Professeur : Hamid Mcheick	Groupe : **En équipe de 2 ou 3 étudiants max**
Trimestre : Aut. 2025	Date de distribution : 03 septembre 2025
Pondération : 15 points	Date de remise : avant 13h00 le 29 sept. 2025

Parmi les sujets donnés ici-bas, **vous devez choisir deux questions parmi les questions suivantes** :
• Les documents demandés doivent être remis (téléversés) à la page web du cours. Des pénalités de 10% par jour seront appliquées pour tout travail remis après cette date.

Question 1

Ce travail consiste à concevoir et développer une application (web ou mobile ou Cloud avec notebook Colab ou Jupyter, Firebase, …) permettant de calculer l'indice de masse corporelle IMC d'une personne. Vous pouvez utiliser des services cloud tels que le stockage (par exemple, Azure MS, Amazon S3, etc.) ou des systèmes de gestion de bases de données locales (MSSQL, MySQL, etc.) et des services d'hébergement (par exemple nginx, Glassfish, Tomcat, IIS, Apache, …), vous pouvez également implémenter et exécuter cette application localement si vous voulez (serveur web, Docker, …). L'étudiant devra choisir un langage de programmation et un framework selon les besoins.
Évaluation de la mission :

- (80 %) Implémenter les fonctionnalités de l'application et fournir un code source bien documenté. Je vous laisse le choix de sélectionner les technologies (ou combinaison) pour développer cette application :
  - L'application peut être réalisée à l'aide d'ASP.NET, JAVA, Python, HTML, CSS, XML, Node.JS, Ruby, PHP, C++, C#, React, Angular JS, Xamarin, Android Studio, Flutter ou d'autres technologies.
  - La connexion entre le Code (Java, C#, C++, PHP, etc.) et la Base de données (Oracle, MSSQL ou MySQL).
  - Le travail peut être exécuté sur plusieurs plateformes (Windows, Linux, Android, iOS, etc.) et l'application doit être accessible de l'extérieur via Internet ou Telnet ou en local (serveur web local sur la même machine). Vous devez fournir l'URL pour accéder à votre application.
- (20%) Présentation et documentation comprenant :
  - L'introduction et description du système
  - La description de l'architecture de l'application
  - Le guide de l'utilisateur
  - Le test (exécution de code)

Question 2

Développer et comparer deux versions d'un même service distribué - Java RMI puis gRPC et faire la comparaison entre ces deux technologies.

Partie 1 : Implémentation Java RMI (40 %)

a) Décrivez l'architecture RMI utilisée (registry, stub/skeleton, sérialisation, etc.).
b) Développez un service « catalogue de promotions » complet (serveur, interface distante, client de test) et fournissez un script d'exécution.

Exemple de l'interface :

```
public interface PromoService extends Remote {
   List<Promo> listPromos() throws RemoteException;
   Promo getPromo(int id) throws RemoteException;
   void  addPromo(Promo p) throws RemoteException;
   void  removePromo(int id) throws RemoteException;
}
```

c) Présentez et commentez les avantages / inconvénients de votre solution RMI.

Partie 2 : Implémentation gRPC (40 %)

a) Expliquez l'architecture gRPC (fichiers, proto, stub générés, Channel, HTTP/2, streaming, etc.).

- **IDL**: service and messages declared in promos.proto; protoc generates strongly-typed stubs.
- **Channel** is a multiplexed HTTP/2 connection; **stub** uses it to send binary **protobuf** frames.
- Four call styles: unary, server-streaming, client-streaming, bi-di.
- **Interceptors** add deadlines, auth, tracing; load balancing happens client-side.
- Because transport is HTTP/2 + TLS, it tunnels through reverse proxies and Kubernetes services naturally.

b) Réalisez la même application « catalogue de promotions » avec gRPC : fichier, proto, serveur, client, script d'exécution (langage de votre choix).

```
syntax = "proto3";
service PromoService {
  rpc ListPromos(Empty) returns (PromoList);
  rpc GetPromo(Id)      returns (Promo);
  rpc AddPromo(Promo)   returns (Empty);
```

```
  rpc RemovePromo(Id)   returns (Empty);

}
```

c) Dressez la liste des avantages / inconvénients de gRPC observés dans votre implémentation.

Partie 3 : Analyse comparative RMI vs gRPC (20 %)

a) Comparez les résultats et discutez : protocoles, sérialisation, portabilité, évolutivité d'API, facilité de développement, etc.

