Vers une gestion fine de la synchronisation dans les middleware robotiques ?

Journées Technologiques Robotex

21, 22 Juin 2017

R. Passama

LIRMM

Contexte : projet BAZAR

- Robot assistant industriel polyvalent
  - Collaboration homme-robot
  - Interaction multi-sensorielle (effort et/ou vision et/ou laser et/ou toucher)
  - Déplacements dans l'environnement
  - Commande mono et bi-manuelle

- Equipe (IDH)
  - Chercheurs : A. Cherubini, P. Fraisse, A. Crosnier
  - Doctorants : B. Navarro, M. Sorour, Z. Abderrahmane, J. Zhu, O. Mazhar, S. Tarbouriech
  - Ingénieurs : M. Haijoubi
  - Extérieurs: A. Khelloufi (CDTA Alger), V. Bonnet (Univ. Paris-Est)

Contexte : projet BAZAR

- Un exemple

A Framework for intuitive collaboration with a mobile manipulator

B. Navarro A. Cherubini A. Fonte G. Poisson P. Fraisse

PRISME - University of Orléans INSA CVL
Orélans, France
LIRMM - University of Montpellier CNRS
Montpellier, France

Rappel : intégration logicielle

- Contrainte : variabilité matérielle
  - Beaucoup de matériel différents (capteurs, actionneurs, moyen de communication) à gérer au niveau logiciel.
  - Architecture « flexible » : facilité pour ajouter / enlever des dispositifs ; facilité d’adaptation à une plate-forme réelle et à un simulateur.
- Contrainte : variabilité logicielle
  - Nombreuses fonctionnalités, complémentaires et/ou partiellement redondantes
- Contrainte : variabilité humaine
  - Plusieurs personnes avec des compétences différentes travaillant sur les mêmes codes

Rappel : intégration logicielle

- Besoin : gérer le cycle de vie des projets
  - Gérer les contributions des personnes aux différents sujets
  - Assurer la pérénité des projets malgré un turn over important

→ **Formaliser les projets (qui, quoi, où, comment)**

- Besoin : Produire des codes interopérables
  - « Blocs » réutilisables
  - Codes pouvant se substituer les uns aux autres suivant les besoins (gestion de la variabilité matérielle et logicielle)

→ **Définir des contraintes (restrictions)**
quant à l'interface et à l'utilisation des blocs fonctionnels

Gestion du cyle de vie : PID

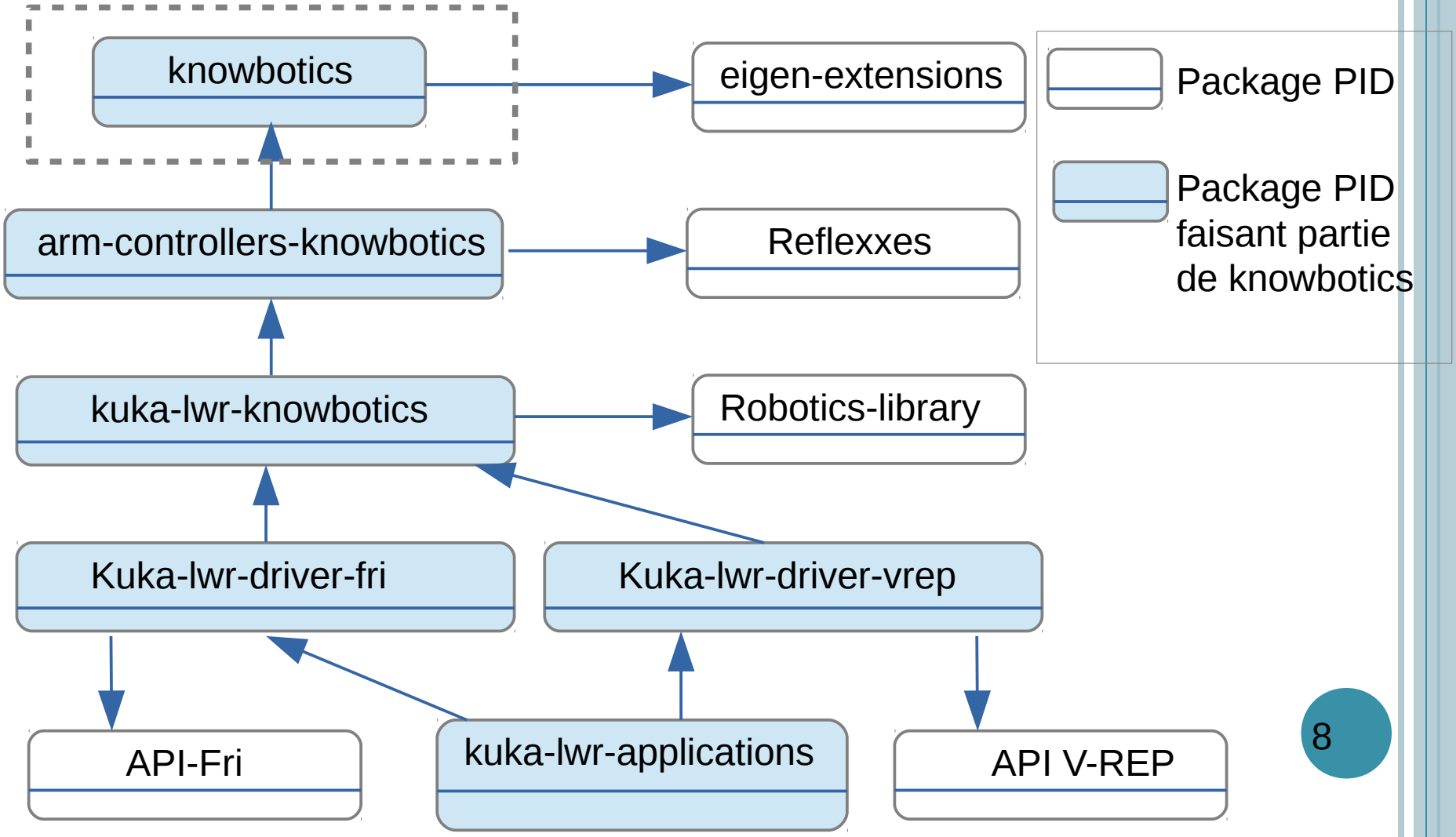
- PID : package integration and deployment
  - Une méthodologie de développement et un framework logiciel basé sur CMake et git (+gitlab côté serveur).
- Idées directrices
  - Développer des **petits projets interdépendants**
    - Limiter le travail en parallèle sur le même code.
    - Favoriser une **stabilisation** rapide des interfaces
  - **Formaliser** la description des projets
    - Normalisation de la structure du projet
    - Description détaillée du contenu et des contraintes
  - **Automatiser** au maximum le processus de développement
    - Vérification (erreurs de description, compatibilité de versions)
    - Simplification : déploiement, construction, test, génération de documentation, gestion des licences, publication.

Interopérabilité : knowbotics

- knowbotics
  - Un framework logiciel (C++) extensible pour la standardisation et l’intégration de codes robotiques.
  - collection de packages PID
- Idées directrices
  - **Suivre une convention** pour combiner les algorithmes
  - **Définir précisémment les contraintes** afin de bien décrire le contexte d’utilisation
  - **Rendre lisible les codes**
    - « parler dans **la bonne langue** » (utiliser des noms et des concepts qui « veulent dire quelque chose »)
    - Imposer **quelques règles** (de nommage, de structure) aux développeurs.

