Réseaux - Création d'une application temps réel en ligne

La programmation réseau est complexe et difficile à prendre en main, mais elle permet de mieux saisir son concept. Pour l'appréhender plus efficacement qu'à travers des exercices basiques, on se proposera d'élaborer une application en établissant un cahier des charges. Cette application sera notée sur la partie TP du module de Réseaux et pourra intervenir également sur l'évaluation de l'examen (à voir).

1. Les logiciels et langages utilisés pour la conception sont libres. Des ressources de cours sur Javascript et Godot Engine pourront être proposées.
2. Le travail sera réalisable par groupes de 2 ou 3 maximum.
3. Les critères d'évaluation sont multiples : complexité de l'application, lisibilité du code, documentation, communication et présentation orale.
4. La présentation orale se fera à la dernière séance. On présentera les technologies utilisées, et des parties de code intéressantes.

1 Cahier des charges initial

Pour tout le monde, il sera demandé de réaliser une interface graphique de chat en client / serveur. Le client doit pouvoir envoyer des messages au serveur, qui les retransmet à tous les utilisateurs, ainsi que recevoir des messages d'autres utilisateurs.

2 Cahier des charges Client-Serveur

Description : on souhaite implémenter une série de Nodes et scripts permettant à plusieurs joueurs de communiquer. On souhaite disposer sur le client :

- d'un menu de connexion au serveur, à partir de l'adresse IP et du port;
- d'un Panel qui affiche les messages reçus, avec une barre de texte qui permet d'écrire;
- d'un Panel qui affiche les utilisateurs présents, régulièrement mis à jour;
- d'un Panel qui affiche l'heure système du client, et l'heure système du serveur; régulièrement mis à jour également.

On réalisera ces composants de telle sorte qu'ils soient **réutilisables**, c-à-d qu'ils se suffisent à eux-mêmes pour fonctionner.

3 Création des composants hors-ligne

Avant de réaliser l'implémentation du programme réseau, il faut créer nos éléments et s'assurer qu'ils fonctionnent correctement hors ligne.

