisateur du caddie virtuel, qui permet à l'utilisateur de voir les produits dans son caddie et de changer leur quantité ou de les supprimer.

3. Le contrôleur (servlet) gère les actions de l'utilisateur, comme l'ajout ou la suppression de produits dans le caddie. Il utilise le modèle pour mettre à jour les données du caddie et envoie la vue actualisée à l'utilisateur.

Le ServletContext joue le rôle de conteneur de servlets et de JSPs dans une application web. Il permet de partager des informations et des ressources entre les servlets et les JSPs, comme une base de données ou un objet de configuration.

Dans le contexte d'un caddie virtuel, le ServletContext pourrait par exemple être utilisé pour stocker les données du modèle de manière qu'elles soient accessibles à tous les servlets et JSPs de l'application.

15. Expliquer le mécanisme d'authentification par signature. En particulier, pourquoi parle-t-on d'authentification "lourde" et "légère" pour respectivement les HMAC et les signatures ?

La signature est un mécanisme d'authentification qui permet de s'assurer de l'intégrité et de l'authenticité d'un message. Elle est utilisée pour vérifier que le message n'a pas été modifié en transit et qu'il a bien été envoyé par une personne possédant une clé secrète.

Voici comment fonctionne la signature :

1. L'expéditeur génère une paire de clés publique et privée (PK et PrK). La clé publique est rendue publique (par exemple, en la publiant sur un annuaire ou en l'incluant dans un certificat numérique) tandis que la clé privée reste secrète.

2. L'expéditeur envoie un message M au destinataire.

3. L'expéditeur calcule la signature du message M en utilisant sa clé privée PrK et en concaténant le message M et la signature obtenue, obtenant ainsi l'expression (M | signature(PrK, M)).

4. Le destinataire reçoit le message M et la signature envoyée par l'expéditeur.

5. Le destinataire vérifie la signature du message M reçu en utilisant la clé publique PK de l'expéditeur et le compare à la signature envoyée par l'expéditeur. Si les deux signatures sont égales, cela signifie que le message n'a pas été modifié en transit et est donc considéré comme authentique.

Les HMAC (Hashed Message Authentication Code) sont considérés comme une forme d'authentification "lourde" car ils nécessitent l'utilisation de clés secrètes partagées entre les deux parties qui communiquent et impliquent le calcul de hashage d'un message à l'aide d'une fonction de hashage spécifique. Cela peut être relativement coûteux en termes de ressources informatiques, en particulier si les messages sont de grande taille ou si de nombreuses authentifications doivent être effectuées.

Les signatures, en revanche, sont considérées comme une forme d'authentification "légère" car elles ne nécessitent pas l'utilisation de clés secrètes partagées et ne nécessitent pas de calcul de hashage. Au lieu de cela, une signature est générée à l'aide d'une clé privée et vérifiée à l'aide d'une clé publique associée. Cela rend le processus de signature moins coûteux en termes de ressources informatiques et plus facile à mettre en œuvre que les HMAC.

Les HMAC sont considérés comme plus sûr mais plus complexe et coûteux à mettre en place qu’une signature.

16. Expliquer le principe et le code java du mécanisme d'authentification d'un utilisateur par digest.

Le principe de l'authentification d'un utilisateur par digest consiste à s'assurer que l'utilisateur qui se connecte au serveur est bien celui qu'il prétend être. Pour ce faire, le serveur peut demander à l'utilisateur de fournir un nom d'utilisateur et un mot de passe. Le serveur peut alors vérifier que le nom d'utilisateur et le mot de passe correspondent à ceux enregistrés dans sa base de données.

Pour renforcer la sécurité de cette authentification, le serveur peut utiliser un algorithme de hachage (par exemple, MD5 ou SHA-256) pour "casser" le mot de passe en un empreinte (appelée "digest"). Le serveur peut alors stocker l'empreinte du mot de passe dans sa base de données et comparer l'empreinte envoyée par l'utilisateur avec celle enregistrée dans la base de données.

Cela permet de s'assurer que le mot de passe n'est pas stocké en clair dans la base de données et donc de protéger l'utilisateur en cas de fuite de données.