Interopérabilité : knowbotics

- Exemple de structure de projet



Interopérabilité : knowbotics

- Exemple : Description de l'architecture

objets dans l'environnement (robots, capteurs, objets d'intéret, etc.) : **définit les types**

```
auto kuka = World::add_To_Environment("kukaLWR",new KukaLWRArm());
```

processus participant au comportement (algorithmes, drivers) : **contraint par les types**

```
auto driver = World::add_To_Behavior("driver",
                 new KukaLWRFRIDriverProcessor( kuka, ...));
```

Utilisable uniquement avec un kuka LWR

```
auto cartesian_controller = World::add_To_Behavior( "CartesianController",
                 new CartesianController<7> (kuka,...));
```

Utilisable uniquement avec un bras série (donc un kuka LWR)

```
auto current_twist_estimator = World::add_To_Behavior(
                 "current_twist_estimator",
                 new TwistEstimator<TwistEstimatorMethod::Kalman>(kuka,...);
```

Utilisable avec tous les objets physiques (donc un kuka LWR)

...

Interopérabilité : knowbotics

- Exemple : pipeline exécutée périodiquement

```
driver->get_Input_Values(true);    // Get the new joint positions
kinematics();                      // Compute forward kinematic and jacobian
current_twist_estimator();         // Estimate the twist at end effector

cartesian_controller();            // Compute the new joint target positions
driver->set_Output_Values();       // Apply the new commands
```

Les algorithmes de commande sont créés par enchainement de ces fonctionnalités « simples »

Synchronisation « à la main » dans le code

Découpage de l'application en fonctionnalités robotiques se fait à granularité fine

PID + knowbotics - réflexions

- Qu’est ce qu’une « granularité fine » ?
  - une vue de l’esprit… qui dépend des habitudes/pratiques de programmation
- Avantage du découpage à grain fin
  - Plus facile de faire des codes **générique** et **réutilisables** s’ils sont limités à un aspect particulier
- Inconvénient du découpage à grain fin
  - Par définition beaucoup plus de « petits bouts » à « recoller ensembles » => plus grande complexité à gérer au niveau architectural.

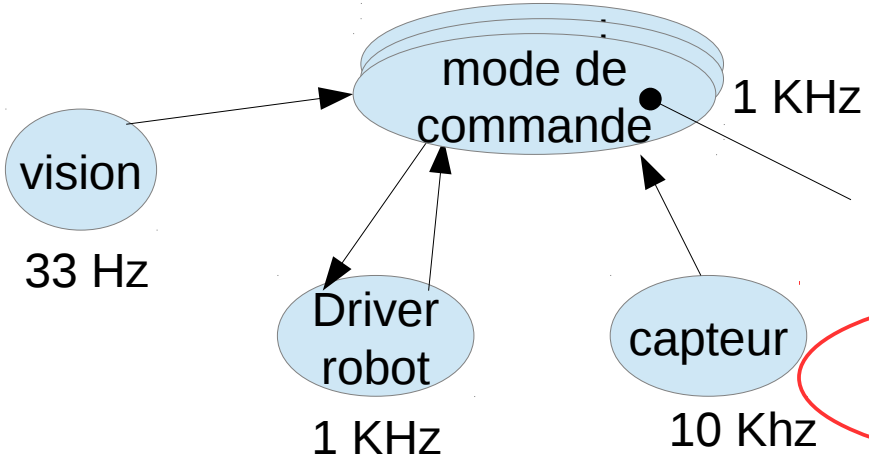
PID + knowbotics - réflexions

- Implémentation à grain fin des fonctionnalitiés robotiques
  - Pas l’apanage de PID + knowbotics
  - Faisable théoriquement dans n’importe quel langage et/ou framework/middleware robotique
  - knowbotics : couche de « standardisation » qui peut être remplacée par une norme de programmation/ de nommage stricte

Comment exploiter ces fonctionnalités robotiques dans une architecture de contrôle complexe ?

Exploiter les fonctionnalités

- Caricature d'architecture ROS (*mais pas tant que ça...*)



Travail : implémenter les modes de commande

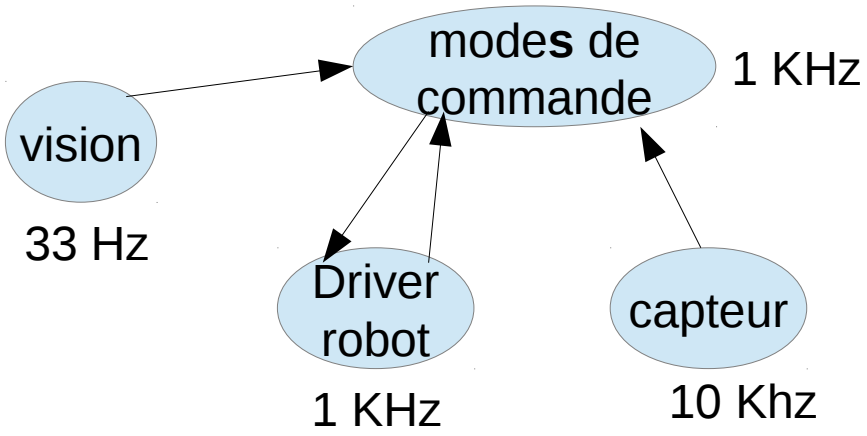
Example :

```
kinematics();
current_twist_estimator();
cartesian_controller();
```

- Remarques
  - Ce qui nous intéresse : **réutiliser les fonctionnalités utilisées pour implémenter les modes**
  - Ce qui nous embête : **dupliquer la même fonctionnalité dans la même architecture**

Exploiter les fonctionnalités

- Nouvelle architecture



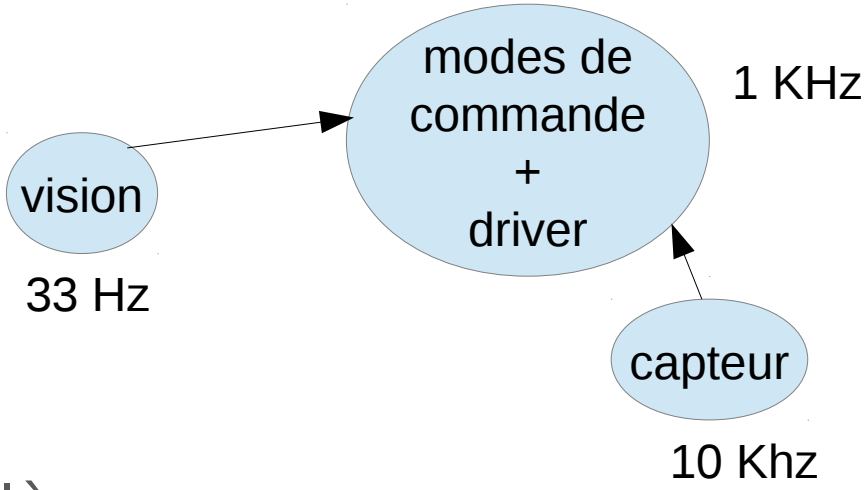
Solution: tout les modes dans un même noeud ROS

- Problème
  - **les boucles « driver » et « mode » sont localement synchrones mais désynchronisées entre elles.**

Impossible d'exprimer des contraintes entre le driver et le mode de commande pour garantir le bon ordonancement « lecture capteurs -> calcul commande -> écriture vers moteurs »