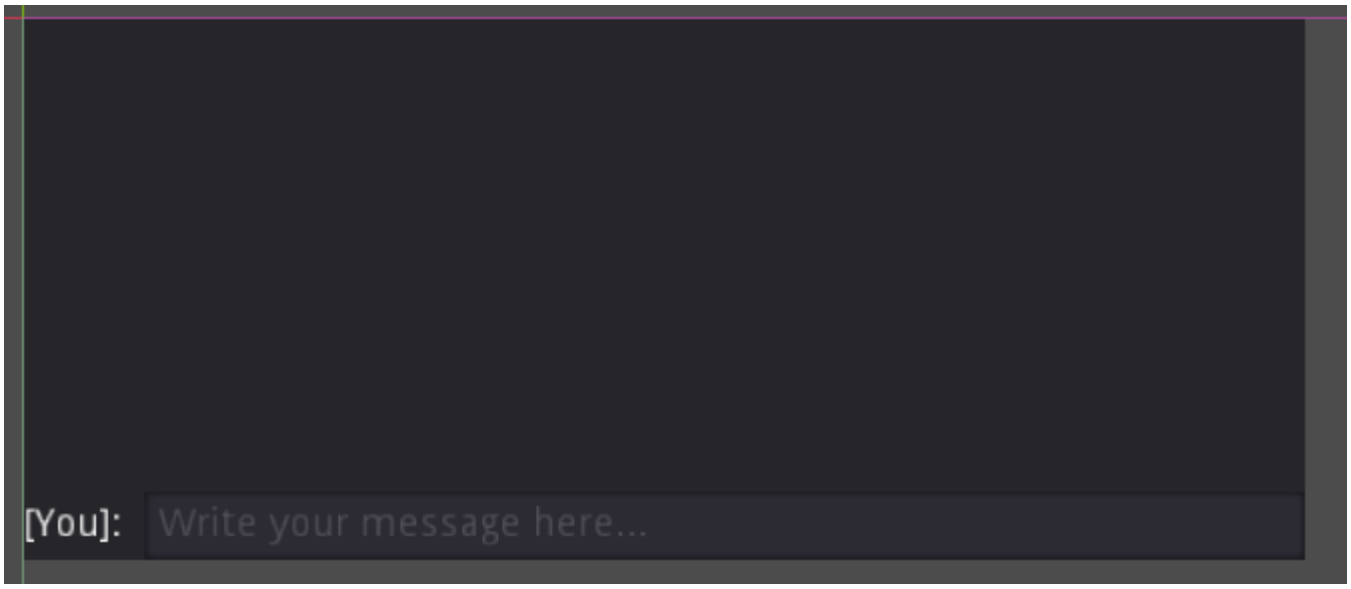

Figure 1: Exemple de résultat

3.1 Fenêtre de chat

1. Déclarer une classe `ChatMessage`, contenant les attributs nécessaires à la transmission du message : identifiant utilisateur (int), contenu du message (String), heure locale (timestamp système: float)

2. Créer un Node représentant un message sous forme écrite (sur la base d'un `Label`), qu'on appellera `LabelMessage`. Ce node devra contenir les informations du message ET écrire sa représentation textuelle (qu'on pourra générer à partir de `ChatMessage`).

3. Créer un Panel qui va contenir une liste déroulante de `LabelMessage` reçus dans l'ordre chronologique. Le script contiendra donc une méthode permettant d'insérer un `LabelMessage` au bon endroit. Le Panel contiendra également une barre d'entrée de texte, qui réagit lorsqu'on appuie sur Entrée en émettant un signal `message_sent` avec le texte écrit. Le texte écrit est effacé après avoir émis le signal.

4. Créer une Scene de test dans laquelle on place la fenêtre de chat, et lorsque le signal `message_sent` est émis, la Scene de test récupère l'heure système, crée l'instance de `Message`, puis appelle la fonction de la fenêtre de chat, qui génère un `LabelMessage` et l'insère au bon endroit. Ce faisant, on simule donc un serveur.

3.2 Liste des utilisateurs

Comme pour la fenêtre de chat, on va d'abord créer un objet `LabelUser` dérivé de `Label`, contenant l'ID et le pseudo du joueur. Ensuite, on crée un `PanelUserList`, qui contiendra un `VBoxContainer` où on y placera nos `LabelUser`. On implémentera des méthodes pour pouvoir ajouter et supprimer des `LabelUser`.

3.3 Création du client et du serveur

1. Créer un singleton `Network`. Celui-ci sera chargé de réaliser la connexion avec le serveur et contient des attributs décrivant l'état de la connexion.

   Morceaux de code gracieusement fournis ci-dessous. Il faudra implémenter les méthodes de callback.

```
# Network.gd (Client)

signal join_success
signal join_fail
signal player_added(pid, pseudo)
signal player_removed(pinfo)
```

