

---

*Objectif* : Programmation UDP en C#

*Notions* : Buffer, Sérialisation, Thread, Ports, Annuaire

---

## Mémo

- Encoder/Décoder un int

```
byte[] BitConverter.GetBytes(int val)
int BitConverter.ToInt32(byte[] buffer, int offset)
```
- Encoder/Décoder un String

```
Encoding.ASCII.GetBytes(String str, int offset, int length, byte[] buffer, int
bufferOffset);
byte[] Encoding.ASCII.GetString(byte[] buffer, int offset, int length)
```
- Créer une Thread

```
Thread thread = new System.Threading.Thread(UneMethode);
thread.Start();
```
- Créer une Thread

```
Thread thread = new System.Threading.Thread(UneMethode);
thread.Start();
```
- Commandes git

```
git pull récupère les modifications des autres
git commit pour valider une modification
git tag -a vX.Y tague l'aboutissement de la version vX.Y
git push origin -tags pousse les modifications et les tags
```

## 1 Buffers, Sérialisation

1. Proposez un buffer permettant de transporter l'objet `ChatMessage`.
2. Ecrire la méthode de sérialisation `public byte[] GetBytes()`.
3. Ecrire la méthode de désérialisation `public ChatMessage(byte[] buffer)`.
4. Ajouter le pseudo dans le buffer.
5. Que se passe-t-il si les champs de taille variable sont plus grands que la place disponible dans le buffer ?
6. Comment optimiser la taille du buffer échangé ?

## 2 Protocole applicatif

Proposez un protocole applicatif permettant de supporter les commandes :

1. `POST` : demande de stockage d'un message sur le serveur.
2. `GET` : demande de recevoir tous les messages stockés sur le serveur.
3. `SUBSCRIBE` : demande de recevoir les messages postés au fur et à mesure.
4. `UNSUBSCRIBE` : demande de ne plus recevoir les messages postés au fur et à mesure.

## 3 Thread et Port

1. Côté client, rendre la lecture non bloquante et continue, afin de rendre inutile la commande `GET` et non définitive la commande `SUBSCRIBE`.
2. Que se passe-t-il si on utilise maintenant la commande `GET` ?
3. Le nouveau code est dit asynchrone. Quels sont ses avantages et inconvénients par rapport au mode synchrone ?

## 4 (Optionnel) Architecture et passage à l'échelle

1. Ajouter une commande `CREATEROOM` dont le champs `data` indique le nom de la chatroom.
2. Que se passe-t-il si on crée une room depuis une room, puis une room depuis le serveur initial ?
3. Ajouter une commande `LISTROOMS` renvoyant l'ensemble des rooms créées sur le serveur.
4. Comment trouver tous les serveurs et rooms lancées ?
5. Ajouter la commande `SIGNALER` qui permet à un serveur ou un room de se signaler à un annuaire, et la commande `LISTSERVEURS` qui renvoie la liste des serveurs enregistrés dans l'annuaire.
6. Quelle est maintenant l'architecture logicielle de cette application ?

## 5 Couche réseau Tron

Il s'agit ici de construire la couche réseau d'un jeu temps réel.

### 5.1 Choix stratégiques

Au préalable, il s'agit de répondre aux questions suivantes :

1. Quel type de protocole de communication *bas niveau* utiliser (connecté ou déconnecté) ?
2. Quelle type d'information échanger (tout ou juste les mises à jour) ?
3. Quelle est la forme des informations à échanger ?
4. Où placer les gestions : déplacement des joueurs, calcul des collisions, gestion de la fréquence (vitesse) ?
5. Comment gérer la fin de partie : déconnecter les joueurs à leur mort ?

### 5.2 Conception

1. Pour le jeu Tron, concevez une couche réseau basique : connexion, échange des paramètres de jeu, échange des données à chaque itération, conclusion.
2. Au choix, gérez :
  - les déconnexions intempestives
  - un lobby avec nombre indéfini de joueurs, puis avec création de partie.
  - les tricheries

#### 5.2.1 Tron

```
Classe Tron
{
    // Constructeur à utiliser côté serveur
    public Tron(byte <taille>, byte <nb. de joueurs>)

    // Constructeur à utiliser côté client
    public Tron(byte <taille>, byte <nb de joueurs>,
                byte <num. joueur>, byte <frequence>)

    // Accesseurs au tableau des directions
    public byte[] getDirection()
    public void setDirections(byte[] <les directions>)

    // Déplacement des joueurs
    public void Deplacement();

    // Détection des collisions (MAJ des directions)
    public void Collision(byte[] <les directions>)

    // Tuer un joueur
    public void Kill(int <numero Joueur>)

    // Retourne vrai si la partie est finie
    public bool IsFinished()
}
```

```
Classe Client
{
    // Moteur du jeu
    private Tron myTron;

    // Constructeur : IP/Port du serveur
    public Client(String <IP serveur>, int <Port serveur>)

    // Appelée au début de la partie
    public Tron Init()

    // Appelée régulièrement à chaque tour de jeu
    public void Routine()

    // Appelée à la fin de la partie
    public void Conclusion()
}
```

