# ROS control et son utilisation pour les Dynamixels

dorian.goepp@inria.fr

des Dynamixels pour nos robots

# Le projet ResiBots

Objectif : développer des algorithmes d'apprentissage par essai-erreur

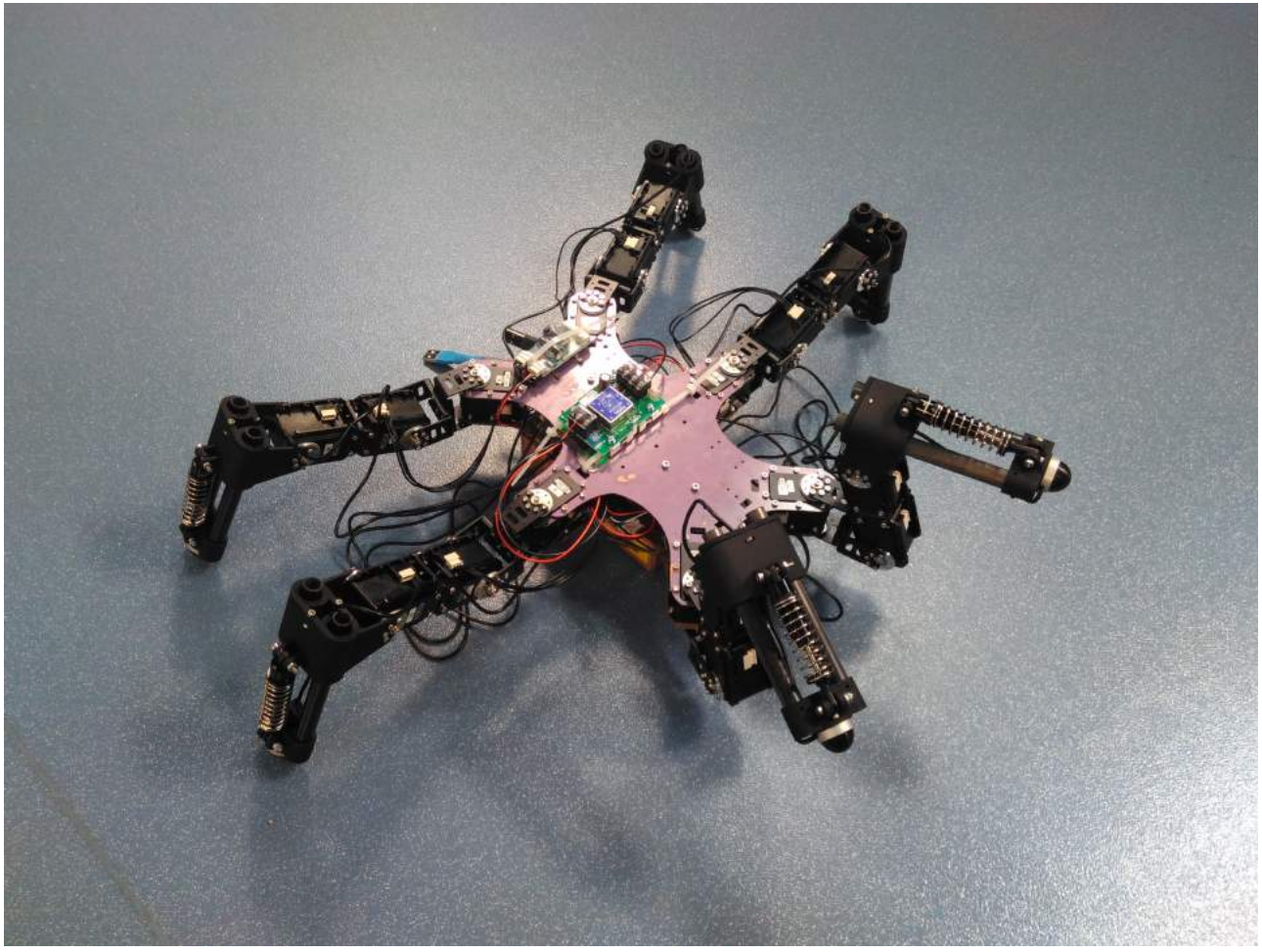
- l'autonomie des robots sur le long terme,
- en gérant des dommages imprévus
- avec des robots abordables

Mots-clefs :

