Jeu de Mozart Distribué - Système Akka Résistant aux Pannes

Vue d'ensemble

Ce projet implémente le Jeu de Mozart dans un système distribué utilisant Akka. Le système supporte 4 musiciens s'exécutant en parallèle et assure une résistance complète aux défaillances.

Architecture de l'Acteur Musicien

Chaque musicien est implémenté comme un acteur Akka (`Musicien`) contenant des acteurs enfants spécialisés : - **DisplayActor** : Gestion de l'affichage et logs
- **PlayerActor** : Lecture MIDI des mesures musicales
- **DataBaseActor** : Accès aux mesures de Mozart (chef uniquement)

Gestion des Pannes du Chef d'Orchestre

Algorithme d'Élection Bully

Quand le chef tombe en panne, les musiciens restants utilisent un **algorithme d'élection bully** :

1. **Détection de panne** : Les musiciens détectent la panne via `Terminated` messages
2. **Élection automatique** : Basée sur `creationTime` (le plus ancien), en cas d'égalité sur l'ID (le plus petit)
3. **Annonce** : Le nouveau chef s'annonce via `ChefAnnouncement`

Système de Heartbeat

- Le chef envoie des battements de cœur (`ChefHeartbeat`) toutes les secondes
- Découverte automatique des chefs via `WhoIsChef` / `ChefHere`

Résistance aux Défaillances

Supervision Strategy

```
override val supervisorStrategy = OneForOneStrategy() {
  case _: Exception => Restart
}
```

- **Redémarrage automatique** des acteurs enfants crashés
- **Tolérance transparente** : la musique continue malgré les pannes locales

Gestion des Pannes de Musiciens

- **Détection** : Le chef surveille les musiciens via `context.watch()`

- **Nettoyage** : Suppression des musiciens crashés des files d'attente
- **Continuité** : Sélection automatique du musicien suivant pour continuer

Règles de Timeout (30 secondes)

- **Attente initiale** : Le chef attend 30s minimum 1 musicien
- **Solitude** : Si le chef reste seul, il attend 30s avant d'arrêter
- **Reconnexion** : Les musiciens peuvent se reconnecter et relancer la performance

Fonctionnalités Avancées

Système d'Enregistrement Robuste

- **Accusé de réception** : `RegisterAck` pour confirmer l'enregistrement
- **Retry automatique** : En cas d'échec de communication
- **États cohérents** : Synchronisation parfaite chef musiciens

Distribution Musique

- **Round-robin** : Distribution équitable des mesures
- **File d'attente** : Gestion des demandes concurrentes
- **Dés de Mozart** : Respect des tables historiques de Mozart

Tests de Validation

- **Réélection** testée et fonctionnelle
- **Départ/retour** de musiciens géré
- **Mode d'attente** avec reconnexion possible

- **Supervision strategy** testée avec crashs manuels

Le système garantit une **continuité musicale parfaite** même en cas de pannes multiples simultanées.

Explications Techniques et Défis Rencontrés

Délais de Récupération

- **Réélection après panne chef** : 3-5 secondes pour stabilisation
- **Récupération après panne musicien** : 3-5 secondes pour réorganisation

Évolution de l'Architecture

Problème initial : Nous avons d'abord implémenté uniquement l'algorithme bully, pensant que cela suffisait. Cependant, nous avons rapidement réalisé que ce n'était pas suffisant.

Défis de concurrence : Quand deux systèmes s'exécutent à des moments proches, les messages peuvent se mélanger car un système d'acteurs local fonctionne plus rapidement que l'envoi de messages. Nous pouvions nous retrouver avec **deux chefs simultanés**.

Solution adoptée : Introduction du système `ChefAnnouncement` avec battements de cœur. Ainsi, quand deux chefs coexistent, ils reçoivent les annonces, revérifient, et suivent le bon chef selon les critères d'élection.

Alternatives Considérées et Rejetées

Acteur chef dédié : Nous avons envisagé d'utiliser un système d'acteurs dédié pour le chef, mais cela aurait posé des problèmes. Si l'acteur chef crashait, la réélection ne fonctionnerait pas correctement. Nous avons donc décidé de ne pas nous limiter à un seul acteur pour la gestion de chef.