Exploiter les fonctionnalités

- Nouvelle architecture



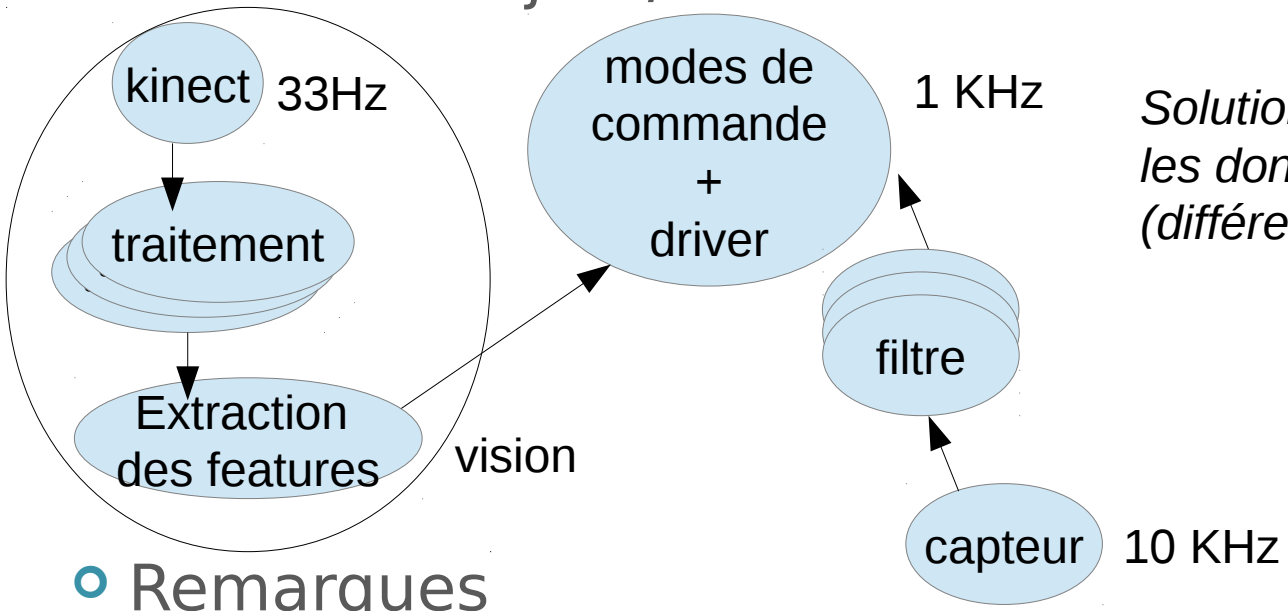
Solution: tout mettre dans un même node (l'ordonnancement est garanti puisque écrit à la main)

- Problème
  - Solution correcte mais objectivement difficile à réutiliser (on créé implicitement un nouveau couplage entre le driver et les autres nœuds ROS)

La pertinence de la granularité proposée par les nodes ROS « classique » se pose s'il faut tout mettre dans un même node...

Exploiter les fonctionnalités

- Le cas de l'ajout/sélection de fonctionnalités



Solution: ajout de nodes filtrant les données capteurs (différents filtres possibles)

- Remarques
  - Problème d'interférence entre nodes « filtre » ou traitement : émettent et produisent même topics.
  - Comment faire : à un moment donné de l'exécution un seul filtre actif ! Etonnament difficile à écrire en ROS.

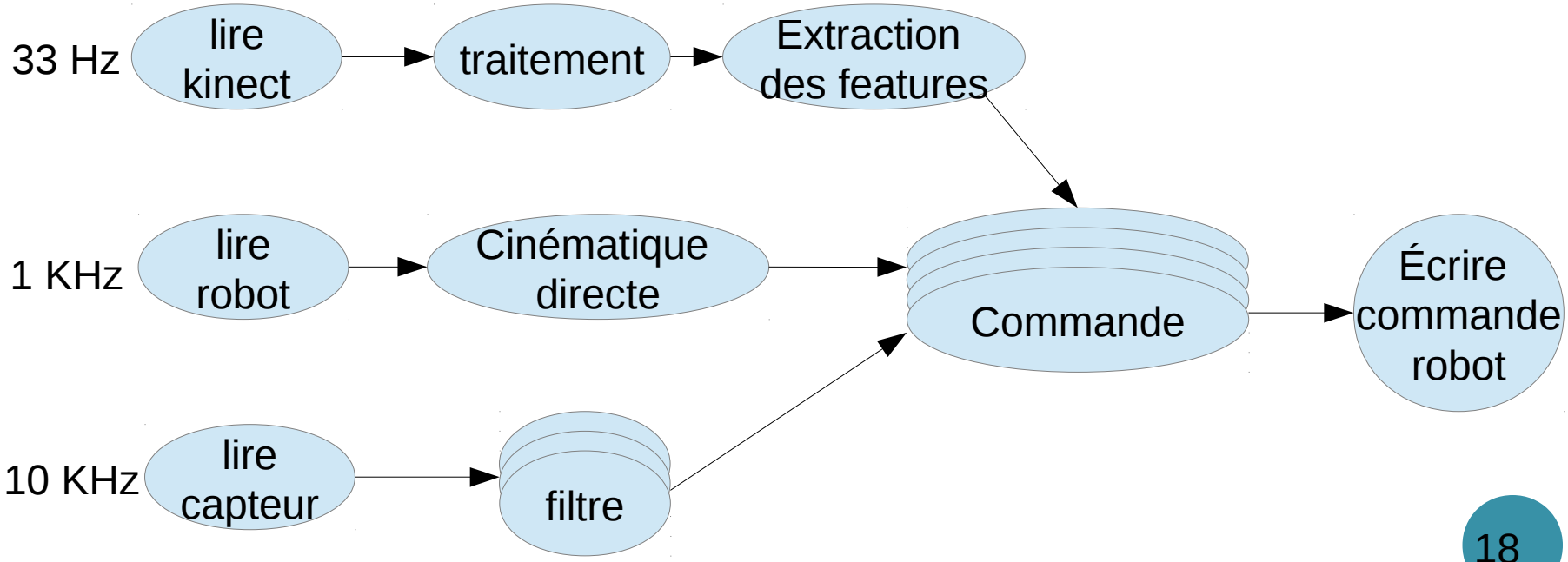
Problème de ROS : pas de contrôle d'exécution depuis l'extérieur des nodes

Remarques

- Tentative de définition
  - Une fonctionnalité robotique est considérée comme « à grain fin » si :
    - 1) elle est (ré-)utilisable de manière indépendante.
    - 2) elle délègue à un tier le contrôle de son exécution, de sa communciation et de sa configuration.
- ROS ne favorise pas un découpage à grain fin
  - Mais **il ne l'interdit pas** formellement
- Hypothèse
  - On peut obtenir un découpage à grain fin avec ROS
  - Sous quelles conditions ?