- a-priori issus de simulations
- optimisation bayésienne
- processus gaussiens

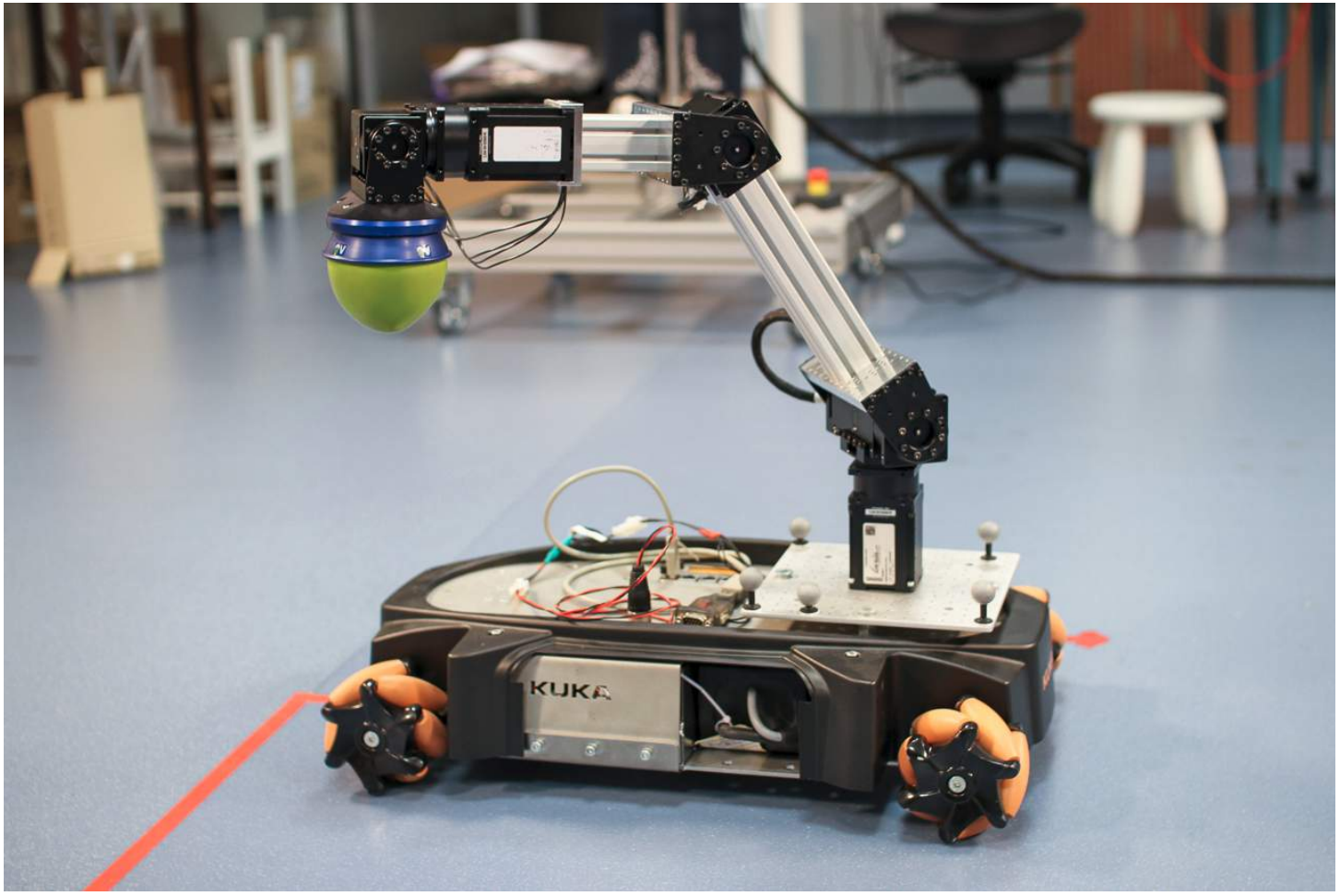
# Robots pour ResiBots

robots à pattes,



# Robots pour ResiBots

robots à pattes, un bras manipulateur et une base mobile Kuka.



# Robots pour ResiBots

robots à pattes, un bras manipulateur et une base mobile Kuka.

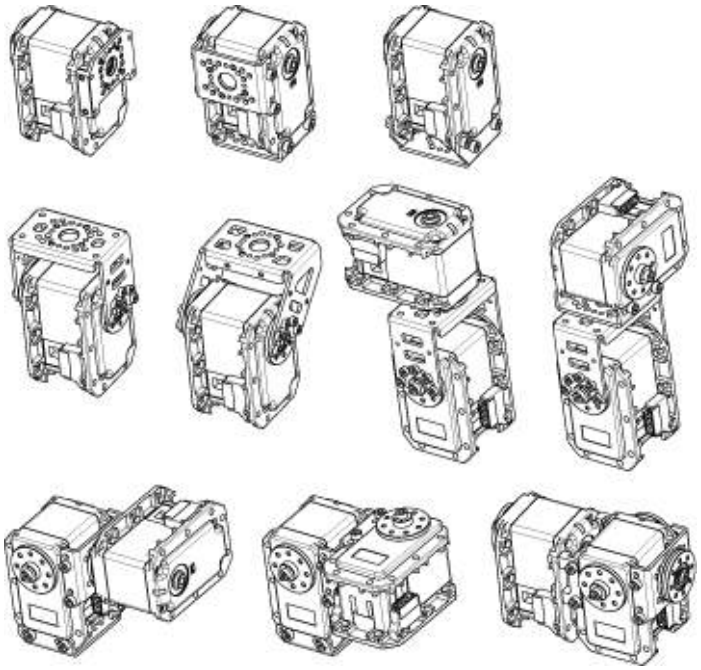
# Dynamixels

# Dynamixels


- toute l'électronique est intégrée
- branchement en cascade -> une seule interface série/USB et une seule alimentation
- informations disponibles sur l'état du servo (position, vitesse, température, etc.)
- le protocole de communication est (presque) complètement documenté
- protégés contre les surcharges et surtensions
- les actionneurs sont *relativement* abordables
- plusieurs gammes

# Dynamixels

- permettent un assemblage facile et modulaire



# Robots tiers basés sur les Dynamixels

Poppy ...

