

# Projet Programmation 2 – Rapport de la partie 3

## Multijoueur et réseau

DABY-SEESARAM Arnaud, VILLANI Neven

### 1 Prise en compte des remarques des parties précédentes

Afin de prendre en compte les remarques effectuées dans les précédents retour que l'on a eu sur les parties 1 et 2 du projet, nous avons pris soin de mieux commenter le code et de corriger des bugs connus (par exemple celui indiqué dans `items_item` du fichier de commandes).

À présent, dans les `match` effectuées sur la longueur de la commande, un commentaire est ajouté pour chaque cas explicitant l'état de la commande à l'exécution du bloc en question.

### 2 Problèmes préliminaires rencontrés

Bien que nous ayons beaucoup séparé notre code en fichiers pour chaque aspect du jeu, nous avons découvert en débutant cette partie que certains aspects étaient trop mélangés.

Par exemple notre classe `Pos` s'occupait à la fois d'aspects UI (réactions aux clics, affichage graphique) et logiques (ensemble d'organismes par case, items, combats). Plus difficile encore, `BodyPart` s'occupait à la fois d'éléments relatifs au jeu (items actifs, organismes vivants) et au joueur (progrès de l'objectif, position).

Il nous a donc fallu commencer par découpler ces rôles :

- `Room`, `BodyPart`, `Game`, `Pos` : logique du jeu
- `DisplayGrid`, `DisplayPos` : affichage graphique
- `LocalRoom`, `LocalPos` : dialogue entre `Room`, `Pos` et `DisplayRoom`, `DisplayPos`

`Local*` contiennent les informations absolument minimales pour l'affichage graphique, `Display*` en sont juste des versions statiques mises à jour régulièrement.

Nous avons par ailleurs profité de cette séparation pour extraire une partie de `BodyPart` dans `Game`, contenant tout ce qui est relatif à un seul joueur. Nous avons fait attention à ce que toutes ces modifications rendent `BodyPart` agnostique au nombre de `Game` qui existent, et donc en particulier qu'une fois la partie réseau mise en place il soit facile de rendre le jeu multijoueur.

À part la performance, rien ne limite le nombre de joueurs à condition de mettre à jour les fichiers d'initialisation de niveaux. Le jeu a été testé à 1, 2 et 3 joueurs.

Cette séparation a pris la majeure partie du temps alloué à la troisième partie, nous aurions aimé avoir un peu plus de temps pour d'autres améliorations mineures. Par exemple, une partie du temps imparti a été utilisée pour essayer de séparer les commandes à effectuer côté serveur et celles à effectuer côté client. L'idée était la suivante : les commandes n'interagissant pas avec le jeu (comme (`help`)) devraient être effectuées par le client et toute commande interactive devrait :

- effectuer les jeux de questions / réponses en local
- envoyer la commande finale au serveur
- recevoir la réponse du serveur

Cette tentative n'a pas abouti et pour terminer à temps nous avons mis toutes les commandes côté serveur.

### 3 Réalisation et ajouts

Côté client, un thread est responsable de dialoguer avec le serveur, parser les données reçues, et sérialiser les données à envoyer, un autre met à jour l'affichage graphique et enregistre les événements (clics, mouvements, commandes).

Côté serveur il se passe le même processus de sérialisation et désérialisation, et après chaque étape du jeu la grille est lue et envoyée à chaque client, puis une courte pause est faite jusqu'à ce que tous les clients aient

confirmé la bonne réception et éventuellement envoyé des commandes à exécuter.  
(Il y a des sécurités contre des clients qui mettraient trop longtemps à répondre)

L’affichage graphique distingue les virus du joueur de ceux des autres joueurs, mais ne distingue pas les adversaires les uns des autres dans le cas de trois joueurs ou plus.

Le système de niveaux et d’objectifs a été conservé : chaque joueur doit compléter les missions, et une fois que l’un d’eux a atteint 100% chaque joueur gagne autant de points que le pourcentage d’avancement de sa mission. Une fois le dernier niveau terminé le joueur avec le plus de points gagne.