Voici un exemple de code Java qui montre comment implémenter l'authentification d'un utilisateur par digest :

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Dans ce code, nous demandons le nom d'utilisateur et le mot de passe à l'utilisateur. Nous utilisons la classe MessageDigest pour calculer l'empreinte du mot de passe en utilisant l'algorithme SHA-256. Nous envoyons le nom d'utilisateur et l'empreinte du mot de passe au serveur. Le serveur vérifie alors que le nom d'utilisateur et l'empreinte du mot de passe correspondent à ceux enregistrés dans sa base de données et renvoie une réponse au client indiquant si l'authentification a réussi ou non.

17. Expliquer ce qu'est un certificat. En décrire le contenu et les différentes caractéristiques (formats, niveaux, fichiers pem/der/cer et leur encoding). Expliquer comment on vérifie un certificat et pourquoi on parle de "chaîne de certificats". Expliquer le mécanisme d'une CSR.

Un certificat est un document électronique qui sert à vérifier l'identité d'une entité (généralement une personne ou une organisation) et à établir une confiance dans ses communications avec d'autres parties.

Les certificats sont souvent utilisés pour sécuriser les communications sur Internet, notamment dans le cadre de la mise en place de connexions chiffrées (comme les connexions HTTPS) et de l'authentification de serveurs et de clients.

Le contenu d'un certificat varie en fonction de son utilisation, mais il peut généralement inclure les informations suivantes :

* Le nom de l'entité pour laquelle le certificat est délivré (par exemple, le nom d'une personne ou d'une organisation)
* L'adresse de l'entité (généralement une adresse e-mail ou une adresse Web)
* La période de validité du certificat (c'est-à-dire la période pendant laquelle le certificat est

considéré comme valide et fiable)

* L'identité de l'autorité de certification (CA) qui a émis le certificat
* La signature numérique de l'autorité de certification, qui permet de vérifier l'intégrité du

certificat et de s'assurer qu'il n'a pas été modifié depuis sa création

Il existe plusieurs formats de certificats, notamment :

- Les fichiers PEM (Privacy Enhanced Mail) sont des fichiers texte encodés en base64 qui contiennent le certificat en question, ainsi que tous les certificats de la chaîne de confiance (voir

ci-dessous). Les fichiers PEM sont souvent utilisés sur les serveurs Web pour stocker les certificats SSL/TLS.

- Les fichiers DER (Distinguished Encoding Rules) sont des fichiers binaires qui contiennent le certificat en question. Ils sont souvent utilisés pour stocker des certificats sur des appareils tels que des téléphones portables ou des clés de chiffrement.

- Les fichiers CER (ou CRT) sont similaires aux fichiers PEM et sont également utilisés pour stocker les certificats SSL/TLS sur les serveurs Web.

Il existe également différents niveaux de certificats, qui déterminent le niveau de confiance accordé à l'entité pour laquelle le certificat est délivré. Les certificats de niveau supérieur sont généralement considérés comme plus fiables que les certificats de niveau inférieur, car ils sont émis par des autorités de certification (CA) qui ont subi un processus de vérification plus rigoureux.

Pour vérifier la validité d'un certificat, il est nécessaire de vérifier la chaîne de certificats ( ou "chaine de confiance") associée à celui-ci. La chaîne de certificats est une série de certificats qui sont liés entre eux par des signatures numériques. Elle commence avec le certificat de l'entité pour laquelle le certificat est délivré (appelé "certificat racine") et se termine avec le certificat de l'autorité de certification qui a émis le certificat racine. Pour vérifier la validité d'un certificat, il faut donc vérifier que chaque certificat de la chaîne est valide et qu'il a été émis par l'autorité de certification qui lui est associée.

Pour vérifier un certificat, il est généralement nécessaire de disposer de la chaîne de certificats complète et de disposer de la clé publique de l'autorité de certification qui a émis le certificat. En utilisant ces informations, il est possible de vérifier la signature numérique de chaque certificat de la chaîne et de s'assurer qu'il n'a pas été modifié depuis sa création.