- **Performance.** gRPC's binary Protocol Buffers over HTTP/2 routinely outpace Java serialization on latency and bandwidth.
- **Language scope.** RMI is confined to the JVM; gRPC ships first-class stubs for most major languages.
- **Network fit.** HTTP/2 on port 443 traverses proxies with ease, whereas JRMP is often filtered.
- **Developer effort.** RMI is setup-light for Java-only projects; gRPC's code-generation repays its extra step with type-safe, cross-language clients.
- **API stability.** Protobuf's additive field model keeps gRPC interfaces forward-compatible; RMI breaks when parameter classes evolve.
- **Streaming.** gRPC offers native bidirectional streams; RMI provides only synchronous calls, forcing bespoke observer logic for real-time flows.

b) Concluez sur les contextes où l'une ou l'autre technologie serait préférable.

gRPC for modern, internet-facing micro-services.

RMI only for legacy or intra-JVM clusters where its simplicity outweighs its limitations.

Question 3

Le but de cette question est de familiariser l'étudiant

1. au développement d'applications réparties en utilisant la communication par message (Socket), par procédure, par objet (RMI), etc.
2. aux concepts de passage de paramètres, de sérialisation, de réflexion, de nommage, de persistance, de collaboration dans les systèmes distribués, etc.

L'objective de ce travail pratique est l'implémentation d'une plateforme de calcule collaborative et distribuée. En effet, l'idée est de permettre à un client, avec ressources limitées (CPU, mémoire, etc.), l'exécution distribuée de certaines tâches quotidiennes. Pour ce faire, le client délègue l'accomplissement d'une ou plusieurs tâches à un serveur (ou machine) distant selon le diagramme ci-dessous :

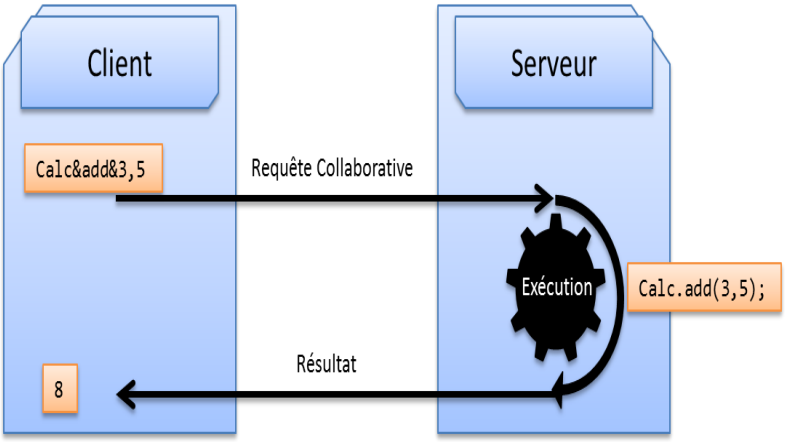

FIGURE 1 : PROCESSUS DU CALCULE DISTRIBUE

Le diagramme (Figure 1) illustre l'exécution d'une simple requête de calcule d'addition de deux nombres entiers. Ce processus évoque l'utilisation des Sockets ou autres techniques afin d'envoyer les requêtes (codes avec les données) à exécuter au serveur, ces requêtes comportent:

- Nom de la Classe et de la Méthode à invoquer, ainsi que les arguments de cette dernière.
- Fichier .java ou .class ou l'objet sérialisé à charger ou désérialiser lors de l'exécution.

Le client peut envoyer les requêtes au serveur en trois façons selon le protocole collaboratif suivant :

1) **SOURCEColl** : Code source de(s) la classe(s) demandé(es) ou
2) **BYTEColl** : Bytecode Code compilé de ces classes ou
3) **OBJECTColl** : Objets de ces classes

Vous devez implémenter seulement les cas (2 et 3).

Selon le cas choisi, ton serveur doit être capable de traiter la requête du client. Par exemple, si le client envoie des byte codes, alors le serveur doit être capable de

i) charger (loader) ces codes
ii) créer des instances (objets)
iii) exécuter la (les) méthode(s) demandé(es)

Le serveur extrait les informations nécessaires (par exemple, classe en code octet-bytecode, nom de la classe, méthode, ses paramètres, ...), exécute le service demandé (méthode dans ce cas), et retourne le résultat au client, qui montre l'information dans son interface. Notons que vous avez besoin de compiler le code source, charger la classe, créer des instances au besoin, exécuter la méthode (service) demandée dans cette classe, et retourner le résultat. J'ai ajouté deux classes (Calc.java et ByteStream.java) qui peuvent vous servir dans ce TP (voir pages suivantes).

Mots clés : Socket, RMI, Thread, InputStream, FileInputStream, Réflexion, OutputStream.
Livrables (en place): utilisez clé USB pour remettre ce TP1, qui sera corrigé en classe.

