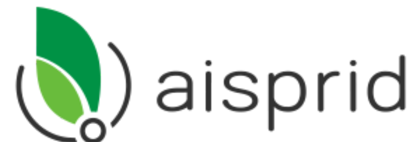
Aisprid - Ingénieur robotique : challenge technique

Le test est en deux parties :

- Une question ouverte sur les outils robotiques
- Un mini-projet d'implémentation software



Question ouverte

Le robot Aisprid a pour but d'effeuiller des plants de tomates. Pour cela, il doit naviguer dans un environnement dense et variable. Cet environnement contient différents obstacles : des obstacles fragiles (tomates), des obstacles durs (poteaux de la serre) et des obstacles occluant potentiellement les organes de perception du robot (feuilles). Le robot doit se placer avec une précision de l'ordre du millimètre, mais tout contact avec la végétation peut faire bouger le point d'intérêt et compromettre la mise en position. Le robot est composé d'une base mobile roulant sur des rails au sol et d'un bras sériel 7 DDL R-P-R-R-R-R-R (R = Rotoïde, P = Prismatique). Pour sa perception, il est muni de deux caméras, une montée sur la base et une seconde de manière distale. Actuellement, une commande articulaire avec calcul a priori de profils de vitesse en trapèze est utilisée. Les trajectoire sont définies par un faible nombre de points de passage (moins d'une dizaine). Les boucles de commande en position, vitesse, courant des moteurs sont réalisées par les variateurs des moteurs. La base de code du contrôleur est réalisée en Go et est structurée avec des serveurs gRPC, la planification de trajectoire est en python.

Sous forme libre (aucun support particulier n'est demandé, les discussions seront faites de vive voix) et succincte, proposer une ou plusieurs approches permettant de réaliser la tâche d'effeuillage en se focalisant sur la commande

et la planification de trajectoires. On pourra par exemple discuter (liste non exhaustive et non obligatoire) :

- L’architecture de commande et les outils mathématiques associés,
- Les contraintes à formuler,
- Les outils ou framework d’implémentation d’intérêts et comment ils pourraient s’interfacer avec notre architecture ou la remplacer.

La discussion doit présenter les arguments pertinents, les limitations et des détails d’implémentation pour chaque proposition.

Projet

Introduction

Ce challenge consister à implémenter le mock d’un mini contrôleur multi-moteurs simplifié.

La logique ainsi que la structuration du code seront évaluées.

Le sujet reste relativement libre tant que le core du concept est implémenté

Spécifications

Fonctionnelles

L’objectif du projet est d’implémenter **client** / **serveur(s) grpc** permettant d’effectuer des mouvements moteurs “synchronisés” multi-axes, pilotés en “vélocité”.

Un serveur principal (MotorsController) est connecté à plusieurs serveurs (Motor).

L’objectif du serveur principal est de synchroniser les moteurs en vitesse afin d’atteindre des angles moteurs désirés au même instant.

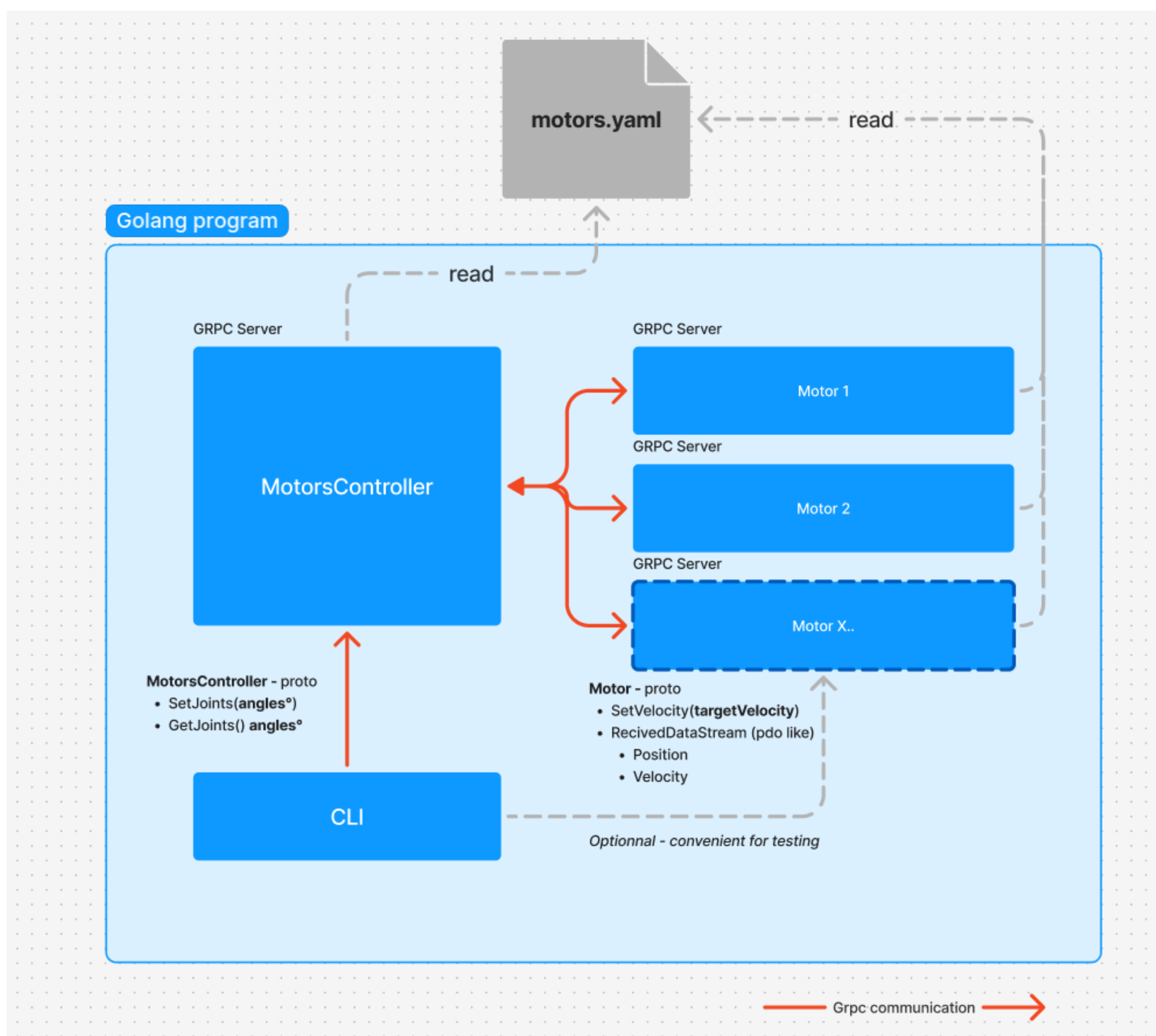
Les moteurs doivent virtualiser un “mouvement” en vélocité basé sur leur consigne reçue.