Une demande de signature de certificat (CSR) est un document qui est généré par une entité (généralement une personne ou une organisation) qui souhaite obtenir un certificat électronique. La CSR contient les informations requises pour générer le certificat, telles que le nom de l'entité, son adresse e-mail ou son adresse Web, ainsi que la clé publique associée. La CSR est envoyée à une autorité de certification, qui utilise ces informations pour générer et signer le certificat. Une fois le certificat signé, il est envoyé à l'entité qui a demandé le certificat, qui peut alors l'utiliser pour sécuriser ses communications.

18. Expliquer ce qu'est un fichier keystore et la classe Keystore. Quelles sont les principales manipulations dont il peut faire l'objet au moyen de l'outil keytool ? Pourquoi les keystores habituels ne contiennent-ils que des clés asymétriques ?

Un fichier keystore est un fichier qui permet de stocker et de gérer des clés et des certificats électroniques de manière sécurisée. Les keystores sont souvent utilisés pour stocker les clés et les certificats nécessaires à l'authentification et au chiffrement de communications sur Internet, comme les clés SSL/TLS utilisées pour sécuriser les connexions HTTPS.

La classe KeyStore est une classe Java qui permet de gérer un fichier keystore et de manipuler son contenu. Elle permet de charger un keystore à partir d'un fichier, de vérifier les clés et les certificats qu'il contient et de les utiliser pour sécuriser les communications.

L'outil keytool est un utilitaire en ligne de commande qui permet de gérer les keystores et leur contenu. Il permet notamment de créer et de gérer des keystores, d'importer et d'exporter des clés et des certificats, de générer des demandes de signature de certificat (CSR) et de vérifier les certificats et les clés.

Les keystores habituels ne contiennent généralement que des clés asymétriques, car ces clés sont généralement plus sûres et plus flexibles que les clés symétriques. Les clés asymétriques sont des clés qui sont composées de deux parties : une clé publique et une clé privée.

La clé publique peut être partagée avec d'autres parties et utilisée pour chiffrer des données, tandis que la clé privée doit être conservée secrète et utilisée pour déchiffrer les données.

Les clés asymétriques permettent donc de sécuriser les communications de manière plus efficace que les clés symétriques, qui doivent être partagées entre les parties impliquées et sont donc moins sûres.

19. Décrire les principales caractéristiques de la programmation des communications UDP : sockets C et Java, schéma classique d'un "serveur" et d'un "client", quasi-nécessité d'un timeout, nécessité d'"étiqueter" les requêtes et les réponses, … Dans quels domaines ce protocole de transport est-il particulièrement utile ?

UDP (User Datagram Protocol) est un protocole de transport qui permet de transmettre des données sur un réseau de manière non fiable et sans connexion. Voici quelques-unes des principales caractéristiques de la programmation des communications UDP : • Sockets : pour utiliser UDP dans un programme, il faut créer un socket UDP en utilisant la fonction socket() du système d'exploitation. En Java, le socket UDP peut être créé en utilisant la classe DatagramSocket. • Serveur et client : dans un schéma classique de communication UDP, il y a généralement un serveur qui écoute les requêtes et un client qui envoie des requêtes au serveur. Le serveur utilise une boucle infinie pour attendre les requêtes et le client utilise une boucle infinie pour envoyer régulièrement des requêtes au serveur. • Timeout : en raison de la nature non fiable de UDP, il est quasi-nécessaire d'utiliser un timeout pour éviter d'attendre indéfiniment une réponse. Il existe deux méthodes couramment utilisées pour implémenter un timeout : la méthode à base de compteur et la méthode à base de temporisation. • Méthode à base de compteur : dans cette méthode, le client envoie une requête au serveur et incrémente un compteur à chaque itération de la boucle de requête. Si le compteur atteint un certain seuil avant de recevoir une réponse du serveur, cela signifie que le timeout est déclenché et que la requête est abandonnée. Cette méthode est simple à mettre en œuvre, mais elle peut être moins précise que la méthode à base de temporisation si le délai de traitement varie beaucoup d'une requête à l'autre.