Poppy, le robot humanoïde de l'INRIA Bordeaux

# Robots tiers basés sur les Dynamixels

Poppy et bien sûr TurtleBot 3

Poppy, le robot humanoïde de l'INRIA Bordeaux

# Donc...

- nous travaillons avec ROS
- nos robots sont à base de dynamixels

Deux approches d'intégration à ROS:

- ad-hoc
- avec ros_control

# ros_control

fortement inspiré de la présentation de Adolfo Rodríguez Tsouroukdissian

# `ros_control`

C'est

- un ensemble de paquets ROS
- un petit framework/cadriciel pour l'écriture de contrôleurs de robots

Il va nous permettre de

- distinguer le code chargé de
  - l'*interaction avec le matériel*, de celui chargé de
  - la *boucle d'asservissement*
- réutiliser le code des autres
- accéder aux outils comme MoveIt! et navigation stack
- asservir en temps réel

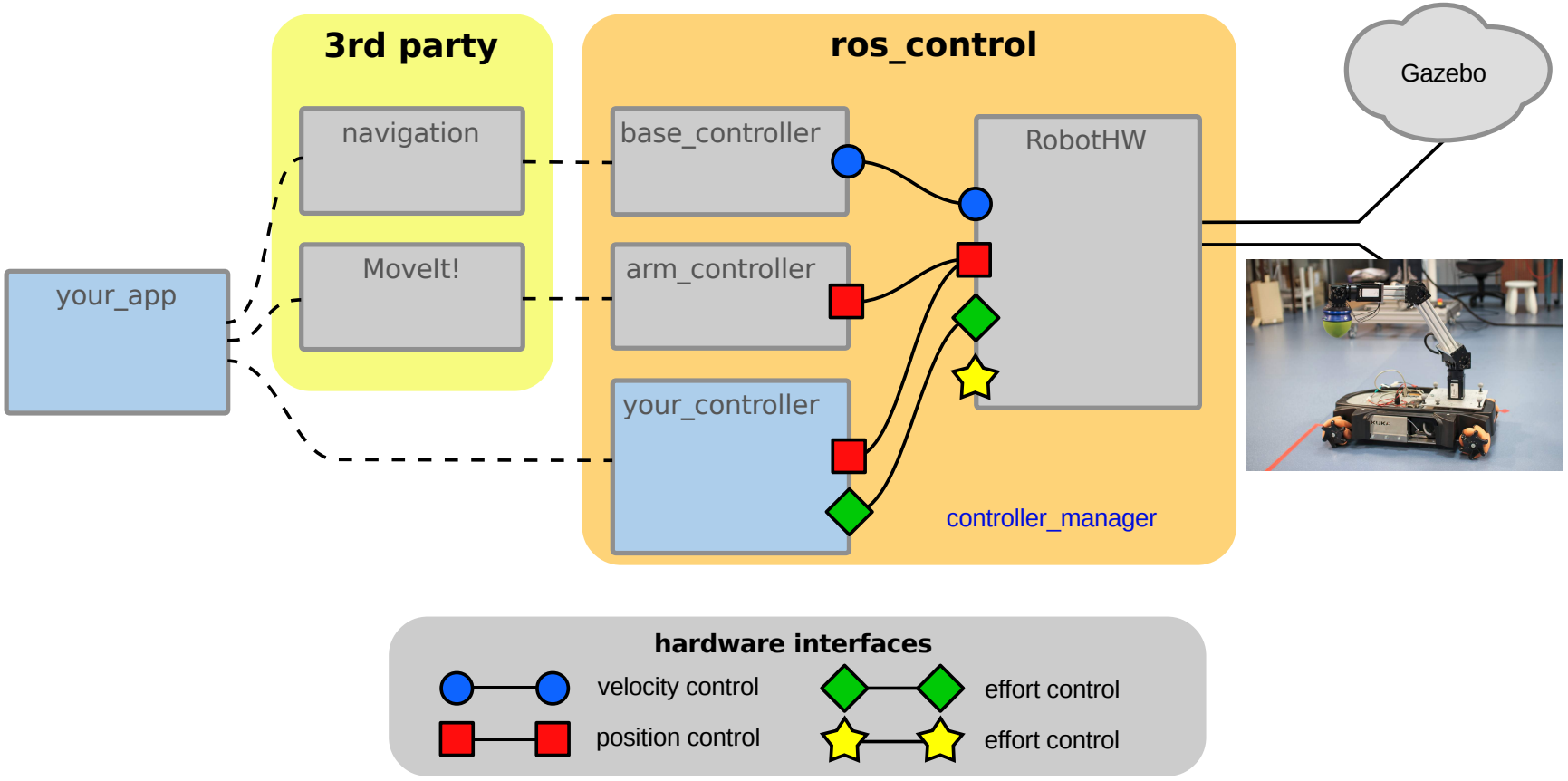
Trois modes de commande : position, vitesse et effort

