Le Framework NAOqi

[Mise à jour le 10/1/19]

Source

Cette page est une traduction du document Conceptes clés.

Qu'est-ce que le cadre NAOqi?

NAOqi est le nom du logiciel principal qui s'exécute sur le robot et le contrôle. Le **Framework NAOqi** est le cadre de programmation utilisé pour programmer les robots SOFTBANK.

Il répond aux besoins courants de la robotique, notamment: parallélisme, ressources, synchronisation, événements.

Principales charactéristiques

Ce Framework permet une communication homogène entre différents modules (mouvement, audio, vidéo), une programmation et un partage d'informations homogène.

Le Framework NAOqi est une *Cross platform* et un *Cross language* qui vous permettent de créer des applications distribuées.

Cross platform

Le framework NAOqi peut être utilisé sous Windows, Linux et MacOS. Il est possible de développer avec le framework NAOqi sur Windows, Linux ou Mac.

Utiliser Python: vous pourrez facilement exécuter votre code sur votre ordinateur ou directement sur le robot.

En utilisant C ++: comme il s'agit d'un langage compilé, vous devrez compiler votre code pour le système d'exploitation ciblé. Donc, si vous voulez exécuter du code C ++ sur le robot, vous devrez utiliser un outil de compilation croisée afin de générer un code capable de s'exécuter sur le système

d'exploitation du robot: NAOqi OS.

Suivez les instructions étape par étape C ++ SDK - Guide d'installation afin de vous assurer que vous avez installé tous les outils requis.

Cross language

Les logiciels exécutés sur le robot peuvent être développés en C ++ et en Python: dans les deux cas, les méthodes de programmation sont exactement les mêmes.

> **NB** : Les débutants devraient commencer par Python. Il est beaucoup plus facile d'apprendre et devrait répondre à tous leurs besoins.
> Les développeurs C ++ qualifiés développeront la plupart du temps:
> - les comportements en Python et
> - les modules en C ++.

Applications distribuées

Une application en temps réel peut être, non seulement un exécutable, mais également plusieurs processus et / ou modules distribués sur plusieurs robots. Quel que soit votre choix, les méthodes d'appel sont toujours les mêmes.

Connectez un exécutable à un autre robot en utilisant son adresse IP et son port, et toutes les méthodes API des autres exécutables sont disponibles exactement de la même manière qu'avec une méthode locale.

> Notez que NAOqi choisit automatiquement entre les appels directs rapides (LPC) et les appels distants (RPC).

Fonctionnement

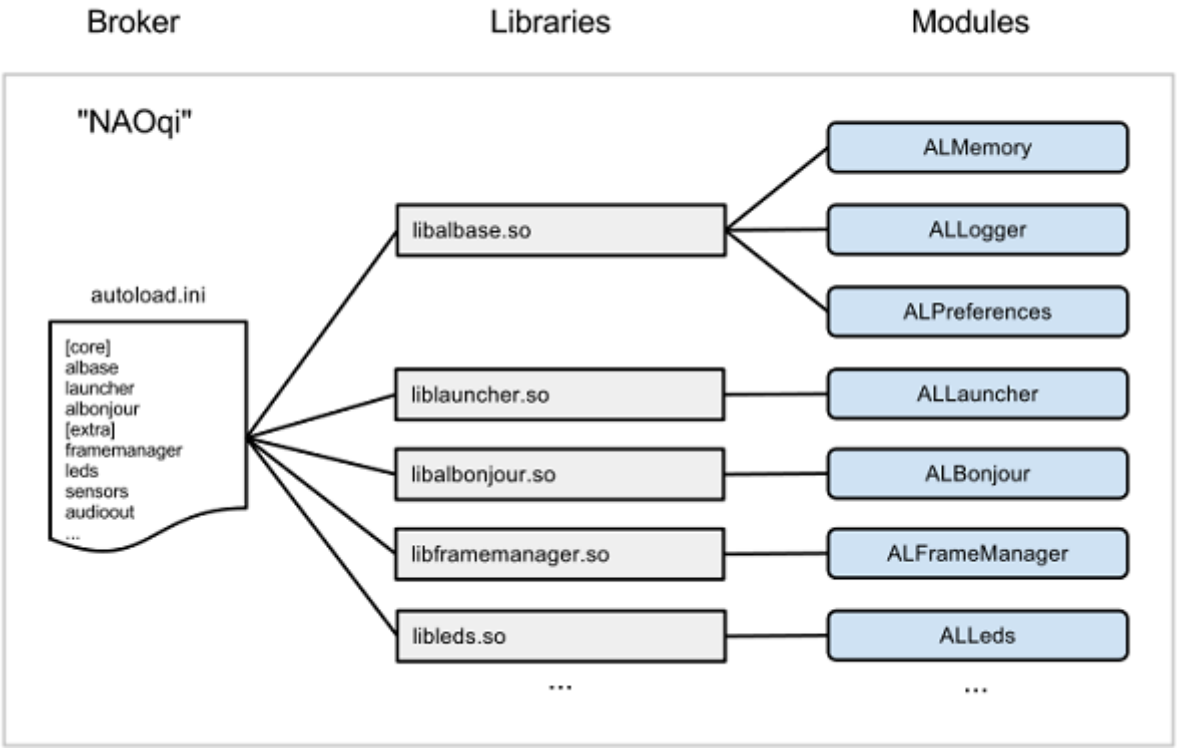
L'introspection est la base de l'API du robot, des capacités, de la surveillance et de l'action sur les fonctions surveillées. Le robot connaît toutes les fonctions API disponibles. La désinstallation d'une bibliothèque supprime automatiquement les fonctions API correspondantes. Une fonction définie dans un module peut être ajoutée à l'API avec un BIND_METHOD (défini dans almodule.h).

