TP - Programmation réseau graphique

Pour mieux cerner l'intérêt de la programmation réseau, nous allons ajouter une interface graphique à notre application de chat. Nous choisissons d'utiliser le moteur Godot [1] pour son éditeur de scènes et d'interfaces par noeuds.

1 Cahier des charges

Un client se connecte au serveur en indiquant son adresse IP, son port et un pseudo. Il affiche les messages reçus par le serveur, et peut envoyer des messages à l'aide d'un espace dédié. Ces messages sont datés et identifiés.

2 Mise en place

— Télécharger le moteur Godot et le dossier compressé de ressources sur l'ENT.

— Créer un nouveau projet TPChat_Client, ainsi qu'un nouveau projet TPChat_Server

Ressources et documentation à exploiter :

— https://docs.godotengine.org/fr/stable/tutorials/networking/high_level_multiplayer.html

— https://docs.godotengine.org/en/stable/classes/class_networkedmultiplayerpeer.htmlclass-networkedmultiplayerpeer

3 Création du client

1. En lançant le client, il faudra créer une première Scène avec un Node2D, qu'on renommera ConnectionBox. Ajouter un Panel, qui contiendra un VBoxContainer, qui contiendra trois LineEdit : une pour l'adresse IP, une pour le port, une dernière pour l'adresse. On ajoute ensuite un Label pour donner un titre à la fenêtre, et un Button pour envoyer la requête de connexion. Penser à enregistrer la Scène sous le même nom.
2. Ajouter un *placeholder* sur chaque LineEdit (fig. 1).
3. Placer les élements à peu près comme dans la figure 2. On pourra utiliser le mode Edition sélectionné par défaut, ou alors le mode Déplacement (2è icone, voir fig. 3).
4. Ajouter un script au Noeud ConnectionBox avec le même nom.
5. Connecter le signal `button_down` du bouton à ConnectionBox (clic droit sur le nom du signal) et appeler la méthode de callback comme indiqué (fig. 4)