# ros_control

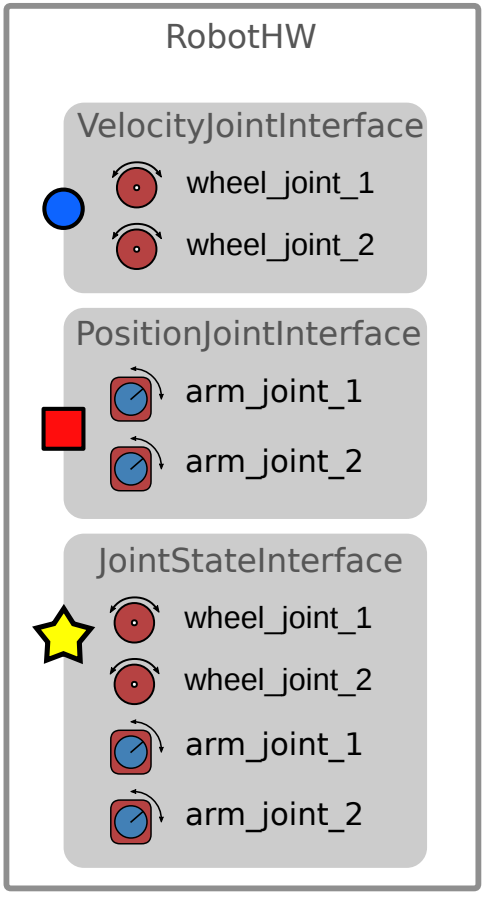
Historique

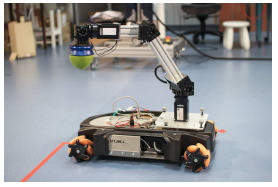
- pr2_controller_manager créé en 2009 par Willow Garage
- ros_control démarré fin 2012

# ros_control

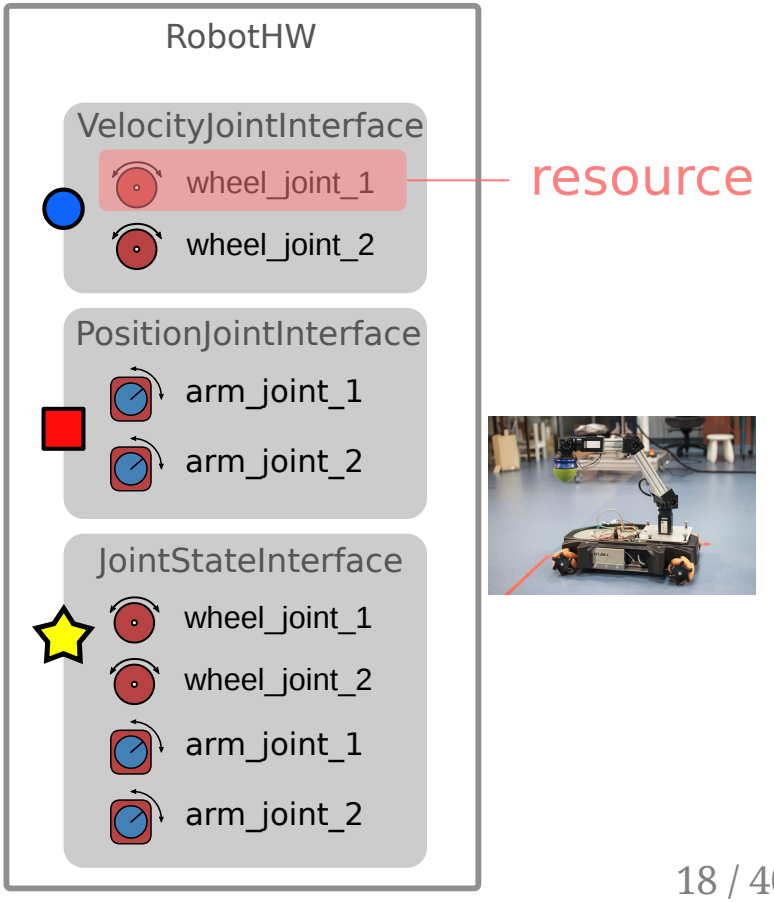

3rd party
navigation
MoveIt!
your_app
ros_control
base_controller
arm_controller
your_controller
RobotHW
controller_manager
Gazebo
hardware interfaces
velocity control
position control
effort control
effort control