Lorsque vous appelez une fonction (avec trois lignes de code source uniquement), vous bénéficiez automatiquement des fonctionnalités suivantes:

- Fonction d'appel à la fois en C ++ et en Python (voir Langage croisé),
- Vous savez si la fonction est en cours d'exécution,
- Exécution de la fonction localement ou à distance (à partir d'un ordinateur ou d'un autre robot) (voir Applications distribuées),
- Appelle de wait, stop, isRunning sur les fonctions.

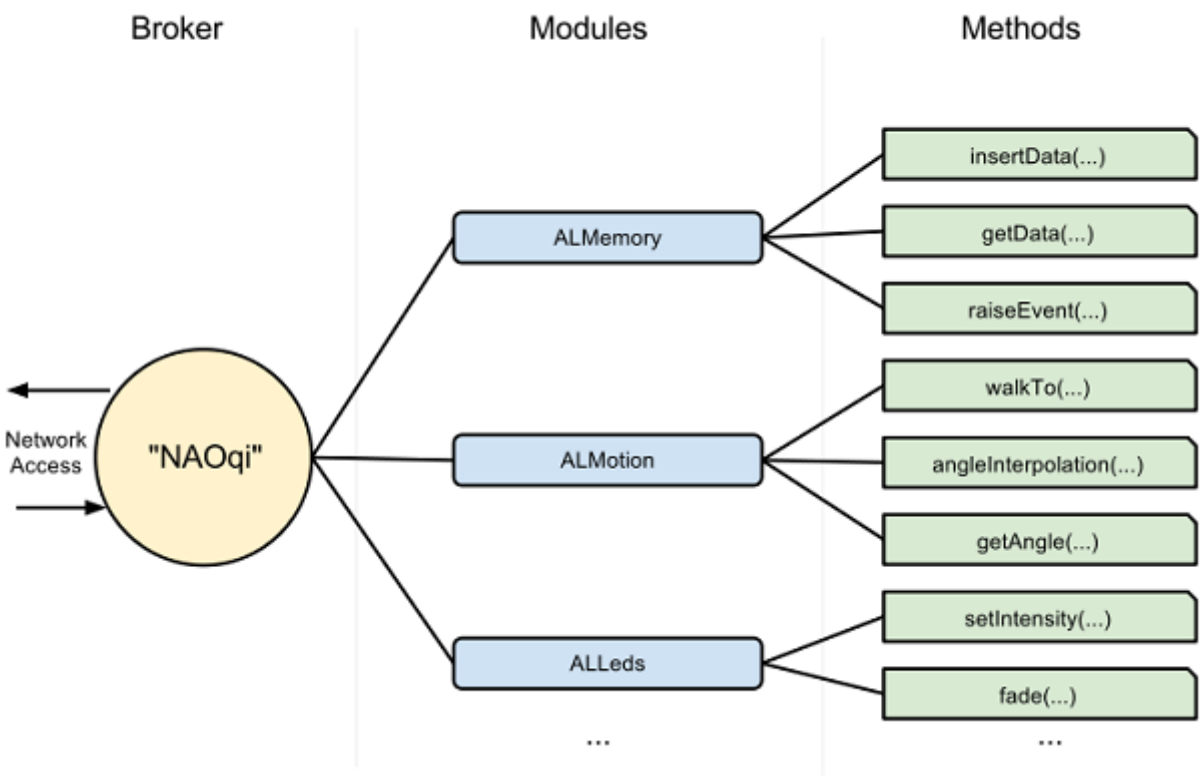
Le processus NAOqi

L'exécutable NAOqi est un courtier (broker). Au démarrage, il charge un fichier de configuration appelé autoload.ini qui définit les bibliothèques à charger. Chaque bibliothèque contient un ou plusieurs modules qui utilisent le courtier pour publier leurs méthodes.



Le courtier fournit des services de recherche afin que les modules de l'arborescence ou du réseau puissent trouver les méthodes exposées.

Le chargement des modules forme une arborescence de méthodes attachées aux modules et aux modules attachés à un courtier.



Courtier (Broker)

Un courtier est un objet qui fournit:

- des services d'annuaire: vous permettant de trouver des modules et des méthodes.
- un accès au réseau: permet d'appeler les méthodes des modules attachés en dehors du processus.

La plupart du temps, vous n'avez pas besoin de penser aux courtiers. Ils font leur travail de manière transparente, vous permettant d'écrire du code qui sera le même pour les appels aux “modules locaux” (dans le même processus) ou aux “modules distants” (dans un autre processus ou sur une autre machine).

Proxy

Un proxy est un **objet** qui se comportera comme le module qu'il représente.

Par exemple, si vous créez un proxy pour le module *ALTextToSpeech*, vous obtiendrez un objet contenant toutes les méthodes de ALTextToSpeech (say, etc).

Pour créer un proxy pour un module (et appeler ainsi les méthodes d'un module), vous avez deux possibilités:

- Utiliser simplement le nom du module. Dans ce cas, le code que vous exécutez et le module auquel vous souhaitez vous connecter doivent se trouver dans le même courtier. Ceci s'appelle un appel local.

```