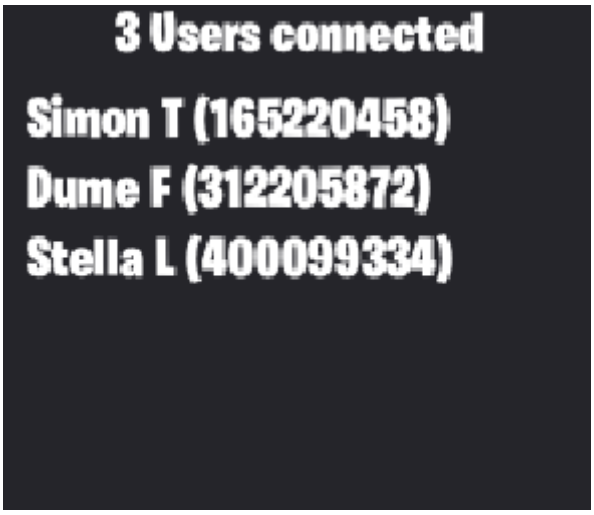

Figure 2: Exemple de résultat

```
signal disconnected(cause_value)
signal kicked(pid, reason)

func _ready():
    multiplayer.multiplayer_peer = null

## Called from the Menu scene.
func join_server(ip, port, pseudo, client_port):
    if multiplayer.multiplayer_peer != null:
        print("A connection already exists")
        return
    var multiplayer_peer = ENetMultiplayerPeer.new()
    player_info["pseudo"] = pseudo
    var error = multiplayer_peer.create_client(ip, port)
    if (error != OK) :
        print("Failed to create client")
        join_fail.emit()
        end_connection()
        return

    print("Client created")

    # When the client is created, we assign the peer
    # to the global multiplayer attribute.
    multiplayer.multiplayer_peer = multiplayer_peer

    # And then connect the signals to the peer.
    get_multiplayer().connected_to_server.connect(_on_connected_to_server)
    get_multiplayer().connection_failed.connect(_on_connection_failed)
    get_multiplayer().server_disconnected.connect(_on_disconnected_from_server)
```

```
# Network.gd (Server)

signal server_created
signal join_success
signal join_fail
signal player_list_changed
signal player_removed(pid)
signal disconnected

func _ready():
    create_server()

func create_server():
    # Initialize the networking system
    var net = ENetMultiplayerPeer.new()
```

```gdscript
    # Try to create the server
    if (net.create_server(server_info.used_port, server_info.max_players) != OK):
        print("Failed to create server")
        return
    print("Server created on port %d !"%[server_info.used_port])

    # Tell the server has been created successfully
    server_created.emit()

    # don't forget to assign the multiplayer attribute to catch signals
    multiplayer.multiplayer_peer = net
    get_multiplayer().peer_connected.connect(_on_player_connected)
    get_multiplayer().peer_disconnected.connect(_on_player_disconnected)
```

Compléter les morceaux de code pour que le client puisse se connecter au serveur en renseignant les informations dans un petit formulaire dans une scène qu'on va créer, en appelant `join`$_s$*erversurlesingleton*`Network`, e

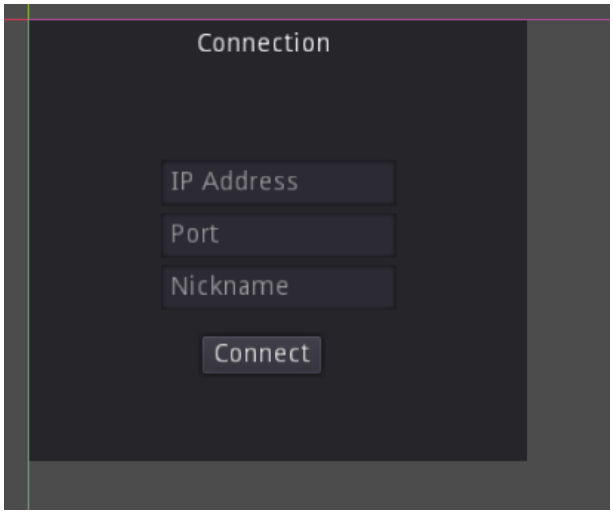

Figure 3: Exemple de formulaire à partir duquel le client se connecte. Pas de critiques sur le design sommaire.

2. Créer un singleton `NetworkChat`, qui sera chargé d'envoyer et de recevoir les messages.
3. Créer un singleton `NetworkClock`, qui sera chargé de la synchronisation de l'horloge.

4 Horloge et synchronisation

Godot dispose d'un singleton `Time` offrant des méthodes pour travailler avec le temps[1].

La norme ISO 8601 décrit la représentation numérique de la date et heure dans le but d'éviter les confusions.

Définition. L'heure UNIX désigne le nombre de secondes écoulées depuis le 1er janvier 1970.

A noter que les systèmes, dont celui de Godot, proposent une implémentation permettant de récupérer l'heure avec une précision plus fine que la seconde.

Attention : comme indiqué dans la documentation, l'heure système est sujette à des changements involontaires, mais aussi volontaires. Le singleton `Time` propose deux méthodes permettant d'effectuer des mesures précises : `get_ticks_msecs` et `get_ticks_usecs`, qui donnent le temps écoulé depuis le lancement du processus, sans qu'il soit possible de le modifier. Nous les utiliserons en parallèle avec `get_unix_time_from_system`. Attention également à vérifier les types de sortie: les méthodes `get_ticks` retournent des `int`, alors que `get_unix_time_from_system` retourne un `float` à 64 bits.

