CHAPITRE 1 – PROGRAMMATION CLIENT-SERVEUR

PLAN DU CHAPITRE

- Programmation TCP simple (mono-thread)
- Programmation UDP simple (mono-thread)
- Méthodes supplémentaires
- Programmation multi-threads

1.1 – PROGRAMMATION TCP SIMPLE (MONO-THREAD)

PARADIGME CLIENT-SERVEUR

- La programmation réseau se base sur le paradigme client-serveur
  - Le serveur attend qu'un (ou plusieurs) client(s) se connecte à lui
  - Quand un client se connecte, une connexion est établie
  - Le client et le serveur peuvent alors échanger des données
- Remarques :
  - Le client est toujours plus simple à programmer
  - Il est très utile d'avoir un serveur opérationnel pour tester le client (par exemple l'outil « netcat »), puis un client opérationnel pour tester le serveur (cela peut être l'outil « netcat » ou « telnet »)

1.1.1 – DÉTAIL DU CLIENT TCP

STRUCTURE DU CLIENT

```java
import java.net.*;
import java.io.*;

public class SimpleTCPClient {

    private Socket socket;
    private BufferedReader input;
    private PrintWriter output;

    public SimpleTCPClient(String name, int port) {
        // …
    }
```

```java
    private void sendHTTPLines(String name) {
        // …
    }

    private void readHTTPLines() {
        // …
    }

}
```

DÉTAIL DU CONSTRUCTEUR

```java
public SimpleTCPClient(String name, int port) {
    try {
        socket = new Socket(name, port);
        input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        output = new PrintWriter(socket.getOutputStream(), true);
        sendHTTPLines(name);
        readHTTPLines();
        socket.close();
    }
    catch (UnknownHostException e) {
        System.out.println("Error: host "+name+" not found");
        e.printStackTrace();
    }
    catch (IOException e) {
        System.out.println("Error: I/O exception");
        e.printStackTrace();
    }
}
```

DÉTAIL DE L'ENVOI ET DE LA RÉCEPTION

```
private void sendHTTPLines(String name) {
    output.println("GET / HTTP/1.0");
    output.println("Host: "+name);
    output.println("Accept-Encoding: identity");
    output.println();
}

private void readHTTPLines() {
    String line;
    try {
        while ((line=input.readLine())!=null) {
            System.out.println("Recv: "+line);
        }
    }
    catch (IOException e) {
        System.out.println("Error: I/O exception");
        e.printStackTrace();
    }
}
```

EXEMPLE D'EXÉCUTION

```
Recv: HTTP/1.1 302 Found
Recv: Date: Tue, 27 Jun 2023 08:30:29 GMT
Recv: Server: Apache
Recv: Location: https://www.uca.fr/
Recv: Content-Length: 203
Recv: Connection: close
Recv: Content-Type: text/html; charset=iso-8859-1
Recv:
Recv: <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
Recv: <html><head>
Recv: <title>302 Found</title>
Recv: </head><body>
Recv: <h1>Found</h1>
Recv: <p>The document has moved <a href="https://www.uca.fr/">here</a>.</p>
Recv: </body></html>
```

1.1.2 – DÉTAIL DU SERVEUR TCP

STRUCTURE DU SERVEUR

```
import java.io.*;
import java.net.*;

public class SimpleTCPServer {

private ServerSocket serverSocket;

public SimpleTCPServer(int port) {
    // …
}

private void readHTTPLines(Socket clientSocket) {
    // …
}

private void sendHTTPLines(Socket clientSocket) {
    // …
}

}
```

DÉTAIL DU CONSTRUCTEUR

```
public SimpleTCPServer(int port) {
    try {
        serverSocket = new ServerSocket(port);
        Socket clientSocket = serverSocket.accept();
        readHTTPLines(clientSocket);
        sendHTTPLines(clientSocket);
        clientSocket.close();
        serverSocket.close();
    }
    catch (IOException e) {
        System.out.println("Error: server socket cannot be bound on port "+port);
        e.printStackTrace();
    }
}
```

DÉTAIL DE LA RÉCEPTION

```
private void readHTTPLines(Socket clientSocket) {
    try {
        BufferedReader input = new BufferedReader(
            new InputStreamReader(clientSocket.getInputStream()));
        boolean end = false;
        while (!end) {
            String line = input.readLine();
            System.out.println("Recv: "+line);
            if (line.equals("")) {
                end = true;
            }
        }
    }
    catch (IOException | NullPointerException e) {
        System.out.println("Error: while reading from client socket");
        e.printStackTrace();
    }
}
```