1. Code complet en Java avec commentaires 80%
2. Guide d'utilisation et trace d'exécution de l'application 20%

Remarque : Exécution et démonstration en classe par l'étudiant.

```
      /** ---------------------------------- Calc.java ----------------------------------------------*/

package edu.uqac.gri.netreflect;

public class Calc {

      public int add(String a, String b){
            int x = Integer.parseInt(a);
            int y = Integer.parseInt(b);
            return x + y;
      }
}
      /** ---------------------------------- ByteStream.java ----------------------------------------------*/

package edu.uqac.gri.netreflect;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;

public class ByteStream {
       private static byte[] toByteArray(int in_int) {
               byte a[] = new byte[4];
               for (int i=0; i < 4; i++) {

                 int  b_int = (in_int >> (i*8) ) & 255;
                 byte b = (byte) ( b_int );

                 a[i] = b;
                }
               return a;
             }

             private static int toInt(byte[] byte_array_4) {
               int ret = 0;
               for (int i=0; i<4; i++) {
                 int b = (int) byte_array_4[i];
                 if (i<3 && b<0) {
                   b=256+b;
                 }
                 ret += b << (i*8);
               }
               return ret;
             }

             public static int toInt(InputStream in) throws
java.io.IOException {
```

```
        byte[] byte_array_4 = new byte[4];

        byte_array_4[0] = (byte) in.read();
        byte_array_4[1] = (byte) in.read();
        byte_array_4[2] = (byte) in.read();
        byte_array_4[3] = (byte) in.read();

        return toInt(byte_array_4);
      }

      public static String toString(InputStream ins) throws
java.io.IOException {
        int len = toInt(ins);
        return toString(ins, len);
      }

      private static String toString(InputStream ins, int len)
throws java.io.IOException {
        String ret=new String();
        for (int i=0; i<len;i++) {
          ret+=(char) ins.read();
        }
        return ret;
      }

      public static void toStream(OutputStream os, int i) throws
java.io.IOException {
        byte [] byte_array_4 = toByteArray(i);
        os.write(byte_array_4);
      }

      public static void toStream(OutputStream os, String s) throws
java.io.IOException {
        int len_s = s.length();
        toStream(os, len_s);
        for (int i=0;i<len_s;i++) {
          os.write((byte) s.charAt(i));
        }
        os.flush();
      }

      private static void toFile(InputStream ins, FileOutputStream
fos, int len, int buf_size) throws
            java.io.FileNotFoundException,
            java.io.IOException {

        byte[] buffer = new byte[buf_size];

        int       len_read=0;
```

```
    int total_len_read=0;

    while ( total_len_read + buf_size <= len) {
      len_read = ins.read(buffer);
      total_len_read += len_read;
      fos.write(buffer, 0, len_read);
    }

    if (total_len_read < len) {
      toFile(ins, fos, len-total_len_read, buf_size/2);
    }
  }

  private static void toFile(InputStream ins, File file, int
len) throws
        java.io.FileNotFoundException,
        java.io.IOException {

    FileOutputStream fos=new FileOutputStream(file);

    toFile(ins, fos, len, 1024);
  }

  public static void toFile(InputStream ins, File file) throws
        java.io.FileNotFoundException,
        java.io.IOException {

    int len = toInt(ins);
    toFile(ins, file, len);
  }

  public static void toStream(OutputStream os, File file)
      throws java.io.FileNotFoundException,
             java.io.IOException{

    toStream(os, (int) file.length());

    byte b[]=new byte[1024];
    InputStream is = new FileInputStream(file);
    int numRead=0;

    while ( ( numRead=is.read(b)) > 0) {
      os.write(b, 0, numRead);
    }
    os.flush();
  }
}
```

Question 4

Le protocole HTTP (HyperText Transport Protocol) est le protocole Web utilisé pour la communication de documents pour le World Wide Web. Il permet la transmission de documents HTML, d'images ou autres entre un serveur web et un navigateur. HTTP est défini dans la RFC 2616. Vous allez implémenter une version de ce protocole HTTP avec la communication par message (Socket). Vous êtes invités à étudier les règles de ce protocole HTTP en consultant la RFC 2616.

Format des requêtes :

Une requête HTTP minimale est un message textuel de lignes ayant la forme suivante :

```
GET Test/texte.html HTTP/1.0
Accept: text/plain , image/*
If-Modified-Since: Wed, 10 Sep 2018 14:20:21 GMT
Referer: http://www.uqac.ca/
User-Agent: Mozilla/2.0
Ligne blanche
```

Cette requête demande le fichier racine / du serveur contacté, en utilisant la variante 1.0 ou 1.1 de la norme HTTP. Attention, cette norme spécifie que les retours à la ligne qui doivent être marqués par des "\r\n" (encore que de simples "\n" semblent être tolérés par certains serveurs).