1. https://godotengine.org/

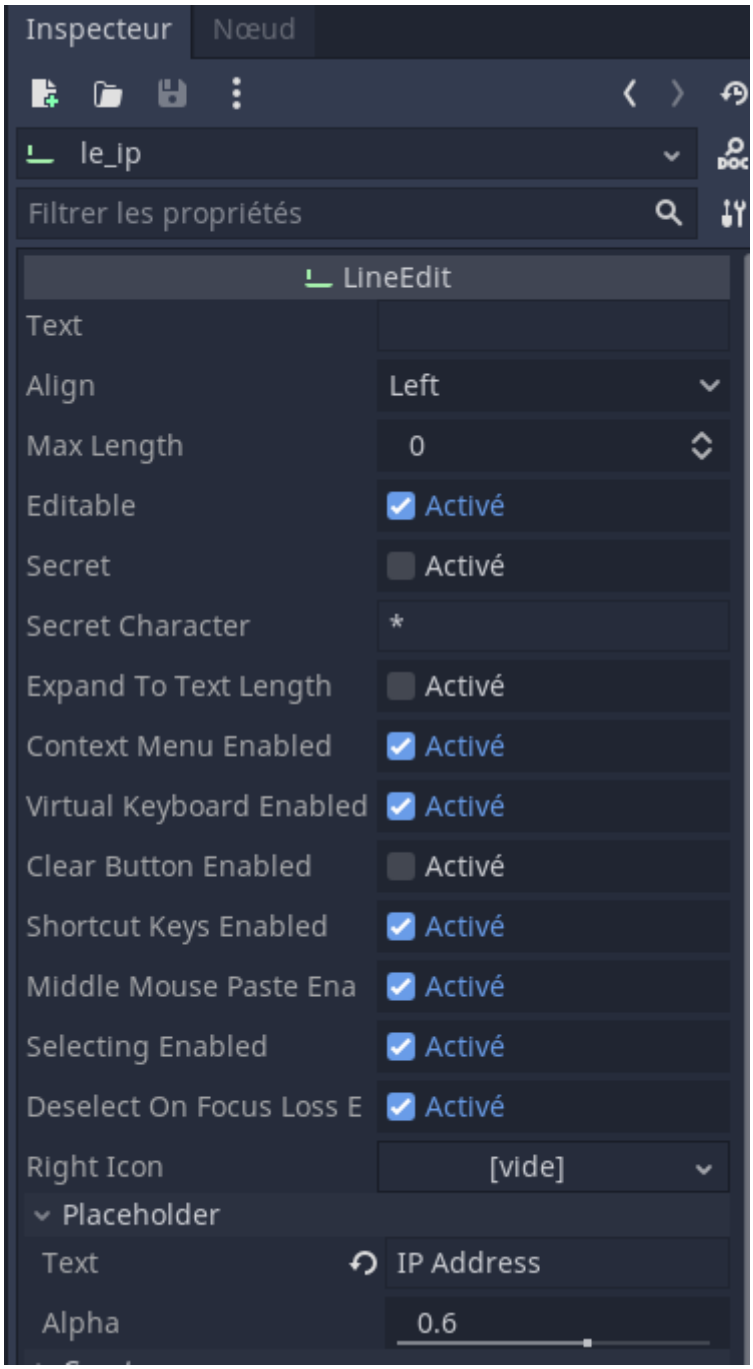

FIGURE 1 – Inspecteur d'un LineEdit. Il permet d'afficher et de modifier ses attributs directement dans l'éditeur.