Pour effectuer une synchronisation des clients avec le serveur, on simulera le `Network Time Protocol` pour retrouver la dérivation de l'horloge client.

[1] Documentation : https://docs.godotengine.org/en/stable/classes/class_time.html

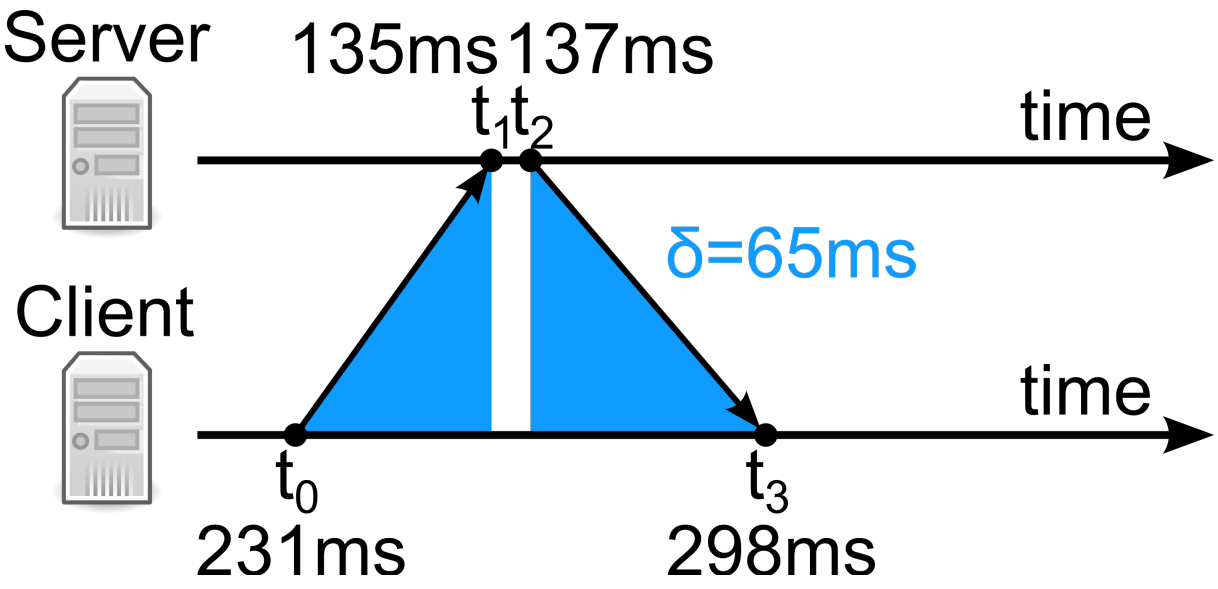

Figure 4: Schéma NTP, où le temps de traitement de la requête est compté.

Algorithme 1 : Algorithme de Cristian

Données : C est le client, S est le serveur
Résultat : Le *timestamp* auquel C doit régler son horloge pour se synchroniser à S.
1. C envoie une requête à S au temps localT_0
2. S répond en donnant le temps réel du serveur T_S de son horloge
3. C reçoit la réponse au temps local T_1. 4. C règle son horloge à $T_S + (T_1 - T_0)/2$

Cet algorithme peut aussi donner une première valeur de latence (RTT, round-time-trip) qui est ici $T_1 - T_0$.

1.

En répétant continuellement l'envoi de paquets, il est donc possible d'estimer le temps moyen de latence, en considérant que le temps de latence aller est équivalent au temps de latence retour.

Algorithme 2 : Algorithme SNTP

Données : Une liste d'entiers L qui va accueillir les valeurs de latence.
Résultat : La latence estimée du client en ms.
tant que taille(L)<10 faire
    Le client C envoie une requête au serveur S, en donnant son *timestamp* T_1
    S réceptionne et répond au client en donnant T
    C réceptionne le résultat contenant T_1
    C récupère le timestamp actuel T_2
    La latence l est $l = (T_2 - T_1)/2$ et on l'ajoute dans L
fin
nb $m \longleftarrow mediane(L)$
Retirer de L toutes les valeurs L_i supérieures à $2m$.
La latence estimée est $moyenne(L)$