Rapport Technique – Projet CodingGame S2

Titre : Développement d’un jeu de réflexion multijoueur en ligne

Projet : Tic Tac Toe (Morpion)

Langage utilisé : C# avec .NET

Auteurs : [Nom(s) de l’étudiant ou des membres du binôme]

Date : [À compléter]

# 1. Introduction

Ce projet s’inscrit dans le cadre du module "CodingGame S2", dont l’objectif est de concevoir un jeu de réflexion multijoueur intégrant un serveur centralisé, une synchronisation en temps réel, ainsi que la gestion des conflits et des règles du jeu. Le projet interdit explicitement l’usage de frameworks de développement de jeux en ligne, imposant ainsi une implémentation bas niveau de la communication et de la logique de jeu.  
  
Le jeu choisi pour cette implémentation est le Tic Tac Toe (ou Morpion), un jeu de société simple, mais adapté à l’expérimentation d’une architecture client-serveur. Le projet propose à la fois un mode multijoueur en ligne et un mode solo contre une intelligence artificielle rudimentaire, avec une interface graphique interactive.

# 2. Architecture générale

L’application est structurée en quatre modules principaux, chacun sous forme de projet indépendant dans une solution .NET.  
  
Le premier module est le serveur, qui constitue le cœur de l’application. Il gère les connexions entrantes à l’aide de la classe TcpListener et affecte à chaque client connecté un thread distinct. Lorsque deux clients sont connectés, une session de jeu est créée et gérée via la classe GameSession, qui contient toute la logique métier du jeu, y compris la gestion du plateau, la validation des coups, l’alternance des tours, ainsi que la détection des conditions de victoire ou de match nul.  
  
Le deuxième module est un client console. Il permet d’interagir avec le serveur en mode texte. Le client se connecte au serveur via une socket TCP, affiche le plateau de manière textuelle et permet au joueur de saisir ses coups via la console.  
  
Le troisième module est un client graphique basé sur WinForms. Ce client propose une interface utilisateur simple composée d’un menu de choix (mode solo ou multijoueur), d’une grille interactive permettant de jouer, et d’un affichage dynamique de l’état de la partie. L’interface s’adapte aux messages reçus du serveur et réagit aux actions de l’utilisateur.  
  
Le quatrième module est un projet partagé appelé Shared, contenant la définition de la classe Message. Cette classe est utilisée pour standardiser les échanges entre les clients et le serveur sous la forme de messages sérialisés en JSON.

# 3. Représentation des données et échanges réseau

Les communications entre le serveur et les clients sont établies par des sockets TCP. Chaque message échangé est représenté sous forme d’objet JSON conforme à la structure suivante :  
  
public class Message  
{  
 public string Type { get; set; }  
 public string Content { get; set; }  
}  
  
Le champ Type désigne la nature du message (par exemple : MOVE, STATE, YOUR\_TURN, END, etc.), tandis que Content contient les données utiles à ce message. Par exemple, un message de type MOVE contient les coordonnées du coup joué sous la forme d’une chaîne "1,2", tandis qu’un message de type STATE contient la représentation actuelle du plateau sous la forme d’une chaîne de neuf caractères (par exemple, "XOX.O....").  
  
L’utilisation de JSON permet une sérialisation et désérialisation automatique, et rend les échanges lisibles et extensibles. Chaque ligne transmise via les flux réseau (StreamWriter / StreamReader) correspond à un message complet.

# 4. Gestion de l’état du jeu et des conflits

L’état du jeu est entièrement géré côté serveur dans la classe GameSession. Le plateau de jeu est représenté sous forme d’un tableau char[3,3]. Chaque coup reçu est d’abord vérifié par le serveur pour s’assurer qu’il provient du bon joueur (c’est-à-dire celui dont c’est le tour), que les coordonnées sont valides, et que la case ciblée est libre.  
  
Cette stratégie garantit l’intégrité de la partie, puisque le client ne peut pas modifier l’état de manière autonome. En cas de tentative invalide (ex : jeu en dehors du tour ou case déjà prise), le serveur retourne un message d’erreur de type ERROR.  
  
Après chaque coup valide, le serveur met à jour le plateau, vérifie si une condition de victoire ou d’égalité est atteinte, et renvoie l’état mis à jour à tous les clients associés à la session. Cela permet une synchronisation en temps réel de l’état de la partie.

# 5. Intelligence artificielle pour le mode solo

Pour permettre au joueur de jouer sans adversaire humain, une intelligence artificielle simple a été développée dans la classe AIPlayerHandler, qui hérite de PlayerHandler. Lorsqu’un seul joueur est connecté, le serveur crée automatiquement une session de jeu entre ce joueur et l’IA.  
  
L’IA suit une logique basique mais fonctionnelle :  
1. Si une case permet de gagner immédiatement, elle est choisie.  
2. Si une case permet de bloquer une victoire adverse, elle est choisie.  
3. Sinon, l’IA joue dans la première case vide disponible.  
  
Cette IA est déterministe et ne possède pas de stratégie avancée, mais elle permet de tester l’ensemble de la logique du jeu en mode solo.

# 6. Interface utilisateur

Le client graphique développé avec WinForms offre une expérience utilisateur simple mais efficace. Un formulaire d’accueil permet au joueur de choisir entre le mode solo et le mode multijoueur.  
  
Une fois la partie lancée, une grille 3x3 interactive permet de jouer en cliquant sur les cases. L’état du plateau est mis à jour en fonction des messages STATE envoyés par le serveur. Des boîtes de dialogue informent le joueur des événements importants (début de partie, erreurs, victoire, égalité, etc.).  
  
Cette interface a été conçue pour être minimaliste, réactive et facile à prendre en main, tout en respectant les contraintes du projet.

# 7. Cas de test et utilisation

Différents scénarios ont été testés pour valider le bon fonctionnement de l’application :  
  
- Connexion de deux clients : une partie démarre automatiquement dès que deux joueurs sont connectés. Les coups sont synchronisés, et la fin de partie est bien gérée.  
- Connexion d’un seul client : une session contre l’IA est lancée. L’IA réagit correctement aux coups du joueur.  
- Scénarios de victoire, d’égalité et de tentatives de triche ont été simulés. Le serveur gère tous les cas correctement.  
  
Le système a été testé en local sur une même machine avec plusieurs fenêtres console ou graphiques. Aucun framework externe n’a été utilisé pour la communication ou la logique réseau.

# 8. Conclusion

Ce projet a permis de mettre en œuvre une architecture client-serveur fonctionnelle à bas niveau, en utilisant uniquement les primitives réseau fournies par .NET. La gestion des connexions, de la synchronisation et des règles du jeu a été conçue de manière robuste et extensible.  
  
L’ajout du mode solo avec une IA simple constitue un bonus notable, tout comme l’intégration d’une interface graphique claire et interactive.  
  
Plusieurs axes d’amélioration sont possibles, notamment l’ajout d’une IA plus sophistiquée (par exemple via l’algorithme Minimax), la persistance des scores, ou encore le déploiement du jeu sur un réseau local ou via le web.  
  
Le projet satisfait l’ensemble des contraintes du sujet, et démontre une bonne maîtrise des notions de communication réseau, de gestion de l’état distribué, et de programmation événementielle.