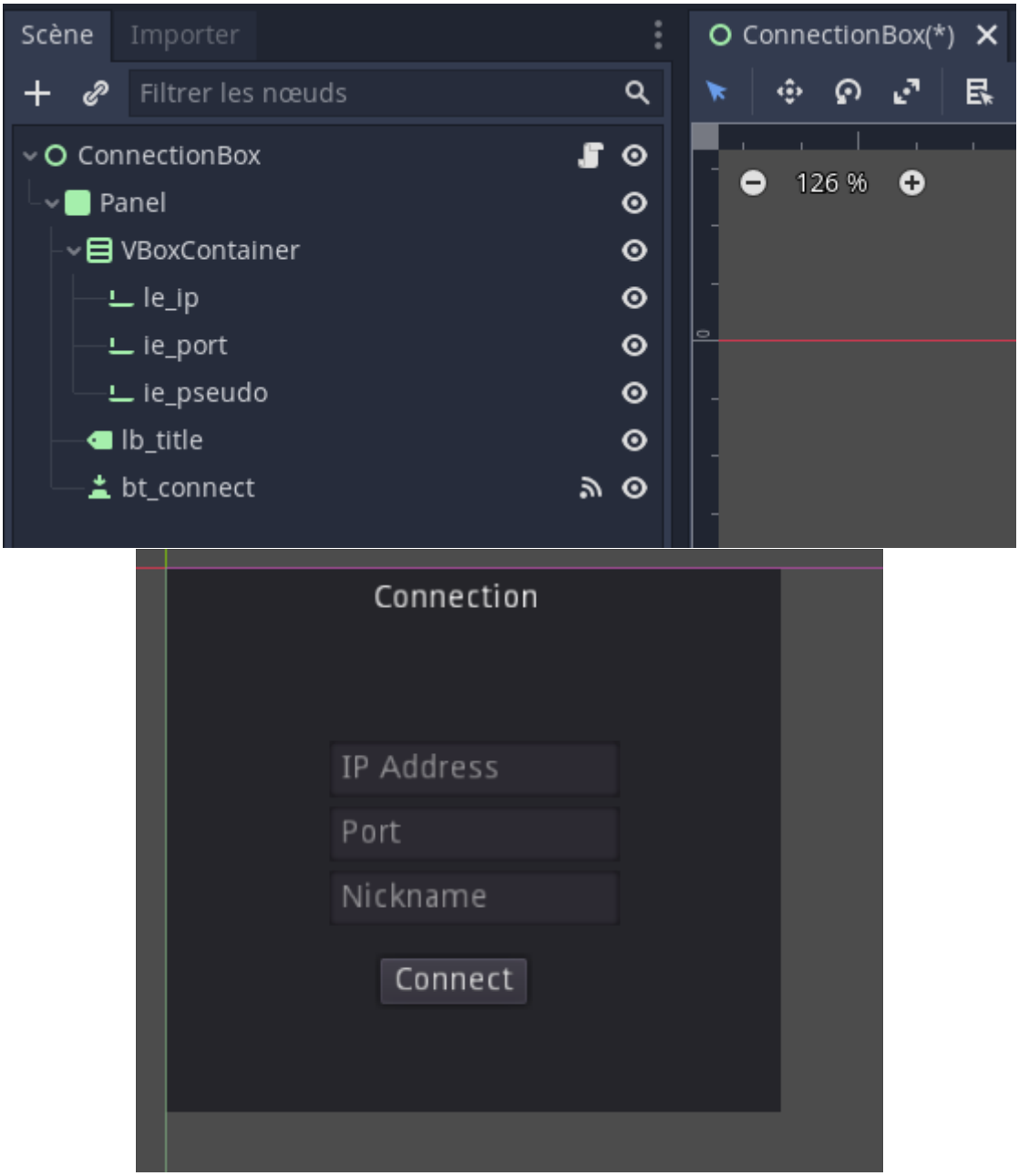

FIGURE 2 – Apparence de la Scène ConnectionBox

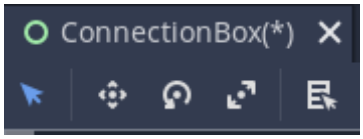

FIGURE 3 – Dans l'ordre : édition, déplacement, rotation, homothétie

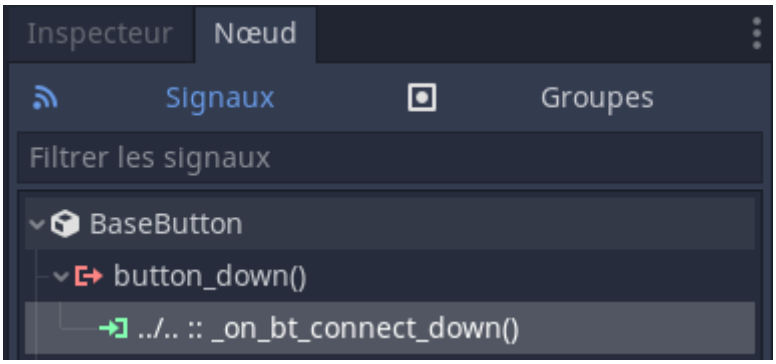

FIGURE 4 – Chaque Noeud émet des signaux en réponse à des événements.

6. Écrire le code ci-dessous :

```
# Permet d'avoir la référence des noeuds déclarés dans la Scène.
onready var input_ip = $Panel/VBoxContainer/le_ip
onready var input_port = $Panel/VBoxContainer/ie_port
onready var input_pseudo = $Panel/VBoxContainer/ie_pseudo

# Appelée une seule fois lorsque tous les noeuds sont prêts.
func _ready():
    print("ConnectionBox ready!")

# Callback du bouton
func _on_bt_connect_down():
    var ip_addr = input_ip.text
    var port = input_port.text
    var nick = input_pseudo.text
```