Un modèle simplifié doit être utilisé avec:

- **nouvelle_position** = **ancienne_position** + **consigne_vitesse** x **delta_temps**.

La configuration des moteurs est transmise aux différents serveurs via un fichier yaml central.

Les commandes sont initiées via une **CLI** effectuant des appels GRPc au serveur **MotorsController.**



Contraintes

- Bien utiliser les **go-routines** ainsi que les **channels**
- Les moteurs n'ont pas tous la même vélocité max
- Gérer quelques cas limites (exemple : le moteur dépasse sa limite en position)
- Les protos sont fournis (mais peuvent être modifiés/étendus au besoin)
- Un seul binaire doit être compilé
- **Les moteurs ont une accélération infinie**. Les profils de vitesses sont donc de valeur constante pour une trajectoire donnée. Les trajectoires doivent être générées à vitesse maximale mais limitée par le moteur le plus lent pour assurer la synchronisation.
- L'évolution de la position des moteurs doit être affichée sur la console des serveurs en sortie standard via **logrus**

Exemples & boostrap

motorsim.proto

```
syntax = "proto3";

import "google/protobuf/empty.proto";
option go_package = ".;motorprotos";

package motorprotos;

// Server converting target angles to velocity profile for mo
service MotorsController {
  rpc SetJoints(Angles) returns (Angles) {}
  rpc GetJoints(google.protobuf.Empty) returns (Angles) {}
}

// Server converting target angles to velocity profile for mo
service Motor {
  rpc SetVolicty(Velocity) returns (google.protobuf.Empty) {}
  rpc ReceiveDataStream(google.protobuf.Empty) returns (strea
}
```

```
message Angles {
  repeated double angles = 1;
}

message Velocity {
  double velocity = 1;
}

message MotorData {
  double angle = 1;
  double velocity = 2;
  string error = 3; // Will be filled if motor encounter an e
}
```

motors.yaml

CECI EST UN EXEMPLE ! non testé - peut être modifié au besoin

```
motors:
    - id: 1
      port: 8081
      min_pos: -250 # degrees
      max_pos: 250 # degrees
    max_vel: 100 # degrees/second
    accel: 200 #### BONUS, pas obligatoire
    - id: 2
      port: 8082
      min_pos: -100 # degrees
      max_pos: 100 # degrees
      max_vel: 200 # degrees/s
    - id: 3
      port: 8083
      min_pos: -500 # degrees
```

```
        max_pos: 500 # degrees
        max_vel: 500 # degrees/s
```

Commandes attendues

```
## Lancer les serveurs sur différents terminaux
./motorsim -c motors.yaml controller serve -p 8080
# > MotorscController listening on port :8080
./motorsim -c motors.yaml motor serve --id 1
./motorsim -c motors.yaml motor serve --id 2
./motorsim -c motors.yaml motor serve --id 3

## Récupérer la position des moteurs
./motorsim -c motors.yaml controller get_joints
# > angles: [81°, 50°, 61.1°]
./motorsim -c motors.yaml controller set_joints -- 80 -50 23
# > Motion in progres ...
# > SUCCESS
./motorsim -c motors.yaml controller set_joints -- 300 -50 23
# > ERROR: out of limits

###### BONUS - optionnel - (peut faciliter le debugging)
## Commandes vers moteur depuis CLi en direct
./motorsim -c motors.yaml motor --id 3 move_vel -- 50 # lance
./motorsim -c motors.yaml motor --id 3 listen # Se branche au
```

Rendu

- Le temps du challenge devrait être de **maximum 8h** (hors tuto Golang 😉)
- Tous les packages peuvent être utilisés (standard ou autre)
  - Pour la CLI https://github.com/spf13/cobra
  - Pour le logger https://github.com/sirupsen/logrus
- Le rendu se fera sous forme d'un repo git (URL) ou bien d'une archive **motorsim.tar.gz**

- Les sources
- Un **Makefile**
  - **build** - generate the binary as **motorsim**
  - **generate-protos** - compile *.proto into go source files
  - (d'autre commandes si besoin)

Ressources

- https://go.dev/doc/install
  - https://go.dev/tour/welcome/1 (tuto go basique)
  - https://go.dev/tour/concurrency/1 (la section sur la **concurrency** idomatique de Go)
- https://grpc.io/docs/languages/go/quickstart/ - convert proto file to .go file
- https://stackoverflow.com/questions/51775247/parse-yaml-with-structs
- https://github.com/spf13/cobra - powerful CLI
- VScode golang plugin → recommandé