DÉTAIL DE L'ENVOI

```java
private void sendHTTPLines(Socket clientSocket) {
    try {
        PrintWriter output = new PrintWriter(clientSocket.getOutputStream(), true);
        output.println("HTTP/1.0 300 OK");
        output.println("Content-Type: text/html; charset=iso-8859-1");
        output.println("");
        output.println("<HTML><HEAD><TITLE>Example</TITLE></HEAD>");
        output.println("<BODY><P>&Ccedil;a marche !</P></BODY></HTML>");
    }
    catch (IOException e) {
        System.out.println("Error: while sending to client socket");
        e.printStackTrace();
    }
}
```

EXEMPLE D'EXÉCUTION (1/2)

```
Recv: GET / HTTP/1.1
Recv: Host: localhost:8000
Recv: Connection: keep-alive
Recv: Cache-Control: max-age=0
Recv: sec-ch-ua: "Not.A/Brand";v="8", "Chromium";v="114", "Google Chrome";v="114"
Recv: sec-ch-ua-mobile: ?0
Recv: sec-ch-ua-platform: "Windows"
Recv: Upgrade-Insecure-Requests: 1
Recv: User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36
```

EXEMPLE D'EXÉCUTION (2/2)

```
Recv: Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng
,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Recv: Sec-Fetch-Site: none
Recv: Sec-Fetch-Mode: navigate
Recv: Sec-Fetch-User: ?1
Recv: Sec-Fetch-Dest: document
Recv: Accept-Encoding: gzip, deflate, br
Recv: Accept-Language: fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7
Recv:
```

1.2 – PROGRAMMATION UDP SIMPLE (MONO-THREAD)

REMARQUE PRÉALABLE

- Attention !
  - Les fonctions « réseaux » en UDP sont souvent plus facile à utiliser…
  - … mais la programmation UDP est significativement plus difficile (car il faut gérer soi-même la non-fiabilité, comme la perte des messages)

1.2.1 – DÉTAIL DU CLIENT UDP

STRUCTURE DU CLIENT

```
import java.net.*;
import java.io.*;

public class SimpleUDPClient {

    private static final int BUFFER_LENGTH = 10;
    private DatagramSocket socket;

    public SimpleUDPClient(String name, int port) {
        // …
    }

    private void sendPing(String name, int port) throws IOException {
        // …
    }

    private void receivePong() throws IOException {
        // …
    }

}
```

DÉTAIL DU CONSTRUCTEUR

```
public SimpleUDPClient(String name, int port) {
    try {
        socket = new DatagramSocket();
        sendPing(name, port);
        receivePong();
        socket.close();
    }
    catch (IOException e) {
        System.out.println("Error: I/O exception");
        e.printStackTrace();
    }
}
```

DÉTAIL DE L'ENVOI ET DE LA RÉCEPTION

```
private void sendPing(String name, int port) throws IOException {
    byte buffer[] = "ping".getBytes();
    socket.send(
        new DatagramPacket(buffer, buffer.length, InetAddress.getByName(name), port));
    System.out.println("Client Send: <ping>");
}

private void receivePong() throws IOException {
    DatagramPacket buffer = new DatagramPacket(new byte[BUFFER_LENGTH], BUFFER_LENGTH);
    socket.receive(buffer);
    String s = new String(buffer.getData(), 0, buffer.getLength());
    System.out.println(
        "Client Recv: <"+s+"> from "+buffer.getAddress()+":"+buffer.getPort());
}
```

1.2.2 – DÉTAIL DU SERVEUR UDP

STRUCTURE DU SERVEUR

```
import java.net.*;
import java.io.*;

public class SimpleUDPServer {

    private static final int BUFFER_LENGTH = 10;
    private DatagramSocket socket;

    public SimpleUDPServer(int port) {
        // …
    }

    private InetSocketAddress receivePing() throws IOException {
        // …
    }

    private void sendPong(InetSocketAddress destination) throws IOException {
        // …
    }

}
```

DÉTAIL DU CONSTRUCTEUR

```java
public SimpleUDPServer(int port) {
    try {
        socket = new DatagramSocket(port);
        InetSocketAddress destination = receivePing();
        sendPong(destination);
        socket.close();
    }
    catch (IOException e) {
        System.out.println("Error: I/O exception");
        e.printStackTrace();
    }
}
```