7. Créer un nouveau Script qu'on appellera Network. Celui-ci sera chargé d'effectuer la communication avec le serveur. Nous le créeons en tant que Singleton, pour ce faire, aller dans Projet/Paramètres/Autoload, sélectionner le script Network.gd et l'ajouter à la liste.
8. Compléter le code du script :

```
extends Node

# Variables
var my_data = {
    "pseudo": "NA",
    "pid": -1,
}

var server_info = {}
var users = {}

func _ready():
    # On connecte manuellement les signaux de la racine de l'arbre (là où se situe l'objet équivalent du socket) à des callbacks.
    get_tree().connect("connected_to_server", self, "_on_connected_to_server")
    get_tree().connect("connection_failed", self, "_on_connection_failed")
```

```
16      get_tree().connect("server_disconnected", self, "
        _on_disconnected_from_server")
17
18  func join_server(ip, port, pseudo):
19      var net = NetworkedMultiplayerENet.new()
20      my_data["pseudo"] = pseudo
21      if (net.create_client(ip, int(port)) != OK) :
22          print("Failed to create client")
23          return
24      get_tree().set_network_peer(net)
25      print("Server joined!")
26
27  func send_message_test():
28      var mess = "Hello!"
29      # envoie un appel de fonction message_from_client au serveur (id
        1)
30      rpc_id(1, "message_from_client", mess)
31
32  # le serveur peut appeler cette fonction à distance
33  remote func message_from_server(mess: String):
34      print("Message received from server: %s"%[mess])
35
36  # Callbacks
37  func _on_connected_to_server():
38      print("Connection success !")
39      var pid = get_tree().get_network_unique_id()
40      my_data["pid"] = pid
41      print("My pid is %d. Sending a test message."%[pid])
42      send_message_test()
43
44  func _on_connection_failed():
45      print("The connection has failed!")
46
47  func _on_disconnected_from_server():
48      my_data["pid"] = -1
49      print("Disconnected!")
```

4 Création du serveur

Le serveur ne sera pas doté d'interface graphique, il servira uniquement les clients.

1. Créer un autre projet, qu'on appellera TPChat_Server.
2. Créer un script, qu'on appellera Network, et écrire le code suivant :

```
1   extends Node
2
3   var SERVER_PORT = 3892
4   var MAX_PLAYERS = 16
5
6   func _ready():
7       var peer = NetworkedMultiplayerENet.new()
8       peer.create_server(SERVER_PORT, MAX_PLAYERS)
9       get_tree().network_peer = peer
10
```

```
11      get_tree().connect("network_peer_connected", self, "
        _on_network_peer_connected")
12      get_tree().connect("network_peer_disconnected", self, "
        _on_network_peer_disconnected")
13
14  remote func message_from_client(mess):
15      var pid = get_tree().get_rpc_sender_id()
16      print("Message recevied from %d: %s"%[pid, mess])
17      # on répond au client
18      rpc_id(pid, "message_from_server", "Hello from server side!")
19  func _on_network_peer_connected(id: int):
20      print("New client connected: %d"%[id])
21
22  func _on_network_peer_disconnected(id: int):
23      print("Client disconnected: %d"%[id])
```

On remarque ici la présence de la fonction message_from_client du serveur qui est appelée côté client. C'est un appel de fonction à distance. Dans notre cas, cette requête est totalement asynchrone : le client n'est pas bloqué en l'appelant. On précise également que les paramètres sont **sérialisés** avant d'être envoyés.

Tester la connexion entre le client et le serveur.

5 Améliorations

L'application créée n'est actuellement pas un chat. Un seul échange est effectué.

5.1 Une vraie zone de discussion

1. Créer une nouvelle scène ChatBox. Le coeur de cette scène sera le `RichTextLabel`, qui permettra d'écrire du texte avec `BBCode`.
2. Créer le script de cette Scène. On ne fera aucun appel direct à Network lorsqu'un message est envoyé, mais on émettra un signal à la place. Cela permet de réutiliser dans d'autres projets cette Scène.
3. Créer le script Message dérivé de Object. Il contiendra les données essentielles pour identifier un message : un identifiant message unique, un identifiant utilisateur, un timestamp, et le contenu du message.

```
1   class_name Message extends Object
2
3   var mess_id: int
4   var net_id: int
5   var timestamp: int
6   var content: String
7
8   # constructeur
9   func _init(id: int, ts: int, m: String):
10      mess_id = 0
11      net_id = id
12      timestamp = ts
13      content = m
```

