IMT Atlantique
Technopôle de Brest-Iroise - CS 83818
29238 Brest Cedex 3
URL : www.imt-atlantique.fr

IMT Atlantique, ILSD* UE CALC

Plateforme d'échecs distribuée basée sur une architecture événementielle avec RabbitMQ

Marion Boulin

Date d'édition : 5 décembre 2025
Version : 1.0

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Sommaire

1. Contexte et objectifs du projet

Dans le cadre de cette unité d'enseignement, j'ai réalisé un projet portant sur les architectures distribuées orientées événements. J'ai choisi de travailler sur une plateforme d'échecs distribuée, car le déroulement d'une partie se décompose naturellement en une suite d'événements successifs, tels que le début de la partie, les coups joués et la fin de la partie.

L'objectif de ce projet était de mettre en pratique les notions de communication asynchrone, de découplage entre composants, de tolérance aux pannes et de reconfiguration dynamique. Pour cela, j'ai utilisé RabbitMQ comme middleware de messagerie afin d'assurer la diffusion des événements entre les différents composants du système.

2. Architecture globale du système

J'ai conçu le système selon une architecture orientée événements. Un programme producteur unique est chargé de gérer le déroulement de la partie d'échecs. Ce producteur génère les événements de jeu et les publie dans RabbitMQ, sans avoir connaissance des services qui vont les consommer.

Les services consommateurs sont indépendants les uns des autres et ne communiquent jamais directement avec le producteur. Cette séparation claire des responsabilités m'a permis de mettre en évidence un fort découplage entre les composants, ainsi qu'une meilleure modularité du système.

3. Mise en place et configuration de RabbitMQ

J'ai déployé RabbitMQ localement à l'aide de Docker en utilisant l'image incluant le plugin de management. Cette configuration m'a permis d'accéder à l'interface web de RabbitMQ afin de visualiser les exchanges, les queues et l'état des messages en temps réel.

À partir de cette interface, j'ai créé un exchange nommé `chess.events` de type `fanout`. Ce type d'exchange permet de diffuser chaque événement à l'ensemble des services consommateurs. J'ai ensuite créé une queue dédiée pour chaque service (`validation.queue`, `analysis.queue`, `storage.queue` et `spectator.queue`) et je les ai liées manuellement à l'exchange.

Ces manipulations m'ont permis d'observer concrètement le fonctionnement de RabbitMQ. Par exemple, lorsque le producteur était lancé avant les services consommateurs, j'ai pu constater que les messages s'accumulaient dans les queues. Lorsque les consommateurs étaient démarrés ultérieurement, les messages étaient alors consommés automatiquement.

4. Modélisation des événements

Les échanges entre les composants reposent sur des messages encodés en JSON. J'ai défini une structure commune à tous les messages, comprenant notamment le type d'événement, un identifiant de partie et un horodatage. Cette structure est construite dans le

code du producteur, lors de l'appel à la fonction chargée de publier les événements.

Par exemple, lors de l'envoi d'un coup joué, le message contient les cases de départ et d'arrivée ainsi que le joueur concerné. Cette structure commune permet à tous les services consommateurs d'interpréter les messages de manière cohérente, sans dépendre de l'implémentation du producteur.

5. Implémentation du producteur

J'ai implémenté le producteur en Python en utilisant la bibliothèque `pika`. Dans le code, le producteur se connecte à RabbitMQ, déclare l'exchange `chess.events`, puis publie les événements à l'aide de la méthode `basic_publish`. Le producteur n'attend aucune réponse après l'envoi d'un message, ce qui garantit une communication entièrement asynchrone.

Lors de l'exécution du producteur, j'ai observé dans l'interface RabbitMQ que chaque événement publié était bien dupliqué dans l'ensemble des queues associées aux services consommateurs. Cette observation m'a permis de valider le bon fonctionnement de la topologie mise en place.

6. Implémentation des services consommateurs

J'ai développé plusieurs services consommateurs, chacun étant implémenté sous la forme d'un programme Python indépendant. Chaque service se connecte à RabbitMQ, déclare sa propre queue et consomme les messages à l'aide d'une fonction de callback.

Le service de validation traite les événements de type `move_played` et affiche les coups reçus. Le service d'analyse simule un traitement plus long en introduisant une temporisation dans le callback, ce qui m'a permis de vérifier que le producteur continuait à publier les événements sans être bloqué. Le service d'enregistrement reconstruit la partie complète à partir des événements et écrit les coups dans un fichier texte lors de la réception de l'événement de fin de partie. Cette approche correspond à un mécanisme d'event sourcing simplifié.

Enfin, j'ai ajouté un service spectateur qui se contente d'afficher les messages reçus. Ce service peut être lancé ou arrêté à tout moment, sans modification du reste du système, ce qui illustre la reconfiguration dynamique permise par l'architecture.

7. Tests de communication asynchrone et de tolérance aux pannes

Afin de valider le comportement du système, j'ai réalisé plusieurs manipulations. J'ai notamment lancé le producteur sans démarrer les services consommateurs, puis observé l'accumulation des messages dans les queues via l'interface RabbitMQ. J'ai ensuite démarré les consommateurs et constaté que les messages étaient traités a posteriori.

J'ai également simulé une panne en arrêtant volontairement un service consommateur pendant l'exécution du producteur. Les messages non acquittés sont restés stockés

dans la queue correspondante et ont été retraités automatiquement lors du redémarrage du service. Ces tests m'ont permis de vérifier concrètement la tolérance aux pannes du système.

8. Conclusion

Ce projet m'a permis de mettre en œuvre une architecture distribuée orientée événements et de comprendre concrètement le rôle d'un middleware de messagerie tel que RabbitMQ. Les différentes manipulations réalisées m'ont permis d'illustrer la communication asynchrone, le découplage entre composants, la tolérance aux pannes et la reconfiguration dynamique.

La plateforme d'échecs développée constitue un cas d'étude simple mais pertinent pour illustrer les concepts abordés dans cette unité d'enseignement.