DÉTAIL DE L'ENVOI ET DE LA RÉCEPTION

```
private InetSocketAddress receivePing() throws IOException {
    DatagramPacket buffer = new DatagramPacket(new byte[BUFFER_LENGTH], BUFFER_LENGTH);
    socket.receive(buffer);
    String s = new String(buffer.getData(), 0, buffer.getLength());
    System.out.println("Server Recv: <"+s+"> from "+buffer.getAddress()+":"+buffer.getPort());
    return new InetSocketAddress(buffer.getAddress(), buffer.getPort());
}

private void sendPong(InetSocketAddress destination) throws IOException {
    byte buffer[] = "pong".getBytes();
    socket.send(new DatagramPacket(buffer, buffer.length, destination));
    System.out.println("Server Send: <pong> to "+destination.getAddress()+":"+destination.getPort());
}
```

1.3 – MÉTHODES SUPPLÉMENTAIRES

1.3.1 – CLASSES UTILITAIRES

CLASSE INETADDR

- Classe InetAddr
  - Permet de représenter des adresses IPv4 ou IPv6
- Méthodes utiles :
  - public static InetAddress getByName(String host);
  - public static InetAddress getLocalHost();
  - public String getCanonicalHostName();
  - public String getHostName();
  - public String getHostAddress();

CLASSES SOCKETADDRESS ET INETSOCKETADDRESS

- Classe SocketAddress : classe abstraite utilisée pour les sockets (de manière générale)
- Classe InetSocketAddress : classe fille de SocketAddress, utilisée pour les sockets Internet
- Méthodes utiles :
  - public InetSocketAddress(InetAddress addr, int port);

CLASSES SOCKET ET SERVERSOCKET

- Classe Socket : permet d'implémenter des sockets (TCP, côté client)
- Méthodes utiles :
  - public void shutdownInput(); et public void shutdownOutput();
  - public void setReuseAddress(boolean b);
  - public void setTcpNoDelay(boolean b); // active/désactive l'algorithme de Naggle
  - public void setSoLinger(boolean b, int linger); // délai introduit à la fermeture d'une connexion TCP pour permettre aux données en instance d'être transmises
  - public void setKeepAlive(boolean b); // active/désactive l'envoi de messages KeepAlive pour les connexions inactives longtemps
- Classe ServerSocket : permet d'implémenter des sockets (TCP, côté serveur)
- Méthodes utiles : similaires à Socket

1.3.2 – SOCKETS NON BLOQUANTES

DÉFINITION ET USAGE

- Par défaut, les sockets sont bloquantes
  - A la réception, l'appel à « recv » bloque jusqu'à ce que des données deviennent disponibles
  - A l'envoi, l'appel à « send » bloque jusqu'à ce que les données aient quitté la couche applicative
- On peut utiliser des sockets bloquantes ou non bloquantes
- Pourquoi utiliser des sockets non bloquantes ?
  - Réception possible de commandes d'une IHM en parallèle de données réseaux
  - Timeout sur la réception des messages
  - Serveur multi-clients

ARCHITECTURE

```java
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.Iterator;

public class NonBlockingTCPServer {

 private static final int TIMEOUT = 10000;
 private static final int BUFFER_SIZE = 10000;

 private ServerSocketChannel serverChannel;
 private Selector selector;
 private boolean end;

 public NonBlockingTCPServer(int port) {
    // …
 }
```

```java
     public void run() {
          // …
     }

     private void processEvents() {
          // …
     }

     private void acceptingClient(SelectionKey key) {
          // …
     }

     private void sendingToClient(SelectionKey key) {
          // …
     }

     private void receivingFromClient(SelectionKey key) {
          // …
     }

}
```

UTILISATION (1/4)

```java
public NonBlockingTCPServer(int port) {
	try {
		serverChannel = ServerSocketChannel.open();
		serverChannel.configureBlocking(false);
		serverChannel.bind(
			new InetSocketAddress("127.0.0.1", port));
		selector = Selector.open();
		serverChannel.register(selector, SelectionKey.OP_ACCEPT);
	}
	catch (IOException e) {
		e.printStackTrace();
	}
}
```

```java
public void run() {
	end = false;
	System.out.println("Server - start");
	while (!end) {
		try {
			selector.select(TIMEOUT);
			processEvents();
		}
		catch (IOException e) {
		}
	}
	System.out.println("Server - stop");
	try {
		selector.close();
		serverChannel.socket().close();
		serverChannel.close();
	}
	catch (IOException e) {
	}
}
```