## ChatMessage

```
0  using System;
1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Text;
4
5  namespace Chat
6  {
7      public enum Commande {POST, GET, QUIT, STOPSERVEUR, SUBSCRIBE, UNSUBSCRIBE};
8
9      public enum CommandeType {REQUETE, REPONSE};
10
11     class ChatMessage
12     {
13         public const int bufferSize = 1500;
14
15         public Commande commande;           // type de la commande
16         public CommandeType commandeType;    // 0 si question, 1 si réponse
17         public String data;                  // données de la commande
18
19         public ChatMessage(Commande commande, CommandeType type, String data)
20         {
21             this.commande = commande;
22             this.commandeType = type;
23             this.data = data;
24         }
25
26         public ChatMessage(byte[] buffer)
27         {
28         }
29
30         public byte[] GetBytes()
31         {
32         }
33
34         public static byte[] GetBytes(Commande commande, CommandeType type, String data)
35         {
36             ChatMessage chatCommande = new ChatMessage(commande, type, data);
37             return chatCommande.GetBytes();
38         }
39
40         public override string ToString()
41         {
42             return "["+commande + "," + commandeType + "," + dataSize + ",\\" + data + "\\"
43         }
44     }
45 }
46 }
```

## Client UDP

```
string serverIP = "127.0.0.1";
int serverPort  =          ;

Socket clientSocket = new Socket(
    AddressFamily.InterNetwork,
    SocketType.Dgram,
    ProtocolType.Udp);

clientSocket.Bind(new IPEndPoint(IPAddress.Any, 22222));

IPEndPoint serverEP = new IPEndPoint(IPAddress.Parse(serverIP), serverPort);

Console.Write("? ");
String msg = Console.ReadLine();

byte[] buffer = System.Text.Encoding.ASCII.GetBytes(msg);

int nBytes = clientSocket.SendTo(buffer, 0, buffer.Length, SocketFlags.None, serverEP);

Console.WriteLine("Nouveau message envoye vers "
    + serverEP
    + " (" + nBytes + " octets)"
    + ": \"" + msg + "\"");

Console.WriteLine("Fermeture Socket...");
clientSocket.Close();
```

## Client Tcp

```
string serverIP = "127.0.0.1";
int serverPort  =          ;

Socket clientSocket = new Socket(
    AddressFamily.InterNetwork,
    SocketType.Stream,
    ProtocolType.Tcp);

Console.WriteLine("Tentative de connexion...");

clientSocket.Bind(new IPEndPoint(IPAddress.Any, 22222));

IPEndPoint serverEP = new IPEndPoint(IPAddress.Parse(serverIP), serverPort);

clientSocket.Connect(serverEP);

Console.Write("? ");
String msg = Console.ReadLine();

byte[] buffer = System.Text.Encoding.ASCII.GetBytes(msg);

int nBytes = clientSocket.Send(buffer, 0, buffer.Length, SocketFlags.None);

Console.WriteLine("Nouveau message envoye vers "
    + clientSocket.RemoteEndPoint
    + " (" + nBytes + " octets)"
    + ": \"" + msg + "\"");

Console.WriteLine("Fermeture Socket...");
clientSocket.Close();
```

## Serveur UDP

```
Socket serverSocket = new Socket(
    AddressFamily.InterNetwork,
    SocketType.Dgram,
    ProtocolType.Udp);

serverSocket.Bind(new IPEndPoint(IPAddress.Any, 11111));

EndPoint clientEP = new IPEndPoint(IPAddress.Any, 0);

byte[] buffer = new byte[80];

int nBytes = serverSocket.ReceiveFrom(buffer, buffer.Length, SocketFlags.None, ref clientEP);

String msg = System.Text.Encoding.ASCII.GetString(buffer, 0, nBytes);

Console.WriteLine("Nouveau message de "
    + clientEP
    + " (" + nBytes + " octets)"
    + ": \"" + msg + "\"");

serverSocket.Close();
```

## Serveur TCP

```
Socket listenSocket = new Socket(
    AddressFamily.InterNetwork,
    SocketType.Stream,
    ProtocolType.Tcp);

listenSocket.Bind(new IPEndPoint(IPAddress.Any, 11111));

listenSocket.Listen(10);

Console.WriteLine("Attente d'une nouvelle connexion...");

Socket connectedSocket = listenSocket.Accept();

byte[] buffer = new byte[80];

int nBytes = connectedSocket.Receive(buffer, buffer.Length, SocketFlags.None);

String msg = System.Text.Encoding.ASCII.GetString(buffer, 0, nBytes);

Console.WriteLine("Nouveau message de "
    + connectedSocket.RemoteEndPoint
    + " (" + nBytes + " octets)"
    + ": \"" + msg + "\"");

Console.WriteLine("Fermeture Socket...");
connectedSocket.Close();
listenSocket.Close();
```



## Grille d'évaluation TP S32

Prénoms Noms :

### Critères

**F** Le code fonctionne en régime normal.

**T** Le code est tolérant aux pannes.

**C** Le code est clair et commenté.

**V** Le versioning est bien géré : les commits sont pertinents et commentés, la version est taguée est fait.

### Grille d'évaluation

Version	Etapes de développement	F	T	C	V
v1.0	Sérialisation				
v1.1	Commandes POST et GET				
v1.2	Commandes SUBSCRIBE et UNSUBSCRIBE				
v1.3	Client asynchrone				
v1.4	Serveur avec rooms				
v2.0	Tron 2 joueurs				
v2.1	Tron n joueurs				
v3.0	Tron + Chat				
v3.1	Fonctionnalités additionnelles				