Akka Clusters : Nous avons également exploré les clusters Akka, mais nous nous sommes sentis un peu perdus sur leur utilisation et cela ne semblait pas correspondre à notre plan. Nous avons donc opté pour notre solution actuelle.

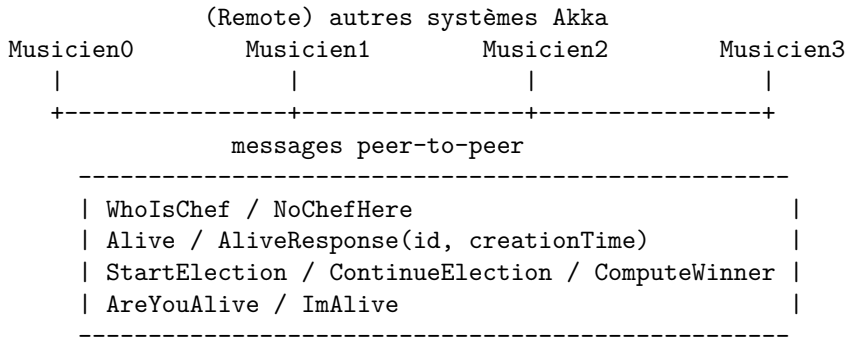
Système d'Accusé de Réception

Problème réseau : L'enregistrement pouvait échouer à cause des communications réseau. Le chef perdait parfois les messages, alors que le musicien pensait être enregistré.

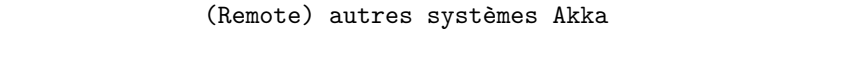
Solution : Implémentation d'un système d'accusé de réception (`RegisterAck`) pour que le musicien soit sûr d'être accepté par le chef, avec retry automatique en cas d'échec.