Plus généralement :

- La méthode GET de l'exemple peut être remplacée par d'autres méthodes. La principale autre méthode est POST, qui ne diffère de GET que par la façon de transmettre les arguments des formulaires, dont nous ne nous occuperons pas ici.
- A la place du premier /, peut se trouver le chemin menant au fichier voulu sur le serveur, du genre /c/est/ici/truc.html, voire même l'URL complète du fichier voulu, comme http://www.foo.fr/c/est/ici/truc.html.
- Entre la ligne initiale et la ligne vide, peuvent se trouver des lignes facultatives précisant par exemple le navigateur utilisé (User-Agent: XXX).

Format des réponses:

La réponse du serveur est constituée d'un entête et d'un corps, séparés par une ligne vide :

- L'entête commence par une ligne de statut, du genre HTTP/1.1 200 OK.
- Après cette ligne de statut se trouver un certain nombre de lignes indiquant des informations telles que l'heure, le type et la version du serveur, le type du fichier retourné (Content-Type : ...), sa taille (Content-Length : ...), etc.
- En cas de succès de la requête, après la première ligne vide, on trouve le corps du message de réponse, constitué du fichier demandé.

```
HTTP/1.0 200 OK
Date: Wed, 12 Sep 2018 14:20:21 GMT
Server: NCSA/1.5.2
Mime-Vesion: 1.0
Content-Type: text/html
Last-Modified: Wed, 12 Sep 2018 14:20:23 GMT
Content-Length: 139
    Ligne blanche
```

<html> ……</html>

Exercice 1 _ Client HTTP simple

Écrivez un programme socket Java ou Python qui se connecte en TCP à un serveur HTTP, qui demande une page HTML et qui affiche sur sa sortie standard le texte HTML reçu.

Simuler HTTP (client) : programme client qui simule HTTP (client). Ce client peut demander d'un serveur Web existant (google.ca, amazon, ...) de récupérer un fichier index.html. Ce fichier index.html va être sauvegardé ou afficher par le client. Si ton client demande un fichier x.html qui n'existe pas, alors ces serveurs web répondre à un code xyz (comme 401) avec un message texte pour dire que ce fichier n'existe pas ou autre message explicatif. Vous pouvez tester la réponse si le code est 401, alors vous pouvez afficher ce message.

Exemple : dans cet exercice 1, le client doit récupérer une page web comme (google.ca, …) et doit l'afficher ou le sauvegarder.

Exercice 2 _ Serveur HTTP Simple

Écrivez un programme socket Java ou Python attendant une connexion sur un port donné (80), capable de recevoir des requêtes HTTP, et de toujours renvoyer une réponse (http) HTML minimale pour toutes les requêtes possibles. Pour simplifier l'implémentation de cette partie, considérons seulement les requêtes qui demandent des pages web html (comme index.html). Ton code doit jouer le rôle d'un serveur Web pour envoyer ces fichiers (index.html) aux clients (navigateurs). Testez votre serveur en utilisant un navigateur web donné.

Programme serveur qui simule HTTP (server). Ce serveur va envoyer le fichier index.html s'il existe. Si non, ce serveur va envoyer un code "401 fichier n'existe pas ou autre message" au navigateur.

Exemple: dans cet exercice 2, le serveur traite les requêtes d'un navigateur utilisé par un utilisateur. Vous pouvez créer de fichiers de côté serveur, puis votre navigateur peut de récupérer ces fichiers.

Vous pouvez développer une seule application qui parle http (Client et serveur). Je vous laisse choisir entre développer deux parties indépendantes ou une seule application HTTP Client et HTTP Serveur.

Livrables: Vous devez remettre une copie complète de deux parties du TP1 contenant :

1) le code complet de deux parties qui fonctionnent sur Eclipse/IntelliJ… ou Python 80%

2) un scenario (trace) d'exécution de chaque partie et 10%

3) un manuel d'utilisation pour chaque partie. 10%

Vous devez montrer l'exécution de ces parties pour évaluer ce TP.

Question 5

Considérons les modèles de services en Cloud Computing suivants :

IaaS, PaaS, SaaS et FaaS

Tâches à réaliser :

a) Expliquer brièvement ces modèles (25 %)
b) Développer un exemple de code complet et exécutable (application exécutable) de chaque modèle de services (50 %)
c) Identifier les avantages et les désavantages de ces modèles (25 %)

Remarques :

Pénalité de retard par jour : -10%, la note devient 0 après 7 jours de retard (une semaine).