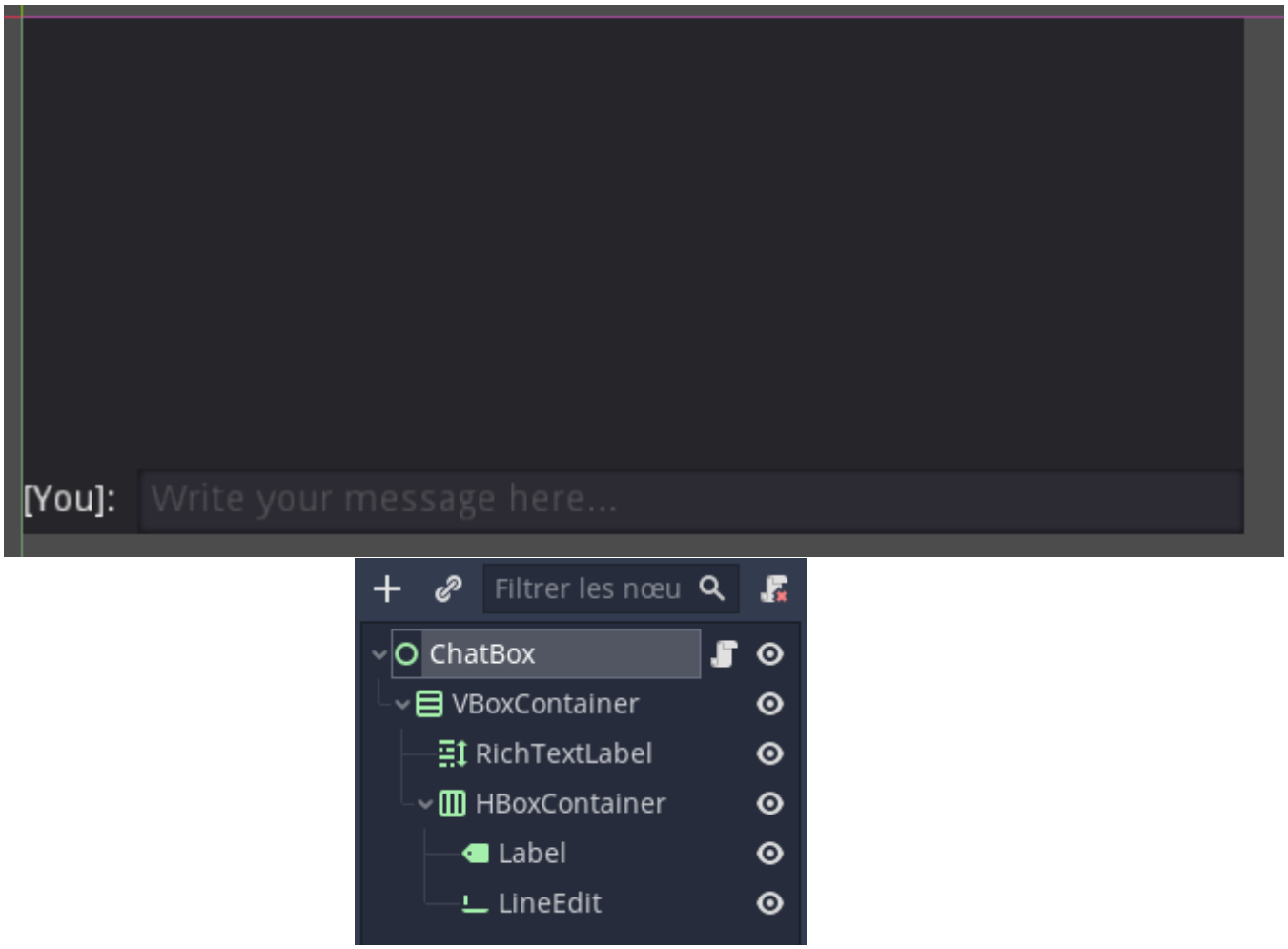

FIGURE 5 – Contenu de ChatBox

---

4. Ajouter quatre méthodes à la classe Message : to_string_display, to_string, to_dict, to_bytes. La méthode to_string_display retourne un String qui sera déstiné à être affiché à l'écran dans ChatBox, alors que to_string retourne un String décrivant le contenu de l'objet, mais dans un format destiné à être affiché dans la console.

   Note : la fonction `var2bytes` retourne un PoolByteArray contenant les octets représentant la variable.
5. Ecrire les fonctions add_message, send_message de ChatBox. La première prend en paramètre un Message et l'affiche à l'écran (dans le RichTextLabel), on pourra insérer un saut de ligne avant d'ajouter le message. La deuxième est appelée quand l'utilisateur appuie sur Entrée en écrivant quelque chose dans LineEdit, et **émet un signal contenant le texte.**. Après envoi, le contenu du LineEdit est vidé. Pour ce faire, on connect le signal `text_entered` de LineEdit sur ChatBox.
6. Créer une Scène ChatScene, qui sera la Scène principale. Elle contiendra une ConnectionBox et une ChatBox et écoutera leur signaux ainsi que les signaux de Network, puis fera les appels nécessaires des fonctions de Network.
7. Connecter le signal d'envoi de message de ChatBox à ChatScene. Lorsque ce signal est détecté, ChatScene appellera une fonction de Network (qu'on pourra appeler

send_message_to_server par exemple), qui prend la chaîne de caractères du message, qui fera le nécessaire pour le transmettre au server, c-à-d récupérer le timestamp et sérialiser les données du message en dictionnaire (le moteur s'occupe alors de la conversion en JSON). On effectuera un RPC de la fonction message_from_client du serveur

8. Côté serveur, implémenter cette fonction, et effectuer la transmission vers l'ensemble des clients. Le serveur pourra appeler un RPC de la fonction message_from_server du client.
9. Implémenter la fonction message_from_server sur le client. Un signal message_received sera émis, détecté par ChatScene, et affichera le message dans la ChatBox.