UTILISATION (2/4)

```java
private void processEvents() {
	Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
	while (keys.hasNext()) {
		SelectionKey key = keys.next();
		keys.remove();
		if (!key.isValid()) {
			System.out.println("client disconnected");
			end = true;
		}
		else {
			if (key.isAcceptable()) {
				acceptingClient(key);
			}
			if (key.isWritable()) {
				sendingToClient(key);
			}
			if (key.isReadable()) {
				receivingFromClient(key);
			}
		}
	}
}
```

```java
private void acceptingClient(SelectionKey key) {
	try {
		SocketChannel socketChannel = serverChannel.accept();
		System.out.println("  accepting client "+
			socketChannel.getRemoteAddress());
		socketChannel.configureBlocking(false);
		socketChannel.register(selector, SelectionKey.OP_READ);
	}
	catch (IOException e) {
	}
}
```

UTILISATION (3/4)

```
private void sendingToClient(SelectionKey key) {
     System.out.println("  sending to client");
     SocketChannel socketChannel =
(SocketChannel)key.channel();
     try {
          socketChannel.write(ByteBuffer.wrap(
               "HTTP/1.0 200 OK\n”+
               “Content-Type:text/html\n”+
               “\n”+
               “<html><body>OK!</body></html>\n“
               .getBytes()));
          // deactivates further "write" notifications
          key.interestOps(SelectionKey.OP_READ);
     }
     catch (IOException e) {
     }
}
```

UTILISATION (4/4)

```java
private void receivingFromClient(SelectionKey key) {
	SocketChannel socketChannel = (SocketChannel)key.channel();
	ByteBuffer readBuffer = ByteBuffer.allocate(BUFFER_SIZE);
	readBuffer.clear();
	int length;
	try {
		length = socketChannel.read(readBuffer);
	}
	catch (IOException e) {
		length = -1;
	}
	if (length==-1) {
		try {
			socketChannel.close();
		}
		catch (IOException e) {
		}
		key.cancel();
	}
```

```java
	else {
		readBuffer.flip();
		byte [] data = new byte[BUFFER_SIZE];;
		readBuffer.get(data, 0, length);
		System.out.println("  receiving from client <"+
			new String(data)+">");
		try {
			socketChannel.register(selector,
				SelectionKey.OP_READ|
				SelectionKey.OP_WRITE);
		}
		catch (IOException e) {
			end = true;
		}
	}
}
```

1.4 – PROGRAMMATION MULTI-THREADS

QUAND FAIRE DU MULTI-THREAD ?

- Un serveur doit être multi-threads (ou multi-processus) s'il doit gérer plusieurs clients en parallèle
  - C'est est quasiment toujours le cas !
- Un client (ou un serveur) doit être multi-threads (ou multi-processus) s'il a une interface graphique, ou si les envois/réceptions peuvent s'entrelacer arbitrairement (ex : IRC)
  - C'est souvent le cas

MULTI-THREADS OU MULTI-PROCESSUS ?

- Un processus est un programme en cours d'exécution
  - Deux processus différents s'exécutent (presque) en parallèle
  - Deux processus différents peuvent communiquer
  - Deux processus différents s'exécutent dans des espaces mémoires séparés
- Un thread est un fil d'exécution d'un processus
  - Deux threads différents s'exécutent (presque) en parallèle
  - Deux threads différents peuvent communiquer
  - Deux threads différents s'exécutent dans un même espace mémoire
  - Un thread est souvent appelé « processus léger »
- Java permet de faire de la programmation multi-threads assez simplement

1.4.1 – COMMENT EXÉCUTER UNE TÂCHE DANS UN THREAD EN JAVA ?

MÉTHODE 1 : ÉTENDRE LA CLASSE THREAD

- Étapes à suivre :
  - La classe qui réalise la tâche doit étendre la classe Thread
  - La tâche qui doit être exécutée dans un thread doit être appelée dans la fonction run()
  - On crée un thread avec le constructeur "Thread(Runnable r, String name)"
  - On lance la méthode "start" de la classe, ce qui démarre le thread

MÉTHODE 1 : EXEMPLE (1/2)

```
import java.util.Random;
public class Multi2 extends Thread {

    private String name;

    public Multi2(String name) {
        this.name = name;
    }
    public void run() {
        Random random = new Random();
        for (int i=0; i<5; i++) {
            System.out.println("Thread<"+name+">="+(i+1));
            try {
                Thread.sleep(random.nextInt(100));
            }
            catch (InterruptedException e) {
            }
        }
    }
}
```