• Méthode à base de temporisation : dans cette méthode, le client envoie une requête au serveur et démarre un temporisateur. Si le temporisateur expire avant de recevoir une réponse du serveur, cela signifie que le timeout est déclenché et que la requête est abandonnée. Cette méthode est plus précise que la méthode à base de compteur, mais elle nécessite l'utilisation d'un mécanisme de temporisation, comme une fonction sleep() ou un timer. Cela peut nécessiter l'utilisation de threads supplémentaires pour gérer la temporisation en parallèle avec l'envoi et la réception de requêtes.  Il n'y a pas de méthode de timeout "meilleure" en général, mais la méthode à utiliser dépend des contraintes et des besoins de chaque situation. Par exemple, si le délai de traitement est relativement constant, la méthode à base de compteur peut être suffisante. Si le délai de traitement varie beaucoup d'une requête à l'autre, la méthode à base de temporisation peut être plus adaptée.

• Étiquetage des requêtes et des réponses : pour s'assurer que les requêtes et les réponses correspondent bien, il est nécessaire de les étiqueter avec un identifiant unique (par exemple, un numéro de séquence). Cela permet de s'assurer que les réponses sont associées aux bonnes requêtes même si plusieurs requêtes sont en cours de traitement simultanément.

UDP est particulièrement utile dans les domaines où la fiabilité est moins importante que la vitesse de transmission, comme la diffusion de données en temps réel (par exemple, la diffusion de vidéo en streaming). UDP est également souvent utilisé pour envoyer des données de manière non fiable dans des environnements à faible latence, comme les réseaux de jeux en ligne ou les réseaux de contrôle industriel.

20. Expliquer les caractéristiques du multicast basé sur les adresses de classe D et son implémentation au moyen du langage Java.

Le multicast est une technique de transmission de données qui permet à un expéditeur de diffuser des données à un groupe de destinataires de manière efficace. Le multicast basé sur les adresses de classe D est une forme de multicast qui utilise des adresses IP spéciales appartenant à la plage de classe D (224.0.0.0 à 239.255.255.255).

Voici quelques caractéristiques du multicast basé sur les adresses de classe D :

• La diffusion est gérée par le réseau : lorsqu'un expéditeur envoie des données en multicast, il n'a pas besoin de connaître l'ensemble des destinataires. Le réseau s'occupe de transmettre les données aux destinataires qui ont demandé à recevoir les données en multicast.

• Les données sont transmises de manière efficace : le multicast permet de diffuser des données de manière efficace, car les données ne sont transmises qu'une seule fois, même si elles sont destinées à plusieurs destinataires. Cela peut être particulièrement utile dans les réseaux de grande taille où les données doivent être transmises à de nombreux destinataires.

Voici pour implémenter le multicast basé sur les adresses de classe D dans les langages C et Java :

• En C : pour implémenter le multicast en C, il faut d'abord créer un socket multicast en utilisant la fonction socket() et en spécifiant l'option SOCK\_DGRAM pour indiquer qu'il s'agit d'un socket UDP. Ensuite, il faut utiliser la fonction bind() pour lier le socket à une adresse multicast de classe D et la fonction sendto() pour envoyer des données en multicast. Pour recevoir des données en multicast, il faut utiliser la fonction recvfrom().

• En Java : pour implémenter le multicast en Java, il faut utiliser la classe MulticastSocket qui étend la classe DatagramSocket et ajoute les fonctions nécessaires pour envoyer et recevoir des données en multicast. Pour envoyer des données en multicast, il faut créer une instance de MulticastSocket, spécifier l'adresse multicast de classe D à laquelle envoyer les données et utiliser la méthode send() de MulticastSocket. Pour recevoir des données en multicast, il faut créer une instance de MulticastSocket, joindre le groupe multicast en utilisant la méthode joinGroup() et utiliser la méthode receive() de MulticastSocket pour recevoir les données.

Il est important de noter que le multicast basé sur les adresses de classe D nécessite la configuration du réseau pour acheminer les données en multicast aux destinataires. Cela peut être réalisé en utilisant un routeur multicast ou en configurant les passerelles du réseau pour acheminer les données en multicast.