self.tts = ALProxy('ALTextToSpeech') # Création d'un objet tts disposant des
```

```
méthodes du module ALTextToSpeech
self.tts.say("Bonjour tout le monde")
```

- Utiliser le nom du module, l'adresse IP et le port d'un courtier. Dans ce cas, le module doit être dans le courtier correspondant.

La différence entre les modules locaux et distants est expliquée dans la section Modules locaux.

Modules

Généralement, chaque module est une **classe** dans une **bibliothèque**. Lorsque la bibliothèque est chargée depuis **autoload.ini**, elle instancie automatiquement la classe du module.

Dans le constructeur d'une classe dérivée de ALModule, vous pouvez “lier” les méthodes. Cela annonce leurs noms et signatures de méthode au courtier afin qu'ils deviennent disponibles pour les autres.

Un module peut être distant ou local.

- S'il est distant, il est compilé en tant que fichier exécutable et peut être exécuté en dehors du robot. Les modules distants sont plus faciles à utiliser et peuvent être facilement débogués de l'extérieur, mais ils sont moins efficaces en termes de vitesse et d'utilisation de la mémoire.
- S'il est local, il est compilé en tant que bibliothèque et ne peut être utilisé que sur le robot. Cependant, ils sont plus efficaces qu'un module distant.

Chaque module contient différentes méthodes. Parmi celles-ci, certaines méthodes sont liées, ce qui signifie qu'elles peuvent être appelées depuis l'extérieur du module, par exemple dans un autre module, depuis un exécutable etc. La manière d'appeler ces fonctions liées ne varie pas si le module est distant ou local: le module s'adapte automatiquement.

L'API du module est visible depuis la page Web du robot.

Modules locaux

Deux modules (ou plus) lancés dans le même processus sont locaux. Ils communique en utilisant **UN** seul courtier (broker).

Comme les modules locaux sont dans le même processus, ils peuvent partager des variables et appeler les méthodes des autres sans sérialisation ni mise en réseau. Cela permet une communication rapide entre les modules.

Si vous devez réaliser un système en boucle fermée (asservissement par exemple), vous **DEVEZ** utiliser des modules locaux.