MÉTHODE 1 : EXEMPLE (2/2)

```
private static void testThreads1() {
    Thread t1 = new Thread(new Multi1("one"), "one");
    Thread t2 = new Thread(new Multi1("two"), "two");
    t1.start();
    t2.start();
}
```

```
Thread<one>=1
Thread<two>=1
Thread<one>=2
Thread<two>=2
Thread<one>=3
Thread<one>=4
Thread<two>=3
Thread<two>=4
Thread<one>=5
Thread<two>=5
```

MÉTHODE 2 : IMPLÉMENTER L'INTERFACE RUNNABLE

- Étapes à suivre :
  - La classe qui réalise la tâche doit implémenter l'interface Runnable
  - La tâche qui doit être exécutée dans un thread doit être appelée dans la fonction run()
  - On crée un thread avec le constructeur "Thread(Runnable r, String name)"
  - On lance la méthode "start" de la classe, ce qui démarre le thread
- En pratique : cette méthode est très similaire à la précédente, mais beaucoup plus utile, car on n'a pas besoin d'hériter d'une classe (pour rappel, Java ne permet d'hériter que d'une seule classe)

MÉTHODE 2 : EXEMPLE (1/2)

```
import java.util.*;
public class Multi2 implements Runnable {

    private String name;

    public Multi2(String name) {
        this.name = name;
    }
    public void run() {
        Random random = new Random();
        for (int i=0; i<5; i++) {
        System.out.println("Thread<"+name+">="+(i+1));
            try {
                Thread.sleep(random.nextInt(100));
            }
            catch (InterruptedException e) {
            }
        }
    }
}
```

MÉTHODE 2 : EXEMPLE (2/2)

```java
private static void testThreads2() {
    Thread t1 = new Thread(new Multi2("one"), "one");
    Thread t2 = new Thread(new Multi2("two"), "two");
    t1.start();
    t2.start();
}
```

```
Thread<two>=1
Thread<one>=1
Thread<two>=2
Thread<two>=3
Thread<one>=2
Thread<two>=4
Thread<one>=3
Thread<two>=5
Thread<one>=4
Thread<one>=5
```

1.4.2 – FONCTIONS UTILES SUR LES THREADS

FONCTIONS SUR LES THREADS

- Méthode « join » :
  - Le thread actuel doit appeler cette méthode sur un autre thread (en utilisant une référence sur l'objet Thread en question)
  - Le thread actuel bloque en attendant que l'autre thread termine
- Méthode « yield » :
  - Le thread actuel passe la main aux autres threads
- Méthode « sleep » :
  - Le thread actuel est bloqué pour une certaine durée
- Méthode « currentThread » :
  - Retourne le thread actuel

1.4.3 – COMMENT FAIRE COMMUNIQUER DES THREADS EN JAVA ?

OBJET MONITEUR ET MOT-CLÉ SYNCHRONIZED

- Objet moniteur
  - Un « moniteur » est un mécanisme de programmation concurrente qui permet (1) de garantir l'exclusion mutuelle de threads, et (2) d'attendre qu'une certaine condition soit vraie (suite à la notification d'autres threads)
  - En Java, les moniteurs sont acquis/libérés en début/fin des blocs « synchronized »
- Mot-clé « synchronized » :
  - Méthode non-statique ➔ le moniteur est l'instance (ex : public synchronized void add(…))
  - Méthode statique ➔ le moniteur est la classe (ex : public static synchronized void add(…))
  - Bloc synchronisé ➔ le moniteur est l'objet désigné (ex : synchronized(o) { … })

MÉTHODES UTILES

- Méthodes
  - Méthode « wait » : le thread actuel est bloqué jusqu'à ce qu'un autre thread appelle « notify »
  - Méthode « notify » : réveille un thread arbitraire qui attendait sur l'objet moniteur
  - Méthode « notifyAll » : réveille tous les threads qui attendaient sur l'objet moniteur
- Remarque : pour pouvoir appeler ces méthodes, il faut avoir un lock sur l'objet moniteur (donc, il faut être dans un bloc « synchronized » sur l'objet moniteur)

RÉENTRANCE

- Remarque :
  - La synchronisation est réentrante : quand un thread a acquis un lock sur un moniteur, il peut le réacquérir un nombre quelconque de fois