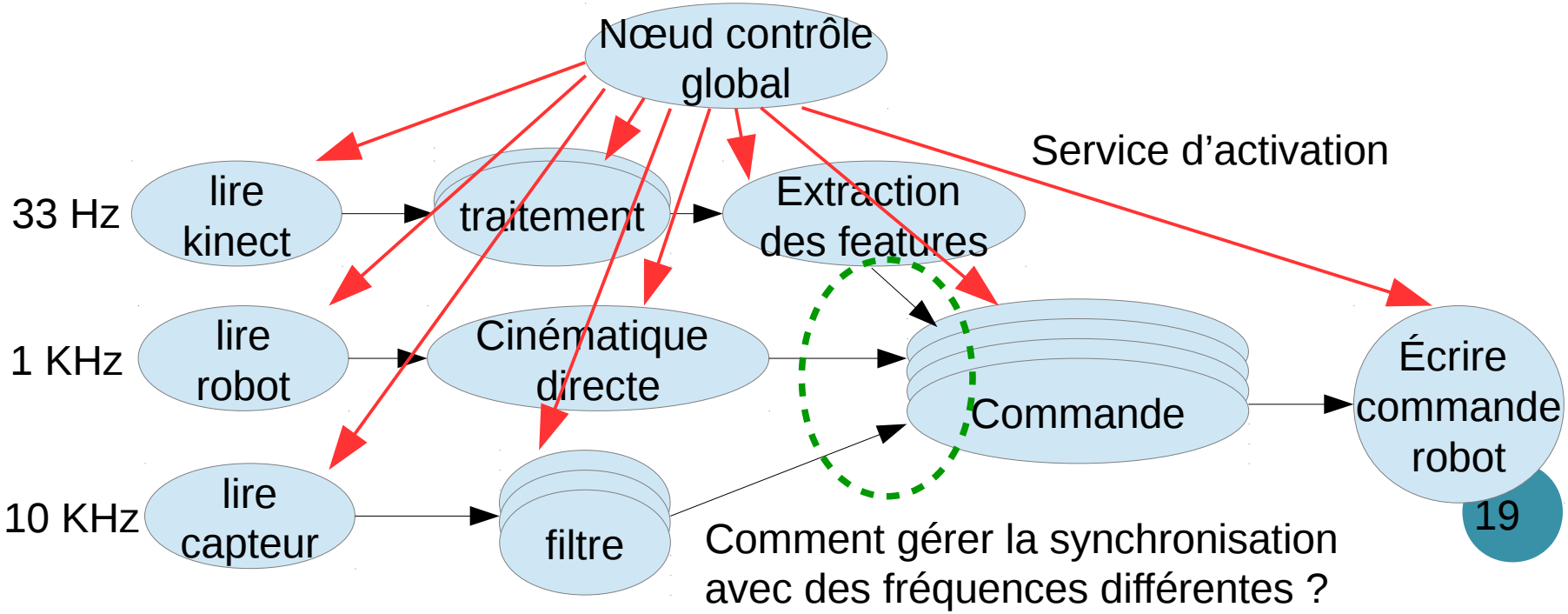
Validation de l'hypothèse

- Essai 1
  - 1 Noeud (ou nodelet) ROS = 1 fonctionnalité « à grain fin »
  - Connexion par topics pour l'échange de données
  - Problème : **interférence entre topics** (flux de données) générées par des nœuds « en conflit ».



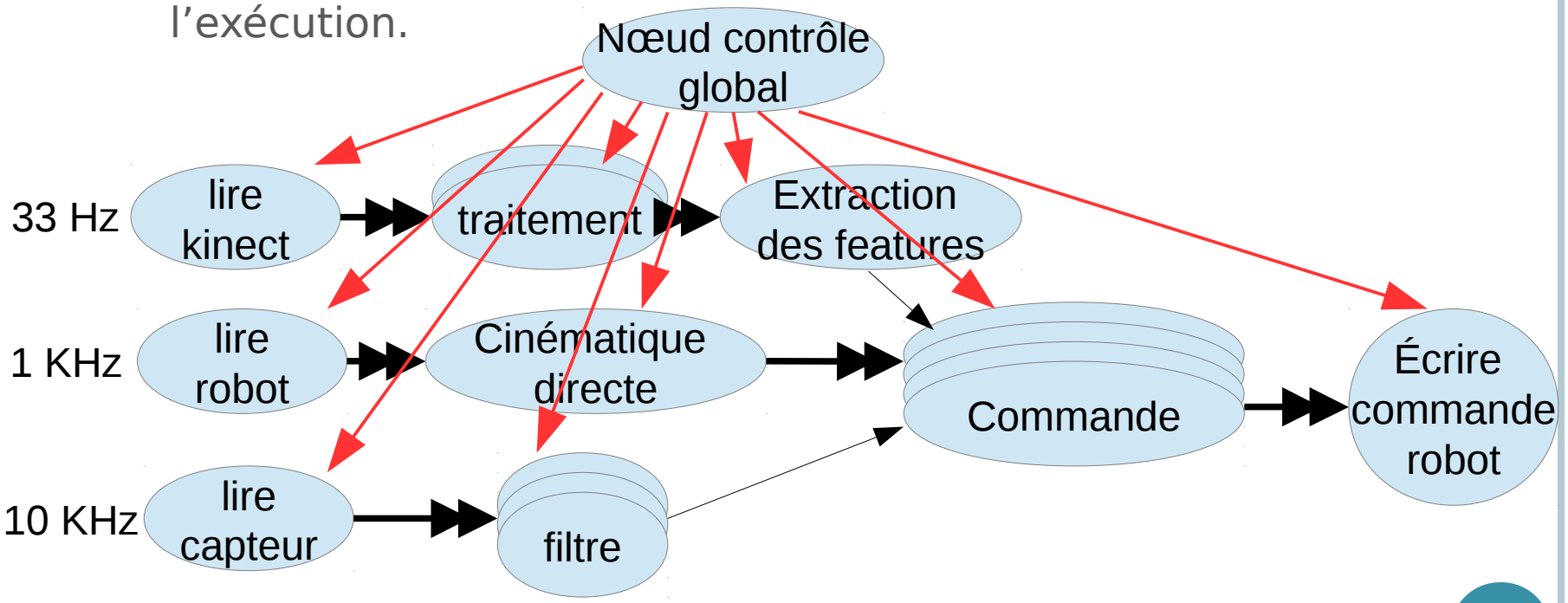
Validation de l'hypothèse

- Essai 2
  - nœud ROS spécifique pour contrôler l'exécution des autres noeuds
  - Chaque nœud propose un service pour l'activer/le désactiver (+configurer ses flux)
  - Problème : **synchronisation des topics** (flux de données)



Validation de l'hypothèse

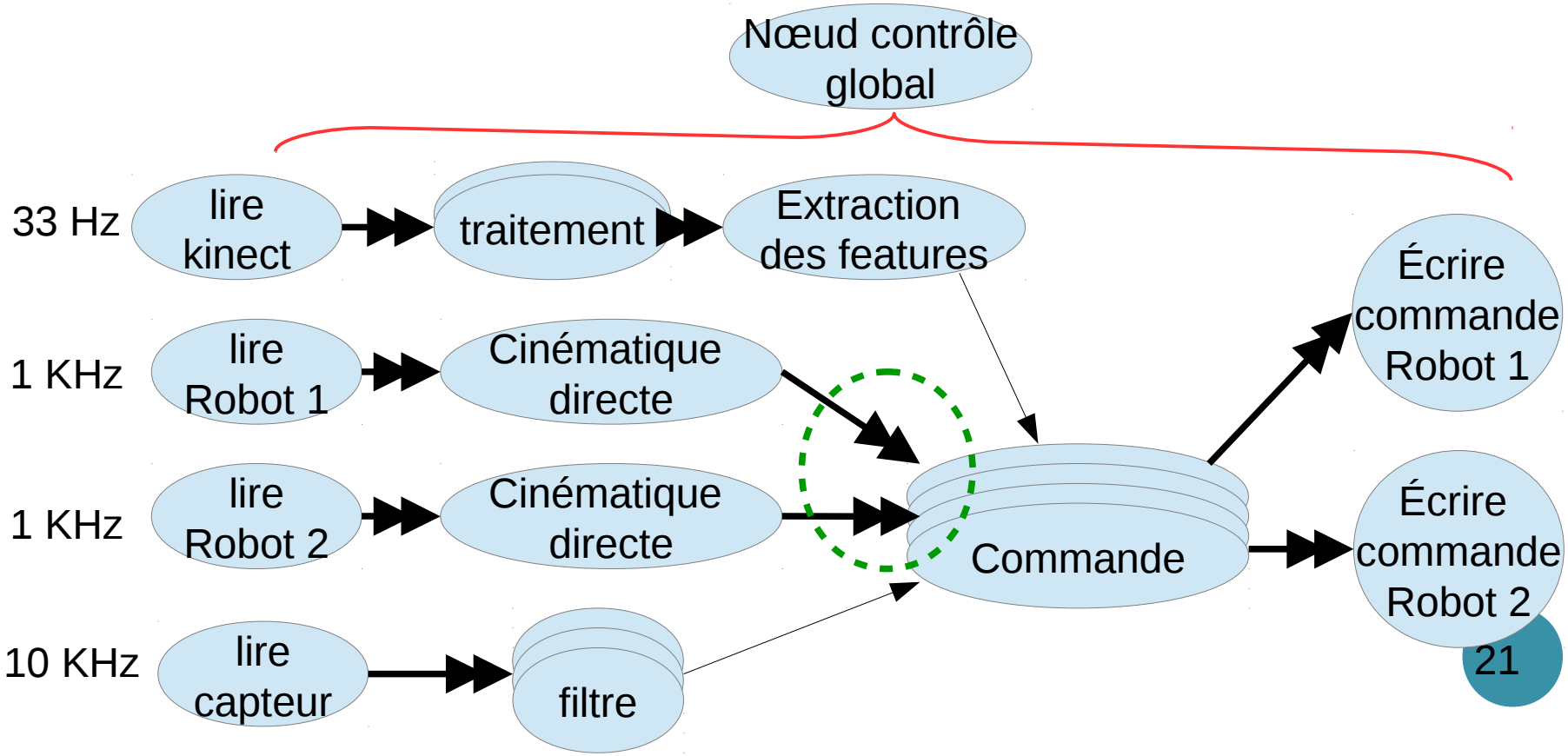
- Essai 3 :
  - l'unique topic « déclencheur de l'exécution » d'un nœud **actif** est celui à la plus haute fréquence de mise à jour.
  - Les autres topics sont simplement lus mais ne déclenchent pas l'exécution.