5.2 Identification des utilisateurs

Le client n'est théoriquement pas notifié automatiquement lorsqu'un autre client se connecte au serveur. C'est à ce dernier de le notifier. De plus, il est plus facile pour un utilisateur d'identifier un autre client à partir d'un pseudo qu'un grand nombre.

1. Ajouter deux fonctions register_user(user_dict) et unregister_user(pid : int) dans le client, qui pourront être appelées par le serveur (rappel : ajouter le mot-clef *remote*). Un dictionnaire `users` contiendra alors un ensemble de clefs (les identifiants) qui pointent sur un dictionnaire décrivant un client (son ID et son pseudo à minima). register_user ajoute alors une entrée au dictionnaire, et unregister_user retire l'entrée correspondant à l'id donné. Ces fonctions émettront chacune un signal qui notifiera l'application ChatScene qu'un client s'est connecté ou déconnecté.
2. Côté serveur, lorsqu'un client se connecte, il doit indiquer à tous les autres clients que ce client est désormais connecté. De plus, il doit également indiquer à ce nouveau client tous les autres clients qui sont déjà connectés. Implémenter cette fonctionnalité.
3. De retour côté client, ajouter une Scène UserList qui contient la liste des noms/id de tous les clients connectés (voir fig. 6). On pourra ainsi créer une fonction qui ajoute un Label dans la liste, une fonction qui retire un Label de la liste, une fonction qui détermine le nombre de labels dans la liste.

```
#instanciation dynamique d'un Label
var new_node = Label.instance()
#pour le retrouver plus tard, son nom sera l'id du client concerné
new_node.name = str(net_id)
# on le rajoute dans la vbox_list
list.add_child(new_node)
# et on donne un texte à ce Label !
new_node.set_text("%s (%d)"%[nickname, net_id])
```

```
# retrait dynamique d'un node par rapport à son nom.

# on cherche le noeud qui a pour nom l'id du client qui se déconnecte
var node_to_remove = list.get_node_or_null(str(net_id))
if node_to_remove:
    # si on l'a trouvé, alors on le retire
    node_to_remove.queue_free()
```

