Rapport du projet UE CALC
Rabbit Invaders

NICOTTE Loann
BAYLE Sei

Janvier 2026

IMT Atlantique

Table des matières

1 Introduction

Dans le cadre de l'UE CALC, nous avons eu à réaliser un projet utilisant **RabbitMQ**, un logiciel d'agent de messages (*Message Broker*) open source. L'objectif était de mettre en évidence les fonctionnalités de ce middleware pour le développement d'applications distribuées, notamment la tolérance aux pannes et le découplage des composants.

Pour ce projet, nous avons choisi de revisiter un classique du jeu vidéo : **Space Invaders**. Ce choix s'explique par la nature "temps réel" du jeu qui représente un défi intéressant pour un bus de messages : le jeu doit rester fluide malgré le passage de toutes les informations (positions, tirs, scores) par le réseau RabbitMQ.

Ce document présente l'architecture retenue, l'implémentation technique et les bénéfices apportés par RabbitMQ à notre application.

2 Buts du projet

Ce projet a pour but de réaliser un jeu de type Space Invaders entièrement distribué. Il a été réalisé en **Python 3**, avec les bibliothèques **Pygame** (pour l'affichage et la gestion des événements) et **Pika** (pour la communication AMQP).

Le cahier des charges que nous nous sommes fixé est le suivant :

— Séparation stricte entre la logique du jeu (Moteur), l'affichage (Viewer) et les contrôles (Controller).

— Capacité du système à survivre à la perte d'un composant (ex : l'écran s'éteint, le jeu continue).

— Gestion d'un cycle de vie complet (Jeu, Victoire, Défaite, Replay).

Nous n'avons pas implémenté de menus complexes ou de gestion de sauvegarde pour nous concentrer sur l'aspect distribué et la robustesse de l'architecture.

3 Utilisation de RabbitMQ et architecture

3.1 Architecture du code

Le code est structuré en plusieurs modules indépendants qui ne partagent aucune mémoire. Ils s'exécutent dans des processus distincts :

— **Engine (`engine.py`)** : C'est le serveur de jeu. Il maintient l'état du monde (`world_state` : positions des aliens, du joueur, des balles). Il calcule les collisions et les déplacements. Il tourne dans une boucle infinie cadencée.

— **Viewer (`viewer.py`)** : C'est un client "bête". Il ne fait aucun calcul de jeu. Il se contente d'écouter les mises à jour de position et de dessiner les sprites correspondants à l'écran.

— **Controller (`controller.py`)** : Il capture les entrées clavier de l'utilisateur et les transforme en messages de commande.

3.2 Ajout de RabbitMQ

RabbitMQ joue le rôle de la "colonne vertébrale" de notre application. Aucun composant ne se parle directement (pas de sockets TCP directs entre eux). Tout passe par un **Exchange** unique nommé `space_invaders_topic` configuré en mode **Topic**.

RabbitMQ a été incorporé de 3 manières principales :

1. **Transfert de l'input** : La manette envoie les ordres (GAUCHE, DROITE, TIR) au moteur.
2. **Diffusion de l'état du jeu** : Le moteur envoie 60 fois par seconde la position de tous les objets au Viewer.
3. **Gestion du cycle de vie** : Le moteur signale les événements majeurs (WIN, LOSE, QUIT) via des messages spécifiques.

3.3 Les queues et topics utilisés

Nous utilisons un Exchange de type **Topic** pour permettre un routage flexible basé sur des clés (*Routing Keys*). Voici les clés utilisées :

— `game.input` : Utilisée par le *Controller* pour envoyer des commandes JSON (ex : `{"action": "SHOOT"}`). Le *Engine* écoute cette clé.
— `game.state` : Utilisée par le *Engine* pour publier l'état complet du monde. Le *Viewer* s'y abonne pour rafraîchir l'écran.
— `game.lifecycle` : Utilisée pour les événements rares mais critiques (Victoire, Défaite). Le *Viewer* écoute cette clé pour afficher les messages de fin de partie ("GAME OVER").

4 Intérêts de RabbitMQ

4.1 Découplage total (Input)

En utilisant RabbitMQ pour transférer les inputs, nous pouvons changer de périphérique d'entrée sans toucher au code du jeu. Nous pourrions remplacer le script `controller.py` par une interface Web ou une IA qui joue toute seule, tant qu'elle envoie les bons messages JSON sur la queue `game.input`.

4.2 Tolérance aux pannes (Affichage)

C'est le point fort de notre architecture. Si le composant `viewer.py` crashe (erreur graphique, fermeture de fenêtre), le `engine.py` n'est pas affecté. Il continue de faire "vivre" le monde et d'envoyer des messages. Dès qu'on relance un `viewer.py`, il récupère instantanément le flux `game.state` et affiche la partie en cours. Cela prouve la robustesse du système asynchrone.

4.3 Flexibilité (Reconfiguration dynamique)

Grâce au mode **Topic** et au pattern Publish/Subscribe, nous pouvons lancer plusieurs Viewers en même temps. Ils afficheront tous la même partie de manière synchronisée sans que le Moteur ait besoin de gérer plusieurs connexions. C'est RabbitMQ qui duplique les messages pour chaque abonné.

5 Le code de RabbitMQ

Pour garder le code propre, nous avons factorisé les appels RabbitMQ dans un fichier utilitaire `rabbit_utils.py`, inspiré de l'approche modulaire vue en cours.

Voici un extrait de la fonction d'envoi utilisée par tous les composants :

```python
def publish_message(routing_key, data):
    """ Publie un message JSON sur l'exchange Topic """
    connection = get_connection()
    channel = connection.channel()

    # Declaration de l'exchange pour etre sur qu'il existe
    channel.exchange_declare(exchange=EXCHANGE_NAME,
                             exchange_type='topic')

    channel.basic_publish(
        exchange=EXCHANGE_NAME,
        routing_key=routing_key,
        body=json.dumps(data)
    )
    connection.close()
```

Listing 1 – Fonction d'envoi générique

Côté réception, nous utilisons des **Threads** pour ne pas bloquer l'affichage ou le calcul physique lors de l'attente des messages (fonction `basic_consume` bloquante).

6 Conclusion

En conclusion, ce projet nous a permis de mettre en pratique les concepts d'architectures orientées messages. Nous avons transformé un jeu monolithique (Space Invaders) en un système distribué résilient.

L'utilisation de RabbitMQ a complexifié légèrement l'infrastructure initiale (besoin de lancer un serveur de messages), mais elle a grandement simplifié la logique du code en séparant les responsabilités. Le Moteur ne se soucie plus de "comment afficher", et la Manette ne se soucie plus de "qui reçoit l'ordre".

Cette architecture pourrait facilement être étendue, par exemple en ajoutant un microservice de **Son** qui écouterait les événements de tir (`game.input`) ou d'explosion pour jouer des bruitages, sans jamais modifier le moteur de jeu existant.