# ros_control

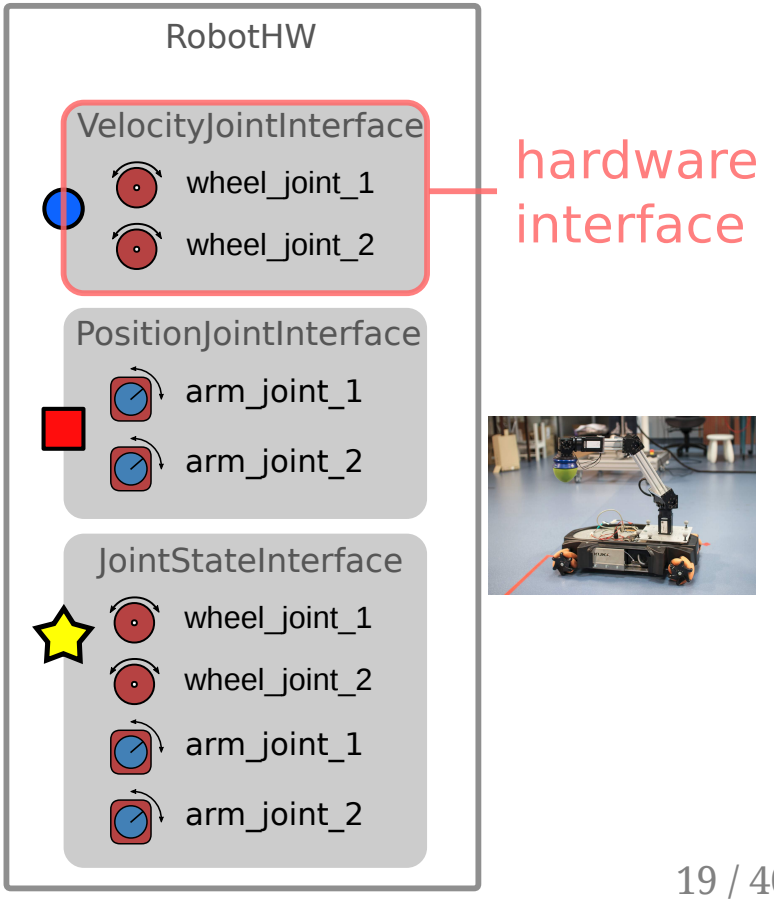

RobotHW
VelocityJointInterface
wheel_joint_1
wheel_joint_2
PositionJointInterface
arm_joint_1
arm_joint_2
JointStateInterface
wheel_joint_1
wheel_joint_2
arm_joint_1
arm_joint_2



# ros_control


RobotHW
VelocityJointInterface
wheel_joint_1
wheel_joint_2
resource
PositionJointInterface
arm_joint_1
arm_joint_2
JointStateInterface
wheel_joint_1
wheel_joint_2
arm_joint_1
arm_joint_2

# ros_control

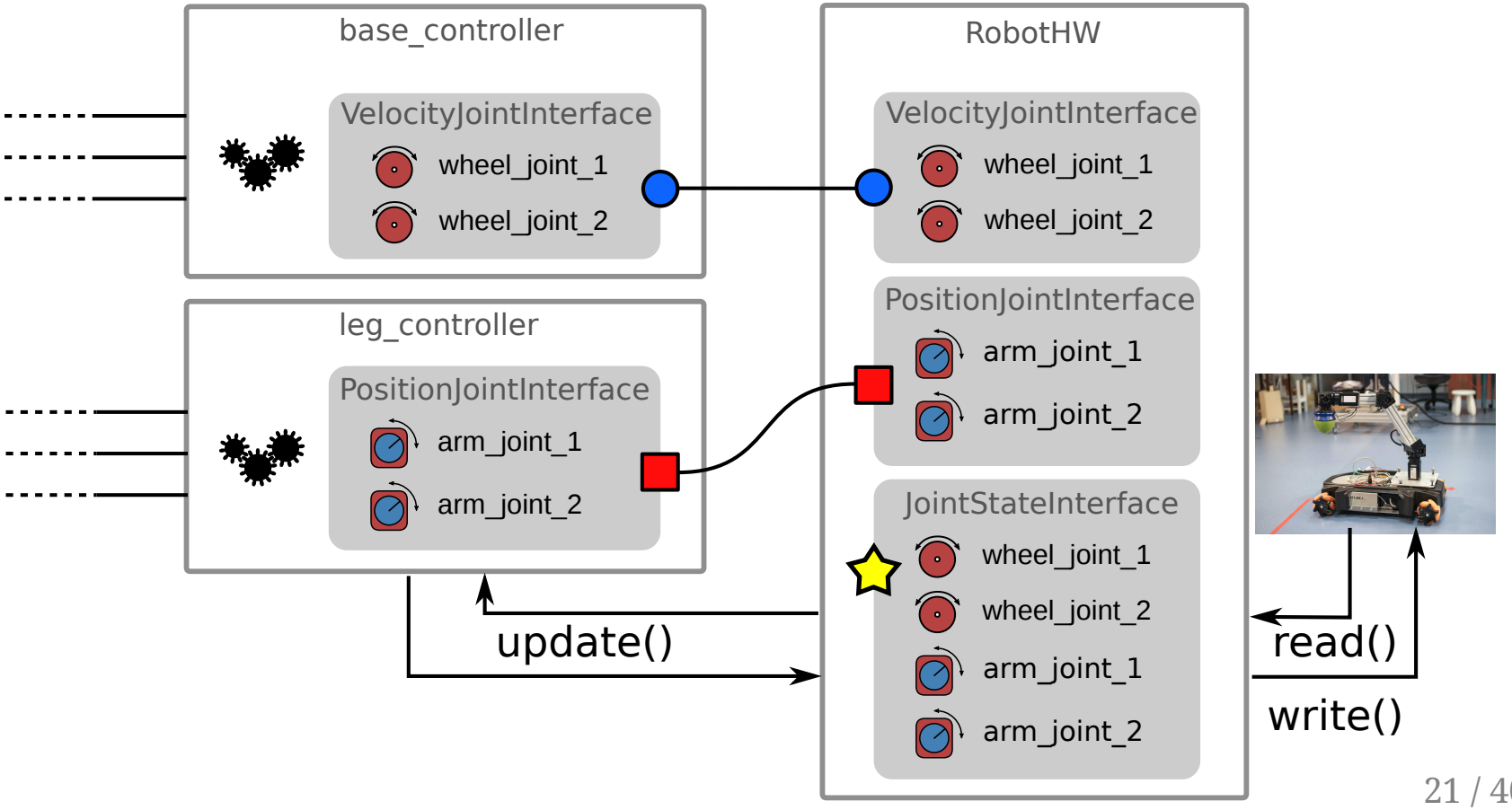

RobotHW
VelocityJointInterface
wheel_joint_1
wheel_joint_2
PositionJointInterface
arm_joint_1
arm_joint_2
JointStateInterface
wheel_joint_1
wheel_joint_2
arm_joint_1
arm_joint_2
hardware interface