Complexité de la mise en œuvre (beaucoup de code glue à écrire)

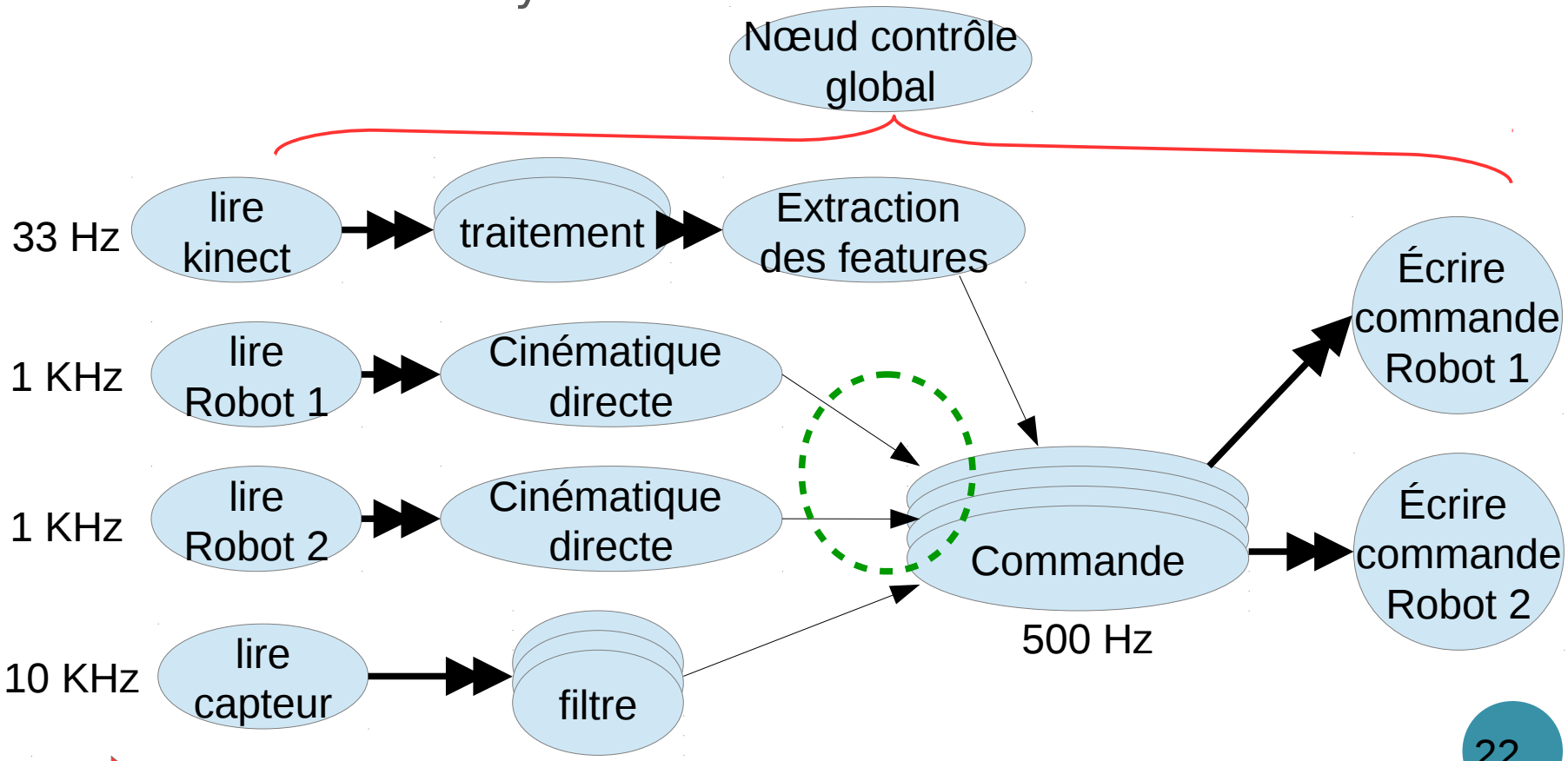
Validation de l’hypothèse

- Essai 4 : Ajout d’un deuxième dispositif commandé
  - Problème : Comment avoir 2 topics déclencheurs ?? Il faut gérer une **barrière de synchronisation**