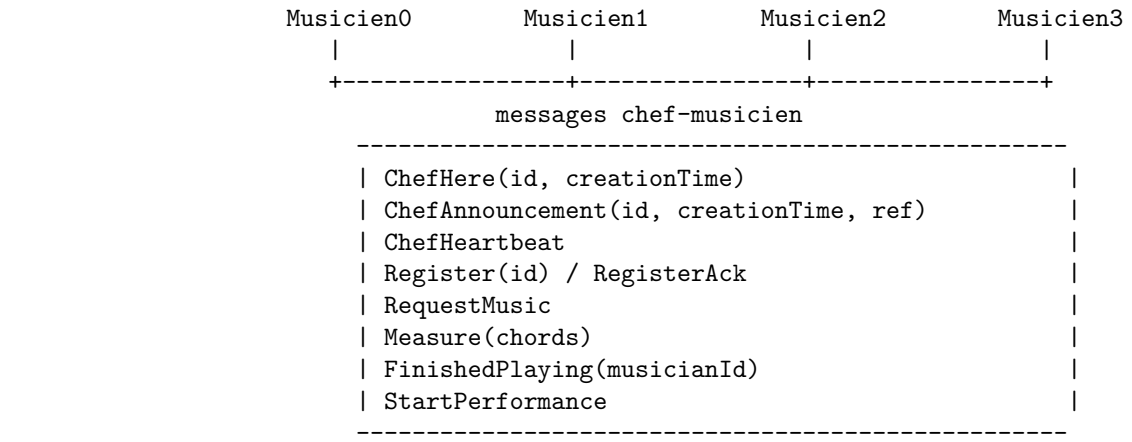
Schémas d'Architecture

1.1. Messages Musician Musician (Élection/Discovery)

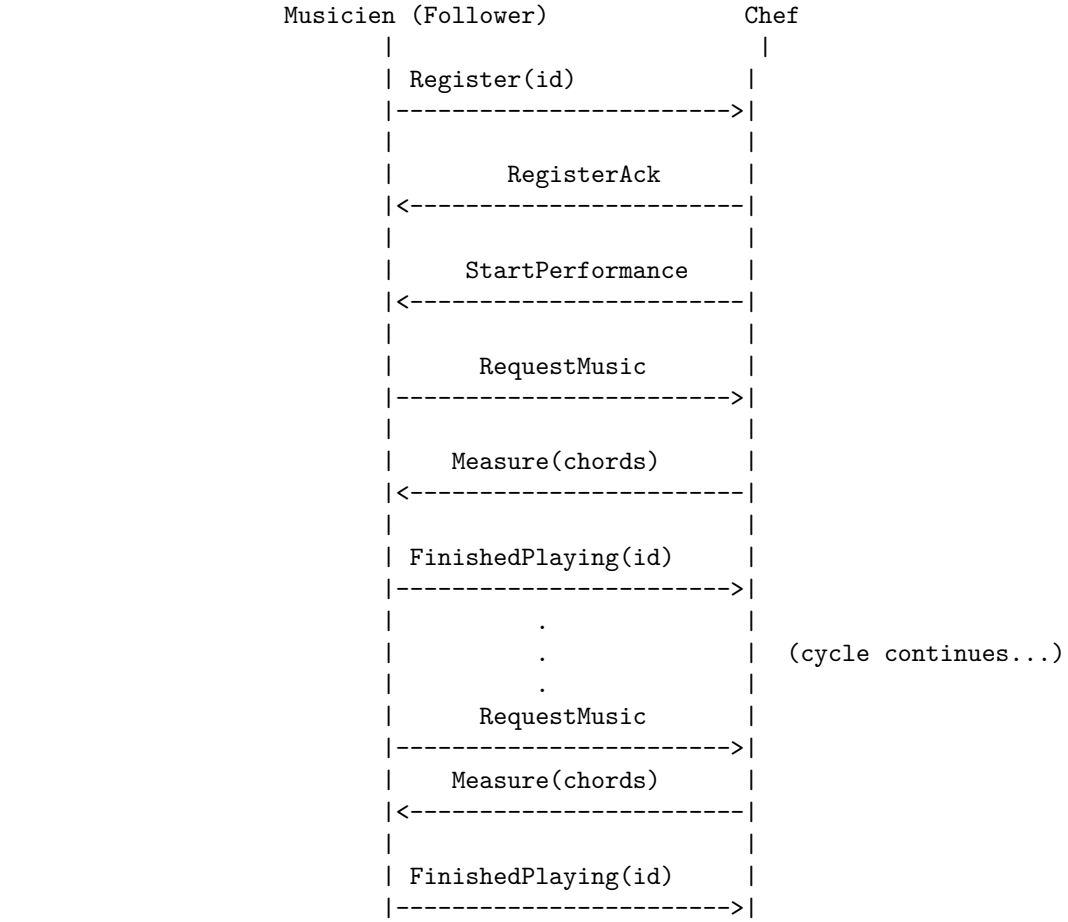


1.2. Messages Chef Musicians (Remote)

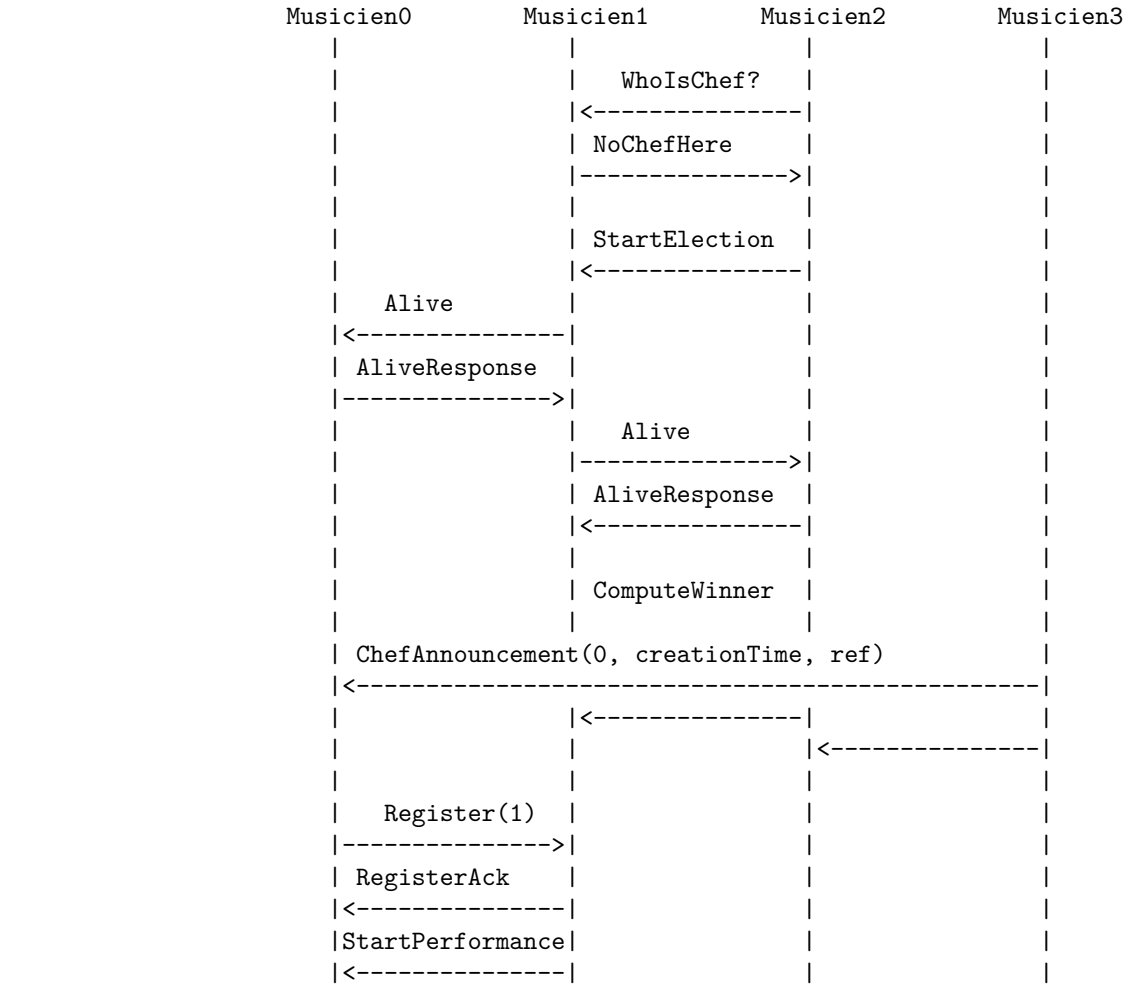




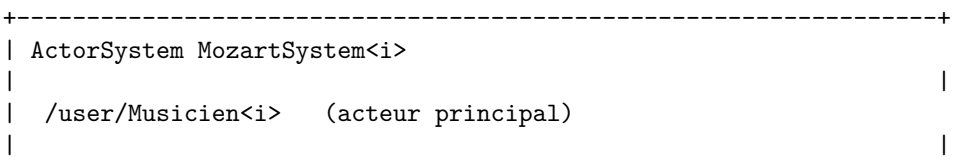
2. Messages Musicien Chef (Deroulement d'une Performance)

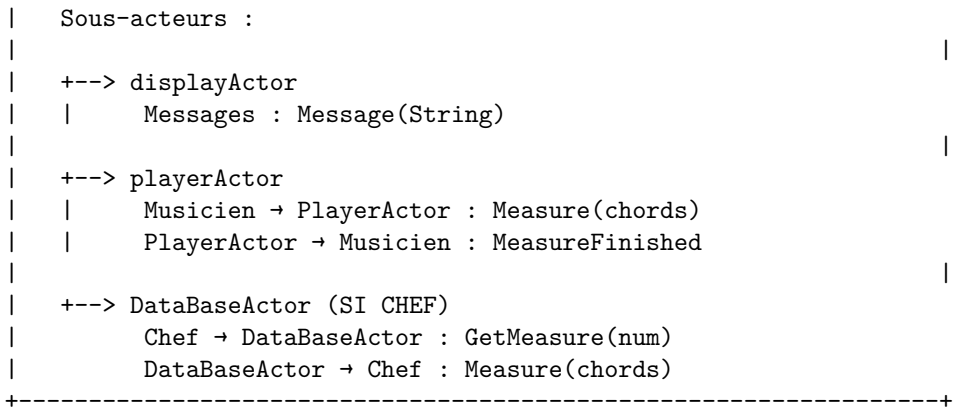


2.1. Élection du Chef (Algorithme Bully)



3. Architecture Interne d'un Node




Sous-acteurs :
+--> displayActor
Messages : Message(String)
+--> playerActor
Musicien → PlayerActor : Measure(chords)
PlayerActor → Musicien : MeasureFinished
+--> DataBaseActor (SI CHEF)
Chef → DataBaseActor : GetMeasure(num)
DataBaseActor → Chef : Measure(chords)