# ros_control

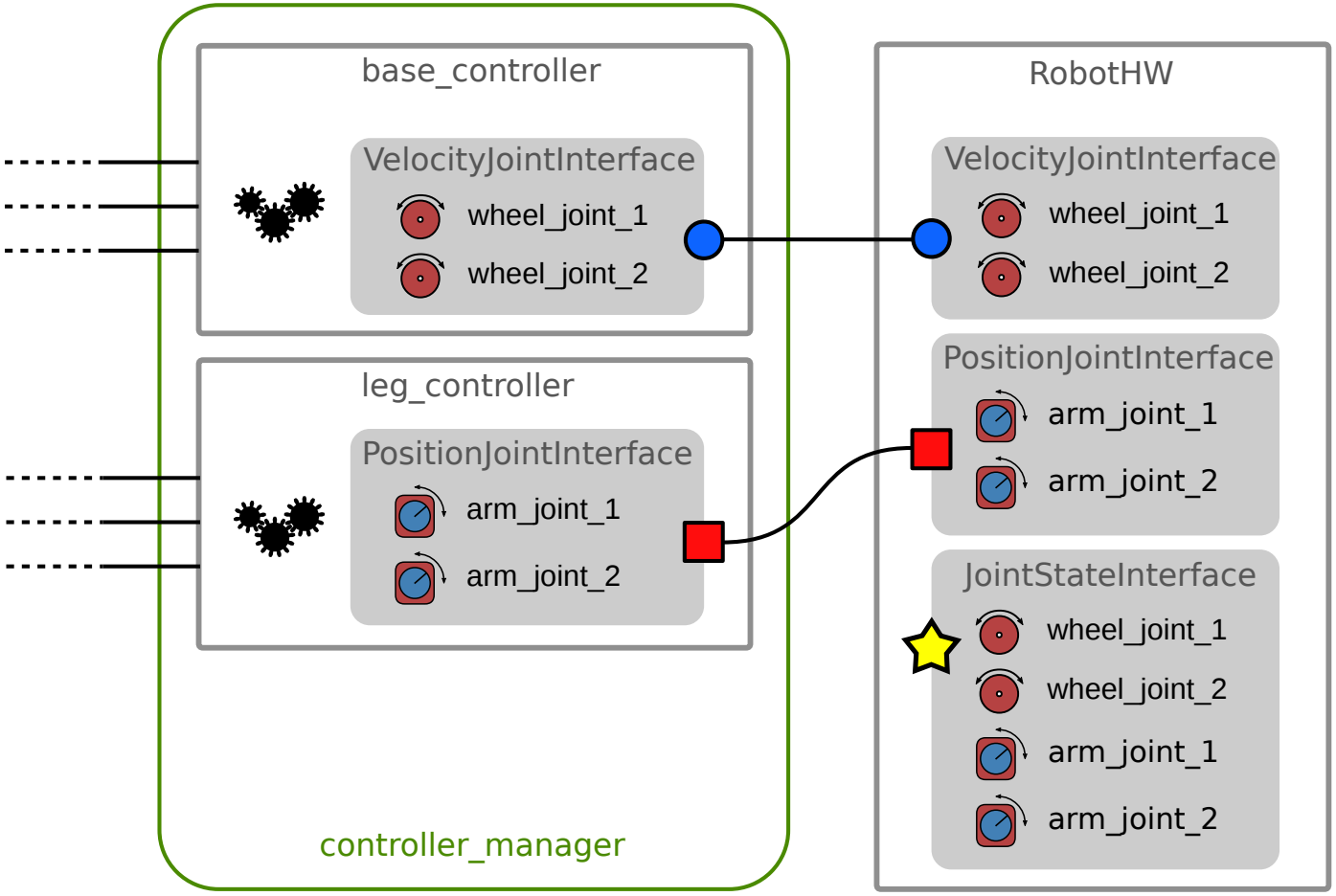
## RobotHW

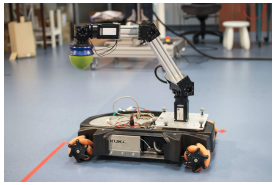
```
class DynamixelHardwareInterface : public hardware_interface::RobotHW {
    public:
        DynamixelHardwareInterface(const std::string& usb_serial_interface,
            const int& baudrate,
            const float& read_timeout,
            const float& scan_timeout,
            std::map<long long int, std::string> dynamixel_map,
            std::map<long long int, long long int> dynamixel_max_speed,
            std::map<long long int, double> dynamixel_corrections);

        ~DynamixelHardwareInterface();

        /// Find all connected devices and register those refered in dynamixel_map in the
        /// hardware interface.
        void init();

        // get the hardware's state
        void read_joints();

        // send the new command to hardware
        void write_joints();
};
```

# ros_control


base_controller
VelocityJointInterface
wheel_joint_1
wheel_joint_2
leg_controller
PositionJointInterface
arm_joint_1
arm_joint_2
RobotHW
VelocityJointInterface
wheel_joint_1
wheel_joint_2
PositionJointInterface
arm_joint_1
arm_joint_2
JointStateInterface
wheel_joint_1
wheel_joint_2
arm_joint_1
arm_joint_2
update()
read()
write()

# Contrôleurs



# Contrôleurs

Plugin chargé dynamiquement par le `controler_manager`.