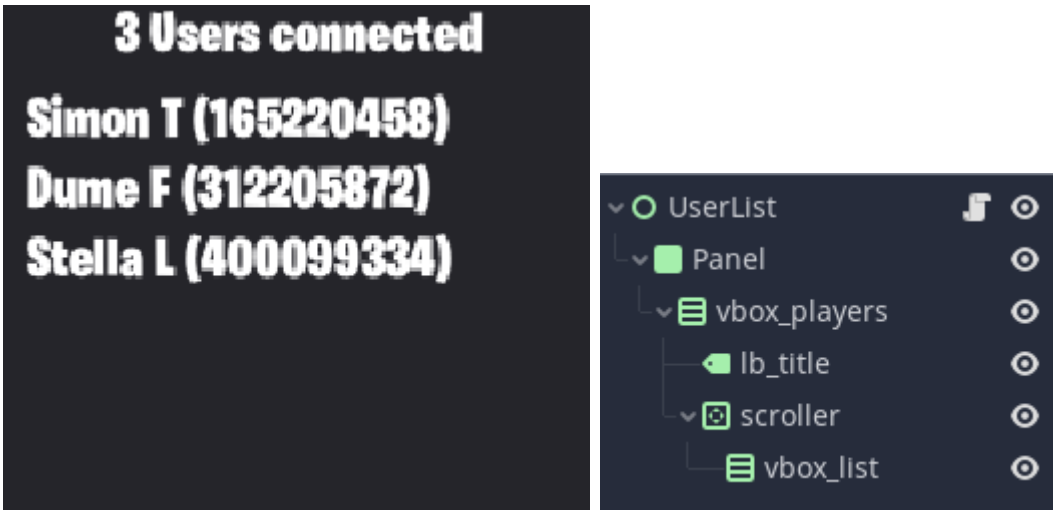

FIGURE 6 – Scène UserList avec 3 clients.

```
# count the numbers of children in the list
func update_nb_users():
    var nb = list.get_child_count()
    set_nb_users(nb)

func set_nb_users(n: int):
    lb_title.text = "%d users connected"%[n]
```

5.3 Synchronisation des horloges

Même si cela n'a pas de réel intérêt pour une simple application de discussion, cela peut l'être pour un jeu qui se joue en temps réel.

Le serveur est maître de son contenu, et le temps en fait partie. L'heure et la date peuvent être radicalement différents entre le client et le serveur, mais cela ne doit pas compromettre la connexion. En gardant en tête le principe de suprématie du serveur, le client doit donc pouvoir à tout moment disposer de l'heure du serveur **sans lui envoyer une requête.**

1. Lorsque la connexion est validée, ajouter un RPC de la fonction fetch_server_time. On mettra en paramètre la timestamp du client au moment de l'envoi.

```
rpc_id(1, "fetch_server_time", OS.get_system_time_msecs())
```

2. Implémenter la fonction fetch_server_time sur le serveur. Celle-ci récupère son timestamp, et renvoie au client son timestamp ainsi que celui du serveur.
3. Lorsque le client reçoit ces informations, il doit déterminer le temps écoulé lors de l'échange, et ajouter le même nombre de millisecondes au timestamp du serveur. Il peut alors l'affecter à une variable de Network.