PROBLÈME DU *DEAD-LOCK*

- Un programme multi-threads peut se bloquer dans certaines conditions
  - Par exemple : un thread A possède un moniteur A et attend un moniteur B ; un thread B possède un moniteur B et attend un moniteur A
  - On parle de « *dead-lock* » (ou d'interblocage)
  - Il est généralement difficile d'identifier les sources possibles de dead-locks, de les reproduire, et donc de les résoudre

EXEMPLE D’INTERBLOCAGE : LE DÎNER DES PHILOSOPHES

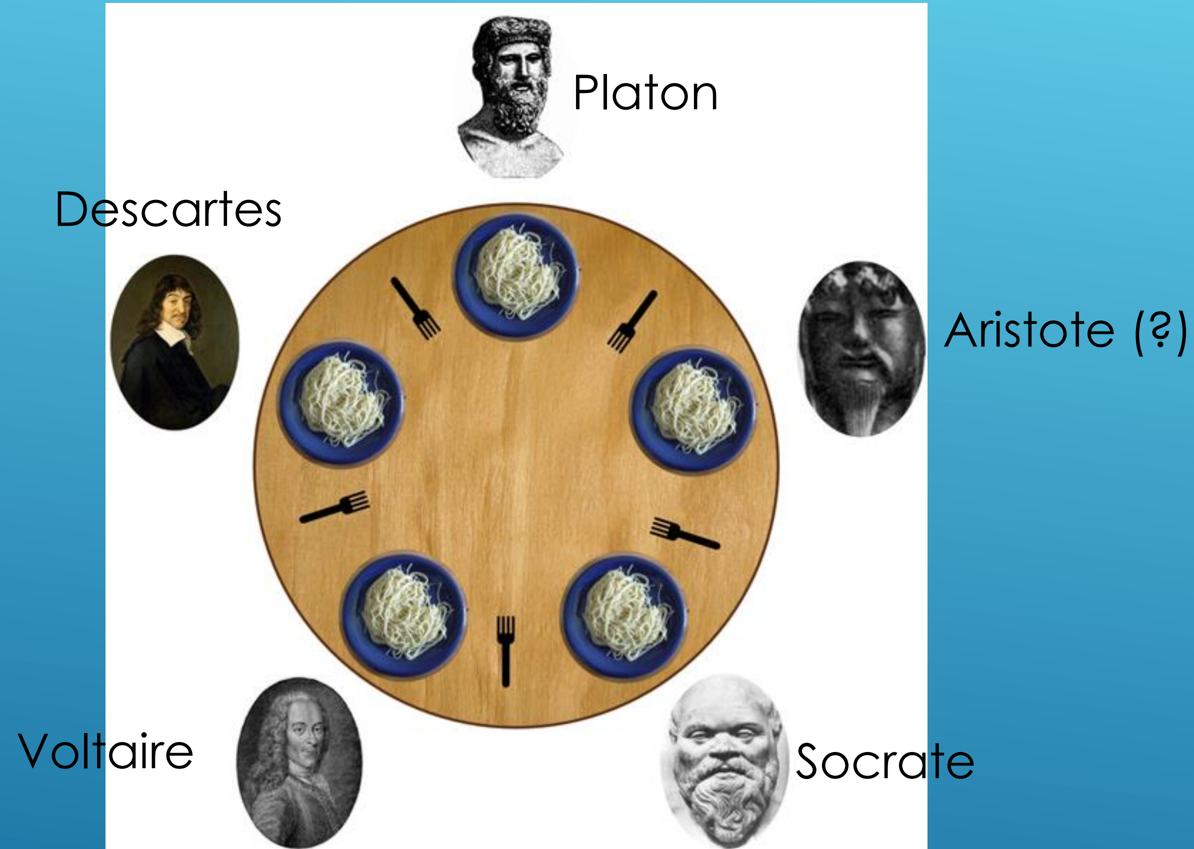

Illustration du problème du dîner des philosophes, proposé par Dijkstra (https://fr.wikipedia.org/wiki/D%C3%AEner_des_philosophes, le 27/06/2023)

1.5 – GESTION SÛRE DE MESSAGES UDP

QUELQUES REMARQUES POUR UNE GESTION SÛRE DES MESSAGES UDP

- Des messages UDP peuvent être :
  - Dupliqués → on peut les numéroter (tout en stockant une liste des derniers messages reçus) pour dé-dupliquer
  - Reçus dans le désordre → on peut les numéroter pour ignorer les messages anciens
  - Perdus → on peut acquitter et retransmettre les messages non reçus (ce qui nécessite de les stocker à l'émission)