Validation de l'hypothèse

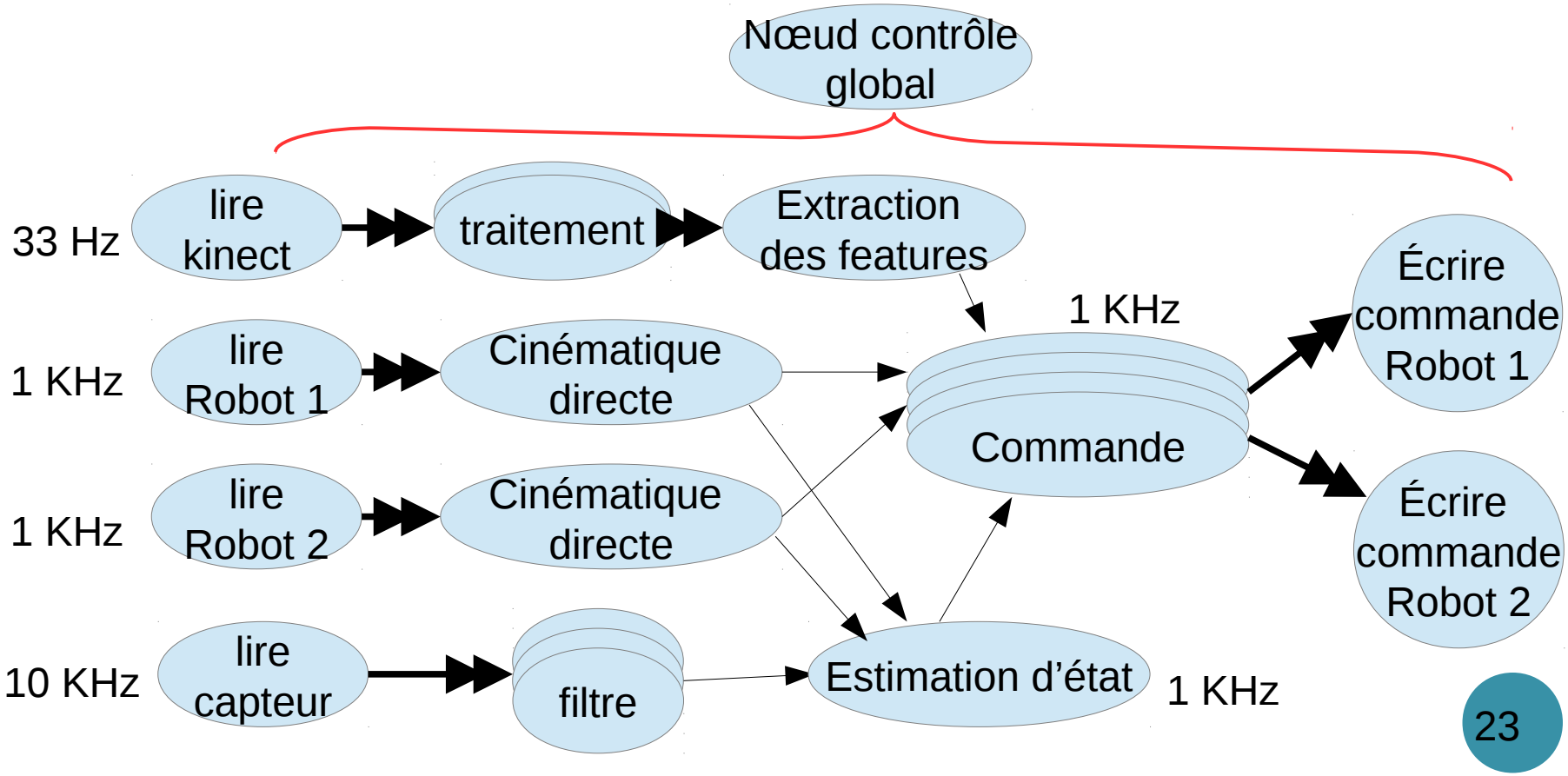
- Essai 5 : Fréquence des commandes différente
  - Barrières de synchronisation inutilisables



Solutions 4 et 5 pas homogènes

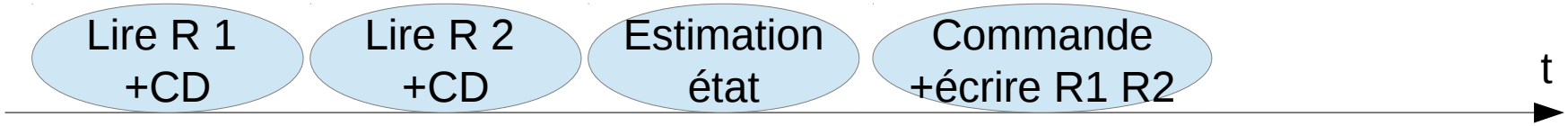
Validation de l'hypothèse

- Essai 6 : Plusieurs nœuds à une même fréquence
  - Problème : Comment garantir que l'odonnancement est correct ?



Validation de l'hypothèse

- Essai 6
  - Même problème posé dans Essai 5
  - On veut garantir que :



  - Problème, sans barrières de synchronisation on peut aussi bien avoir (suivant le cycle):

Problème d'ordonnancement : d'un point de vue temporel l'estimation d'état utilisée par la commande n'est pas cohérent avec les autres données.

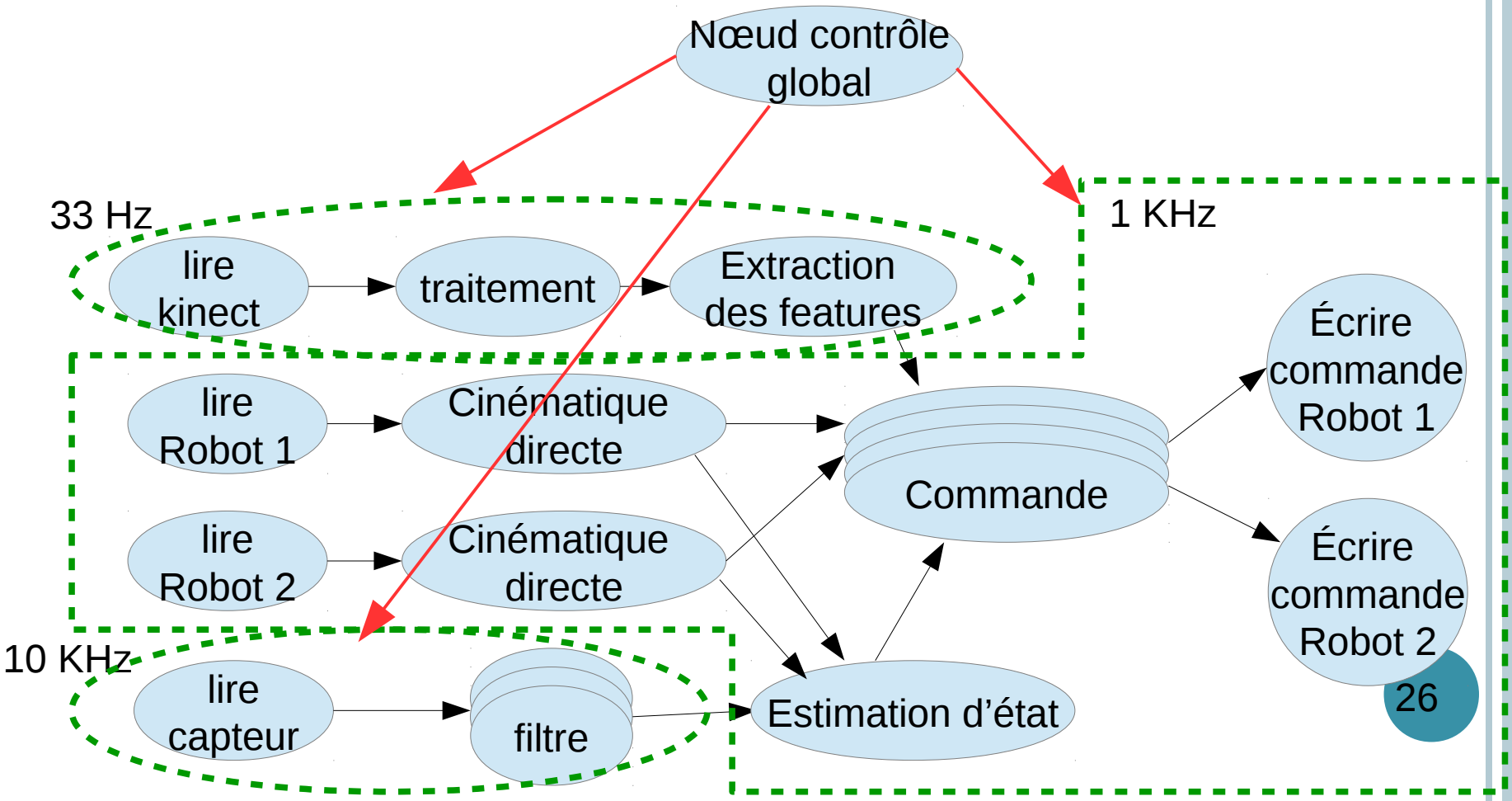
Hypothèse

- Solution **homogène** repose sur :
  - synchronisation logico-temporelle **globale** pour l'ensemble de l'architecture
  - Synchronisation gérée à l'extérieur des fonctionnalités
- Implémentation dans ROS :
  - Un nœud **contrôle l'exécution** des fonctionnalités
    - Doit implémenter un algorithme d'ordonnancement
    - « Alimenté » avec des **contraintes d'ordonancement** (fréquence d'exécution, précédence , exclusion)
  - Les échanges entre fonctionnalités se font uniquement par **topic non-bloquants**
    - pas de risque de dead-lock
  - Chaque nœud fournit une interface d'activation de sa fonctionnalité

Hypothèse avec ROS

Activation/Désactivation d'un ensemble de nœuds sous contrainte

Ensemble de contraintes d'ordonnancement



ROS & la synchronisation globale

- Conclusion :
  - possible de travailler à une granularité fine avec les nœuds ROS.
  - requiert des **efforts très importants** :
    - Standardiser la manière d'écrire les nodes : pas de boucle ; pas de synchronisation par topic ; service d'exécution normalisé
    - Ecrire un nœud capable de jouer le rôle d'**ordonnanceur applicatif.**
    - Spécifier pour chaque architecture les contraintes imposées aux nœuds ROS.
  - Pour que l'approche soit réellement utilisable en pratique il faudrait que ces mécanismes soient « fournis par défaut »
    - Nouveaux services dans ROS ?