Heureusement les objectifs ainsi que la création de niveaux ont été très faciles à rendre multijoueurs.

Au bout d’un certain temps, si un joueur est inactif il est déclaré forfaitaire : le jeu reprend sans lui. Ses virus sont toujours générés, mais son curseur ne bouge plus.

À l’initialisation du jeu, une fenêtre s’ouvre, affichant les options de connexion par défaut (ip de l’hôte, port et timeout de connexion). Cette fenêtre permet d’éditer les options de connexion du client. Si la connexion ne peut pas se faire, une nouvelle fenêtre s’ouvre pour en informer l’utilisateur, qui peut soit changer les valeurs sus-décrites, soit quitter l’application.

## 4 Installation

La procédure de lancement du jeu (et la structure du projet) ont été un peu compliquées par l’ajout d’un serveur, mais elles sont également des preuves que la séparation serveur/client a été bien réalisée.

**server/** contient le code qui exécute le jeu, **client/** contient la partie graphique. Ils partagent un fichier de code par un lien symbolique.

L’exemple ci-dessous décrit comment lancer une partie à deux joueurs, il s’étend facilement à 1 ou 3 joueurs.

À exécuter une seule fois :

```
$ cp -r player{-example,1}
$ cp -r player{-example,2}
$ echo "8001\n8002" > server/server.cfg
```

Au lancement une fenêtre de dialogue s’ouvrira, il faudra renseigner **host=127.0.0.1** (ou autre), **port=8001** pour le joueur 1, **port=8002** pour le joueur 2, **timeout=20000** (ou autre), puis éventuellement enregistrer les préférences.

À exécuter à chaque nouvelle partie, dans trois shells différents :

```
$ cd server && sbt run
$ cd player1 && sbt run
$ cd player2 && sbt run
```

Le dossier **src** de **player-example** est un symlink vers **client/src**, de même que nous avons choisi que **client/\*\*/communication.scala** soit un symlink vers **server/\*\*/communication.scala** pour que la partie dialogue entre le serveur et le client n’ait à être modifiée qu’à un seul endroit.

En particulier cela permet sans aucune duplication de code que **client** et **server** partagent la même classe **Connection**, et que les sérialisations et désérialisations de messages soient groupées ensemble.

## 5 Limitations et améliorations envisagées

- nous avons envisagé de rétablir un système de **play/pause** pour pouvoir interrompre le jeu, possiblement lui ajouter des restrictions pour empêcher un utilisateur de gêner les autres en mettant le jeu en pause à répétition
- les items qui sont envoyés entre le serveur et le client sont des chaînes de caractères avec séparateurs fixés comme **"/"/** et **"|"**. Cela n’est pas dangereux car l’utilisateur ne peut envoyer que des commandes qui sont validées par **Command** et les messages sont encapsulés dans un block **try**, mais si un ajout d’une autre fonctionnalité permettait au client d’envoyer au serveur autre chose que des commandes il faudrait ajouter d’autres mécanismes de validation pour éviter des injections sous la forme de l’envoi

- d'une commande "|||<commande malicieuse>" qui serait découpée sur le "|||" et interprétée comme une commande vide provenant de l'utilisateur suivie d'un message provenant du client.
- certaines commandes pourraient être déplacées côté client : il pourrait ne pas y avoir besoin de faire appel au serveur pour utiliser **help**.
  - les commandes **load**, **save** et **level** sont interdites si il y a deux joueurs ou plus, on pourrait les autoriser si tous les joueurs sont en faveur. Cela supposerait de mettre à jour le format de fichier de sauvegarde pour accomoder plusieurs joueurs.
  - pour l'instant si un message est perdu en route ou corrompu il est ignoré, on pourrait faire en sorte que le receveur envoie une demande d'obtenir un second message pour remplacer celui corrompu.