Un contrôleur peut être dans deux états :

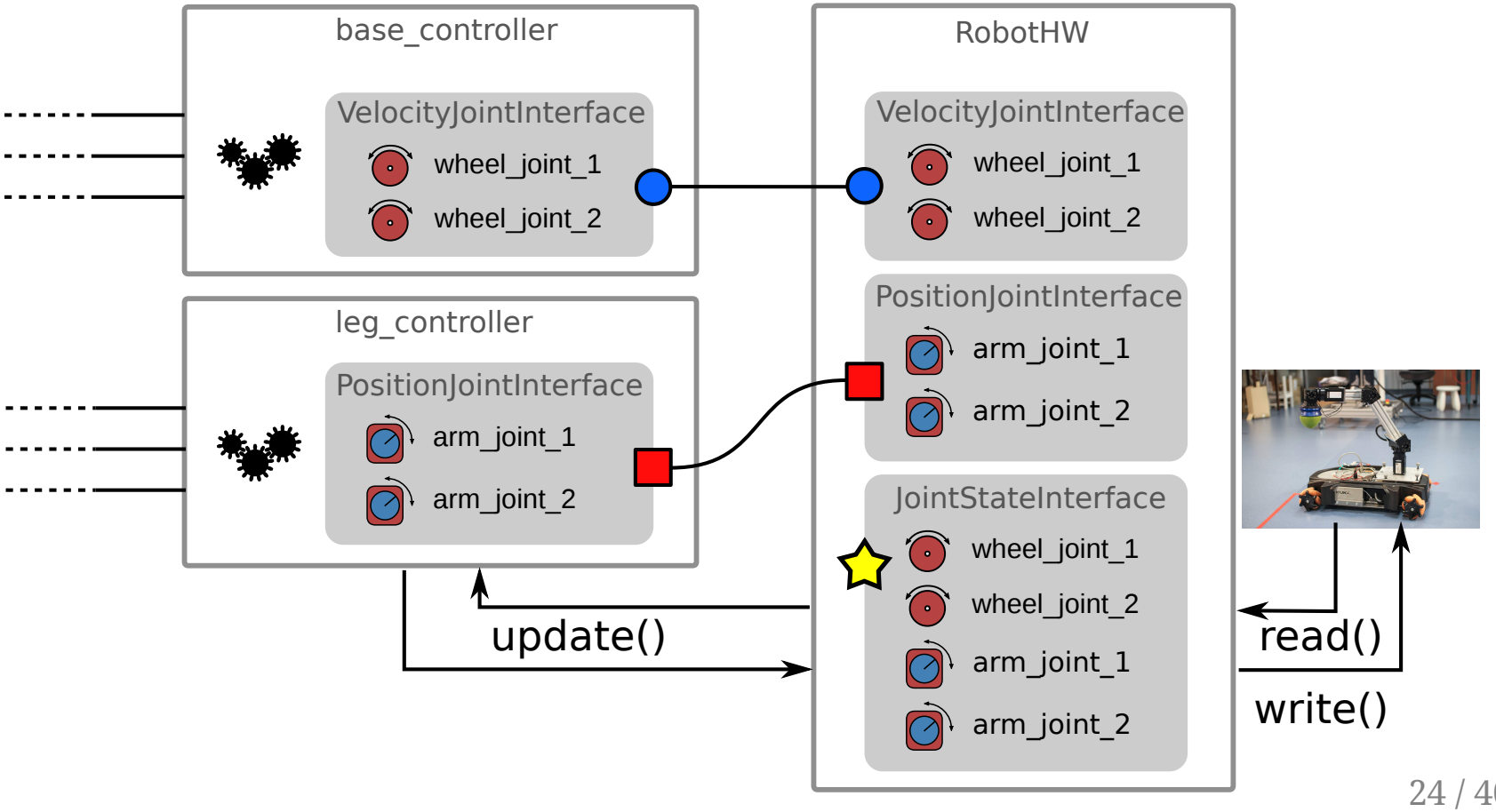
- `stopped`
- `running`

via des services ROS, `controler_manager`

- charge les contrôleurs
- gère leur cycle de vie

Les contrôleurs sont exécutés **périodiquements** et **séquentiellement**

# Circulation des données



# Circulation des données

```
void DynamixelLoop::update(const ros::TimerEvent& e)
{
    // Input
    // get the hardware's state
    _hardware_interface->read_joints();

    // Control
    // let the controller(s) compute the new command (via the controller manager)
    _controller_manager->update(ros::Time::now(), _elapsed_time);

    // Output
    // send the new command to hardware
    _hardware_interface->write_joints();
}
```