Autres middleware

- YARP:
  - Mêmes problématiques qu'avec ROS.
- OROCOS:
  - Approche de programmation plus naturellement orientée vers un découpage à granularité fine.
  - + : Communication non bloquantes par défaut (ports de données)
  - - : Pas de gestion globale de l'ordonnancement (composants sont « autonomes » du point de vue de l'exécution)
- Genom (pocolibs):
  - Mêmes remarques que pour OROCOS

Implémenter cette approche avec des middleware RT semble plus « naturel »

Conclusion

- Travailler sur des composants à granularité fine
  - séparer les **aspects fonctionnels** des aspects **contrôle d'exécution** (tâches, communication, synchronisation)
  - Requiert de gérer les **synchronisations** entre composants à un niveau **global** (« architectural »)
  - Permet de contrôler l'exécution système à un niveau de précision plus élevé (vers des **garanties à l'exécution**)

- Middleware robotiques
  - Ne fournissent pas de mécanisme adapté pour l'ordonnancement global.

Vers de nouveaux services pour les middleware robotique ?
Peut-être plus simple de faire un nouveau middleware...

Merci de votre attention !

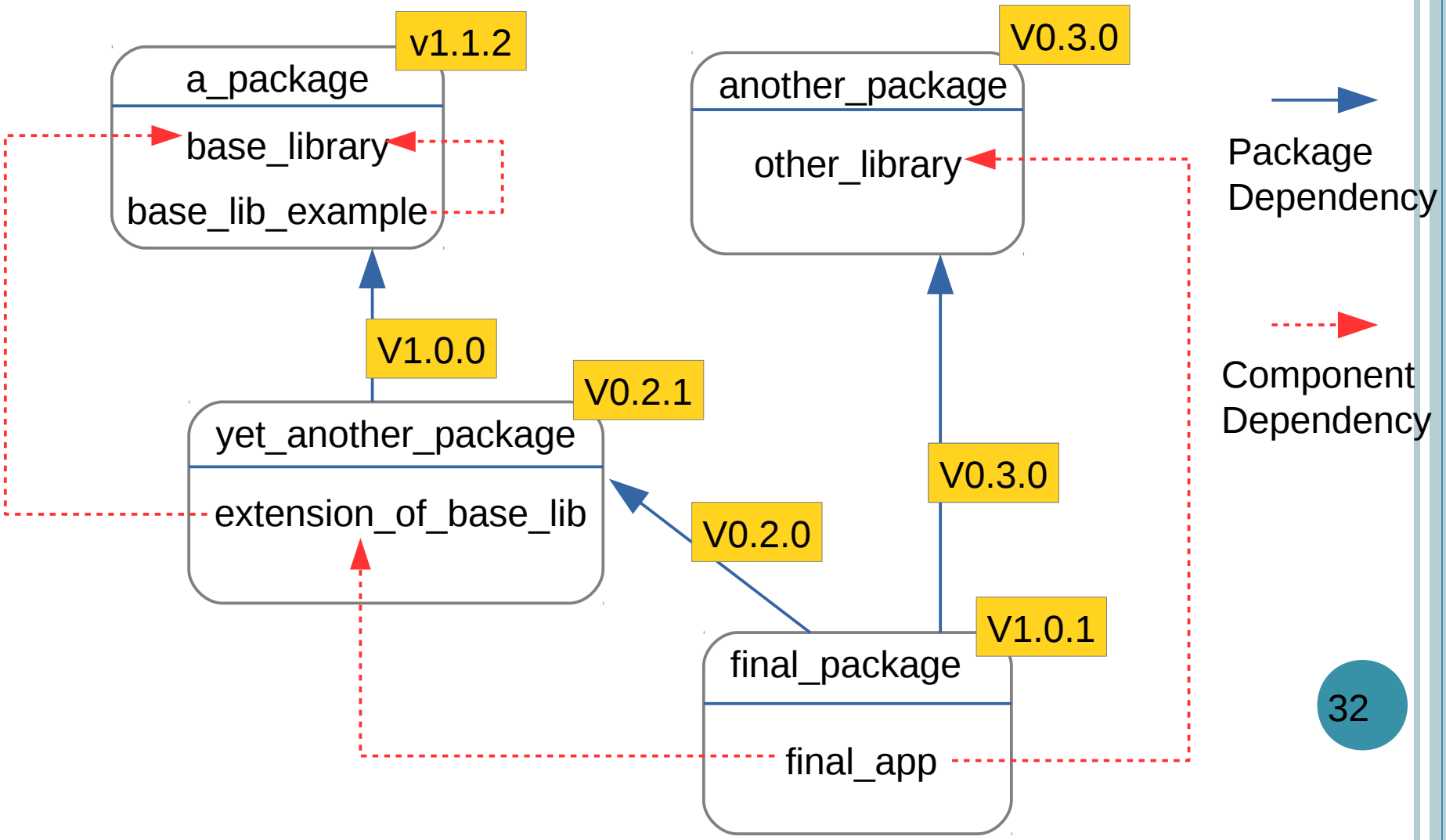
Gestion du cyle de vie : PID

- Techniquement
  - des commandes (générées par CMake) pour contrôler divers processus (build, déploiement, publication, génération de la documentation en ligne, etc.)
  - des API CMake utilisées pour descrire les projets.

- D’un point de vue fonctionnel
  - Reprend certains principes de catkin, maven, pkg-config, jrl-cmake-modules
    - Automatiser au max. toutes les étapes du processus de dév.
  - Description homogènes des packages
    - Information décrite **une seule fois**
    - Pas de multiples format de fichiers de configurations