4. Implémenter la fonction _physics_process(delta : float). Elle est appelée en moyenne 60 fois par seconde. delta est le temps écoulé en **secondes**. Dans cette fonction, on incrémente la variable contenant le temps serveur d'autant de ms écoulées. Attention : delta est un float. On prendra donc la partie entière et on stocke la partie décimale dans une autre variable qu'on incrémentera, et lorsqu'elle est supérieure à 1, cela voudra dire qu'une milliseconde supplémentaire se sera écoulée.

```
func _physics_process(delta: float) -> void:
    client_clock += int(delta*1000)
    decimal_collector += (delta * 1000) - int(delta * 1000)
    if decimal_collector >= 1.0:
        client_clock += 1
        decimal_collector -= 1
```

Cependant, cette synchronisation est unique et ne tient pas compte des variations de la latence.

5.4 Évaluation de la latence

La commande *ping* est réalisée avec le protocole ICMP (*Internet Control Message Protocol*), encapsulé dans un datagramme IP. Elle sert à vérifier l'accessibilité d'une autre machine, et évalue le temps de réponse, désigné comme le *round-time trip*.

Dans notre programme, le client enverra périodiquement une requête au serveur. On utilisera alors à bon escient le node Timer [2].

1. Ajouter **dynamiquement** un Timer dans Network lorsque la connexion au serveur est validée. Celui-ci émettra un signal toutes les demi-secondes qui se connecte à la fonction determine_latency, qui appelle la fonction homonyme du serveur avec un timestamp.

```
var timer = Timer.new()
timer.wait_time = 0.5
timer.name = "ping_timer"
timer.autostart = true
timer.connect("timeout", self, "determine_latency")
```

```
func determine_latency():
    rpc_id(1, "determine_latency", OS.get_system_time_msecs())
```

2. Lorsque cette fonction est appelée sur le serveur, ce dernier renvoie alors le même timestamp au client.

```
# coté serveur
remote func determine_latency(client_timestamp):
    var player_id = get_tree().get_rpc_sender_id()
    rpc_id(player_id, "return_latency", client_timestamp)
```

2. https://docs.godotengine.org/en/stable/classes/class_timer.html

3. Nous allons implémenter ici une version simplifiée de l'algorithme de Marzullo, utilisé par le protocole SNTP [3], proposée par Zachary Booth Simpson [4] [5].

Algorithme 1 : Algorithme de détermination de la latence

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
$m \longleftarrow mediane(L)$
Retirer de L toutes les valeurs L_i supérieures à $2m$.
La latence estimée est $moyenne(L)$

Note : il n'est pas nécessaire d'utiliser un tant que ici, car les requêtes sont asynchrones. On déclarera en outre dans Network les variables `latency, delta_latency, latency_array`.

On pourra donner la fonction appelée par le serveur suivante (incomplète) :

```
remote func return_latency(client_timestamp):
    latency_array.append((OS.get_system_time_msecs() -
    client_timestamp)/2)

    if latency_array.size() < 9:
        return

    # évaluation de la latence appelée new_latency
    # ...
    # ...


    delta_latency = new_latency - latency
    latency = new_latency

    emit_signal("ping_updated", latency, delta_latency)
    latency_array.clear()
```

On pourra alors améliorer la fonction _physics_process pour tenir compte de cette différence de latence évaluée, en moyenne toutes les 5 secondes.

```
func _physics_process(delta: float) -> void:
    client_clock += int(delta*1000) + delta_latency
    delta_latency -= delta_latency
```

3. *Simple Network Time Protocol*
4. *A Stream-based Time Synchronization Technique For Networked Computer Games, 1er mars 2000, disponible à l'adresse https://web.archive.org/web/20160310125700/http://mine-control.com/zack/timesync/timesync.html*
5. Si le lien n'est pas accessible, essayer celui-ci : https://timonpost.medium.com/game-networking-2-time-tick-clock-synchronisation-9a0e76101fe5

```
    decimal_collector += (delta * 1000) - int(delta * 1000)
    if decimal_collector >= 1.0:
        client_clock += 1
        decimal_collector -= 1
```