# dynamixel hardware interface

https://github.com/resibots/dynamixel_control_h

# `libdynamixel`*

Nous avons développé la bibliothèque `libdynamixel`*. Caractéristiques principales :

- bibliothèque open source
- écrite en C++11
- fonctionne avec les version 1 et 2 du protocole
- couvre (presque) tous les actionneurs
- offre deux approches :
  - template : bas niveau
  - classes : haut niveau, masque les différences entre les modèles d'actionneurs

*https://github.com/resibots/libdynamixel

# libdynamixel*

```
dynamixel::StatusPacket<dynamixel::protocols::Protocol1> status;

dynamixel::controllers::Usb2Dynamixel serial(_usb_serial_interface, _baudrate);

// goal position in ticks (0-2048)
serial.send(dynamixel::servos::Mx28::set_goal_position_angle(2, goal_position));

serial.recv(status);
```

*https://github.com/resibots/libdynamixel

# libdynamixel*

```
dynamixel::StatusPacket<dynamixel::protocols::Protocol1> status;

dynamixel::controllers::Usb2Dynamixel serial(_usb_serial_interface, _baudrate);

auto _dynamixel_servos = dynamixel::auto_detect<dynamixel::protocols::Protocol1>(serial);

// goal position in radians (0-2π)
serial.send(
    _dynamixel_servos[2]->
        set_goal_position_angle(goal_position));

serial.recv(status);
```

*https://github.com/resibots/libdynamixel

# ros_control

Pour nous :

- abstrait la question du modèle de Dynamixel (pour un même protocole)
- suivre l'évolution les robots (nb de servos, nouveaux modèles) avec un minimum de changements logiciels.

En somme, moins de code écrit en dur, plus de flexibilité.

Cependant, les Dynamixels disposent déjà d'un asservissement intégré (type PID).

# `dynamixel_control_hw`

Ce noeud utilise

- `ros_control_boilerplate`*
- la bibliothèque `libdynamixel` développée dans l'équipe

Fonctionnalités :

- nombre arbitraire d'actionneurs (fichier de configuration)
- commande en radians (pas de conversion)
- décalage et vitesse limite pour chaque actionneur

Charactéristiques :

- fréquence ≥ 50 Hz pour 18 servo-moteurs Dynamixel

*https://github.com/davetcoleman/ros_control_boilerplate/

# Exemple de fichier de configuration

```yaml
dynamixel_control_hw:
  loop_frequency: 50
  cycle_time_error_threshold: 0.1
  serial_interface: /dev/ttyACM0
  baudrate: 1000000 # in bauds
  dynamixel_timeout: 0.05 # in seconds
  dynamixel_scanning: 0.05 # in seconds
  # correspondance between hardware IDs of the actuators and their names in ROS
  hardware_mapping:
    arm_joint_1: 2
    arm_joint_2: 3
  # offset to be added, in radians, to the position of an actuator
  hardware_corrections:
    "2": 3.14159265359
    "3": 3.14159265359
```

# Exemple de fichier de configuration

```yaml
dynamixel_controllers:
  joint_state_controller:
    type: joint_state_controller/JointStateController
    publish_rate: 50

  first_traj_controller:
    type: position_controllers/JointTrajectoryController
    joints:
      - arm_joint_1
      - arm_joint_2
```

# Exemple de roslaunch

```xml
<?xml version="1.0"?>

<launch>

  <!-- URDF robot description -->
  <param name="robot_description" command="cat $(find dynamixel_control_hw)/urdf/sample.urdf" />

  <!-- Publish robot's state in the tf tree -->
  <node pkg="robot_state_publisher" type="robot_state_publisher" name="rob_state_pub"/>

  <!-- Parameters for the hardware interface and controllers -->
  <rosparam file="$(find dynamixel_control_hw)/config/traj.yaml"/>

  <!-- launch our hardware interface -->
  <node pkg="dynamixel_control_hw" type="dynamixel_control_hw" name="dynamixel_control_hw"
       output="screen"/>

  <!-- Ask controller_manager to load a controller for our dummy robot -->
  <node name="controller" pkg="controller_manager" type="spawner" respawn="false"
    output="screen" args="/dynamixel_controllers/joint_state_controller
      /dynamixel_controllers/first_traj_controller" />
</launch>
```

# Démo avec deux actionneurs

# Conclusion

# Pour résumer

`ros_control` c'est cool, mais la documentation lacunaire.

L'écriture d'une interface matérielle est probablement aussi complexe que celle d'un noeud ad-hoc, par manque de documentation.

# Et au delà

- ros_control peut faire plus encore…
  - intégration avec gazebo
  - transmission interface
  - limites d'articulation
  - interface matérielle combinée
  - politique de gestion des ressources
  - ros_controllers

- ros_control devrait dans le futur pouvoir faire…
  - gérer le changement de mode de commande

- future work
  - la commande en vitesse (wheel mode)
  - limites angulaires
  - protocole 2

# Et au delà

- ros_control peut faire plus encore…
  - intégration avec gazebo
  - transmission interface
  - limites d'articulation
  - interface matérielle combinée
  - politique de gestion des ressources
  - ros_controllers

- ros_control devrait dans le futur pouvoir faire…
  - gérer le changement de mode de commande

- future work
  - la commande en vitesse (wheel mode)
  - limites angulaires
  - protocole 2

dorian.goepp@inria.fr