Gestion du cyle de vie : PID

- Exemple de structure de projet basée sur PID



Gestion du cyle de vie : PID

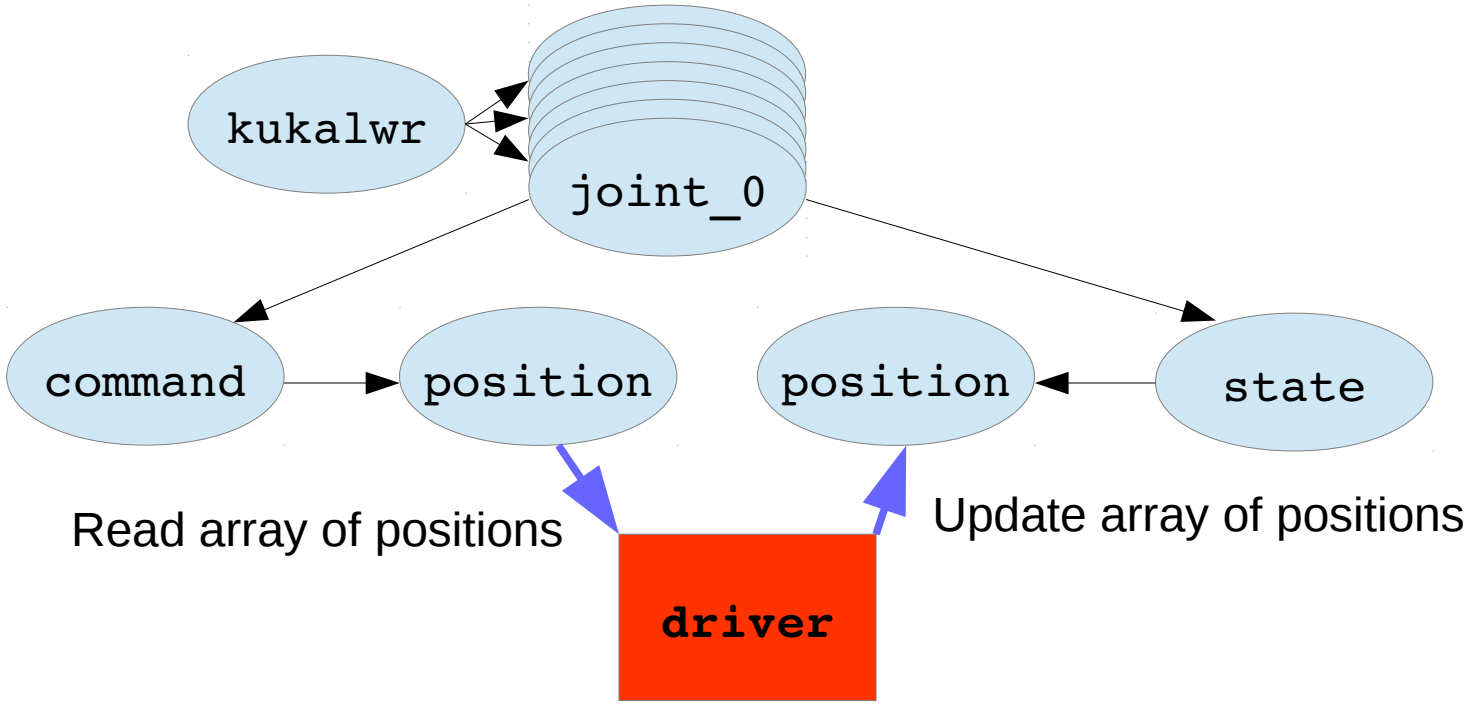
- Bilan
  - Globalement positif :
    - permet de gérer des projets avec des structures complexes
    - Amélioration notable de la qualité du code
    - Processus de publication / déploiement des packages permet de facilement diffuser les mises à jours et les nouveaux packages.
  - Point à améliorer :
    - La gestion de la validité des numéros de versions de package par automatisation des tests d’intégration (formalisation du semantic versionning).

Interopérabilité : Constat

- Problèmes pratiques posés par l’existant
  - Chaque API définit son système de type
    - Incompatibilité intrinsèque
  - API ont des recouvrement parfois importants (e.g. librairies de vision comme ViSP, opencv, arruco, cvd, etc.):
    - Chacune a des + et des – (généricité , optimisation, fiabilité, complétude), **difficile de faire un choix définitif**.
    - Objectif : les **utiliser de manière sélective**
  - API ne reposent pas sur des concepts et espaces de nommage standards (il n’y en a pas …ou peu cf. URDF)
    - induisent des variations dans les concepts manipulés, des collisions de nom sur des concepts différents (« tâche »).
    - Objectif : **« standardiser » ces concepts** … comment ??

Interopérabilité : knowbotics

- Exemple : interface des processors
  - contrainte exprimée par l'interface du driver
    - « lire les commande en positions des joints du KUKA LWR »
    - « mettre à jour les positions courantes des joints du KUKA LWR »



Interopérabilité : knowbotics

- Techniquement :
  - Un package fournissant une **API C++ coeur**
    - Définition des concepts standards
    - Mécanismes de base (description des contraintes en terme d'entrées / sortie, de propriétés)
  - Des package fournissant des **API génériques**
    - Implémentation d'algorithmes génériques à partir de code propre ou de logiciels tiers
    - Extension des concepts de base ou ajout de nouveau concepts.
  - Des package fournissant des **API spécifiques**
    - Implémentation d'algorithmes, de concepts et de drivers spécifiques à du **matériel**

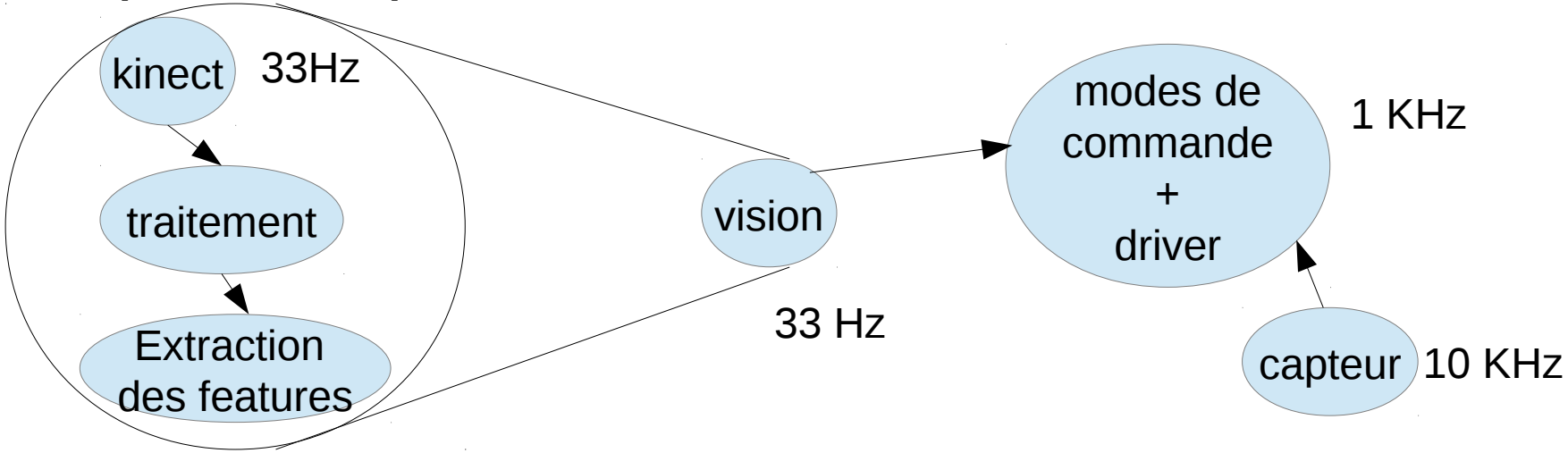
Interopérabilité : knowbotics

- Bilan
  - Points positifs :
    - Lisibilité du code
    - Facilité d’utilisation de librairies tierces
    - Fiabilité : la cohérence structurelle est vérifiée
    - **Favorise le travail à « granularité fine »** : description et contrôle de l’exécution à un niveau détaillé offre plus de possibilités
    - Surcout négligeable en terme de performances par rapport aux algorithmes eux-mêmes.
  - Point à améliorer (WIP):
    - Le standard requiert une meilleure formalisation des concepts pour permettre d’écrire des fonctions plus génériques, plus facilement configurables

Exploiter les fonctionnalités

- Le cas de la vision

Pipeline théorique sous forme de nodes



- Problèmes
  - pipeline de vision dans un unique node ! Raison : les données de vision « coûtent cher »
  - Pour changer de capteur, ou modifier un élément de la pipeline, il faut réécrire un node !