Modules distants

Les modules distants sont des modules qui communiquent via le réseau. Un module distant a besoin d'un courtier (broker) pour parler aux autres modules. Le courtier est responsable de toute la partie

réseau. Vous devez savoir que les modules distants fonctionnent avec SOAP sur le réseau. Vous ne pouvez pas faire d'accès rapide en utilisant un module distant (accès direct à la mémoire par exemple).

Connexion courtier à courtier

Vous pouvez connecter deux modules en connectant leurs courtiers.

Par exemple, vous avez deux modules B et C. Lorsque vous connectez leurs courtiers, B peut accéder aux fonctions de C et C peut accéder aux fonctions de B.

Pour connecter des modules de cette manière, vous devez spécifier l'adresse IP et le numéro de port du courtier principal. (–pip, option de ligne de commande –pport lorsque vous démarrez votre module). Ensuite, vous pouvez accéder au module en obtenant un proxy:

```
AL :: ALProxy proxy = AL :: ALProxy (<modulename>);
```

Étant donné que le courtier du module est déjà connecté à l'aide de –pip et –pport, vous n'avez pas besoin de spécifier l'adresse IP et le numéro de port lorsque vous créez un proxy.

Connexion proxy à courtier

Vous pouvez connecter votre module à un autre sans spécifier –pip et –pport. Pour ce faire, vous devez créer un proxy dans votre module et le connecter à l'adresse IP et au numéro de port du courtier que vous souhaitez.

Par exemple, vous avez deux modules B et C. Lorsque vous connectez B à C simplement en utilisant un proxy, B peut accéder aux fonctions C MAIS C ne peut pas accéder aux fonctions B.

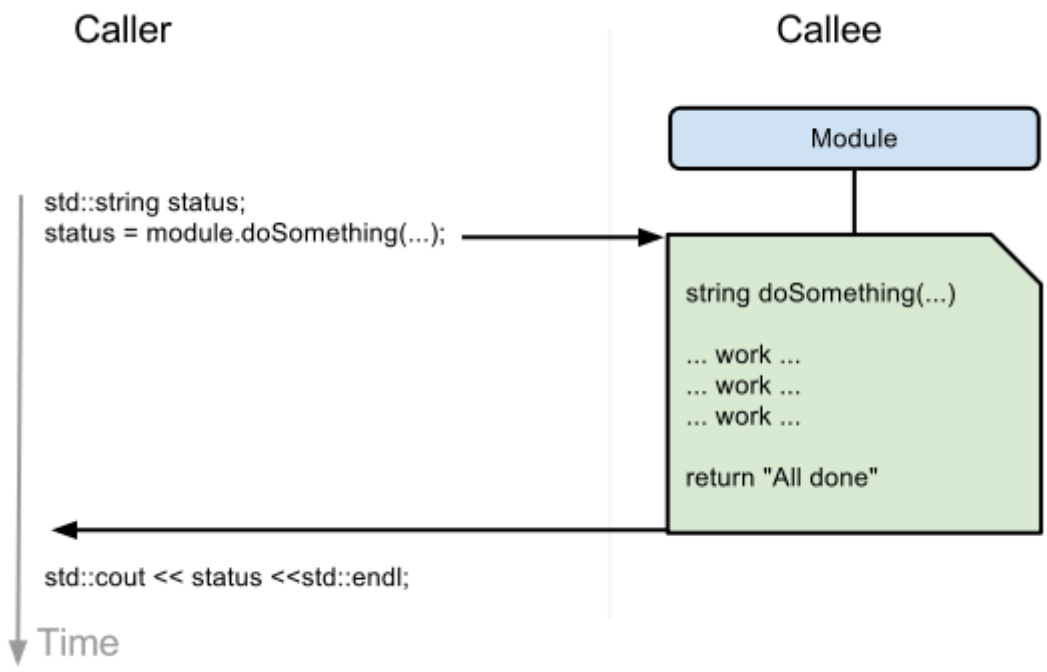
```
// A broker needs a name, an IP and a port to listen:
const std::string brokerName = "mybroker";
// NAOqi ip
const std::string pip = "127.0.0.1"; // local NAOqi
// NAOqi port
int pport = 9559;

// Create your own broker
boost::shared_ptr<AL::ALBroker> broker =
  AL::ALBroker::createBroker(brokerName, "0.0.0.0", 54000, pip, pport);
AL::ALProxy proxy = AL::ALProxy(broker, <modulename>);
```

Appels bloquants et non bloquants¶

NAOqi offre deux possibilités pour appeler des méthodes:

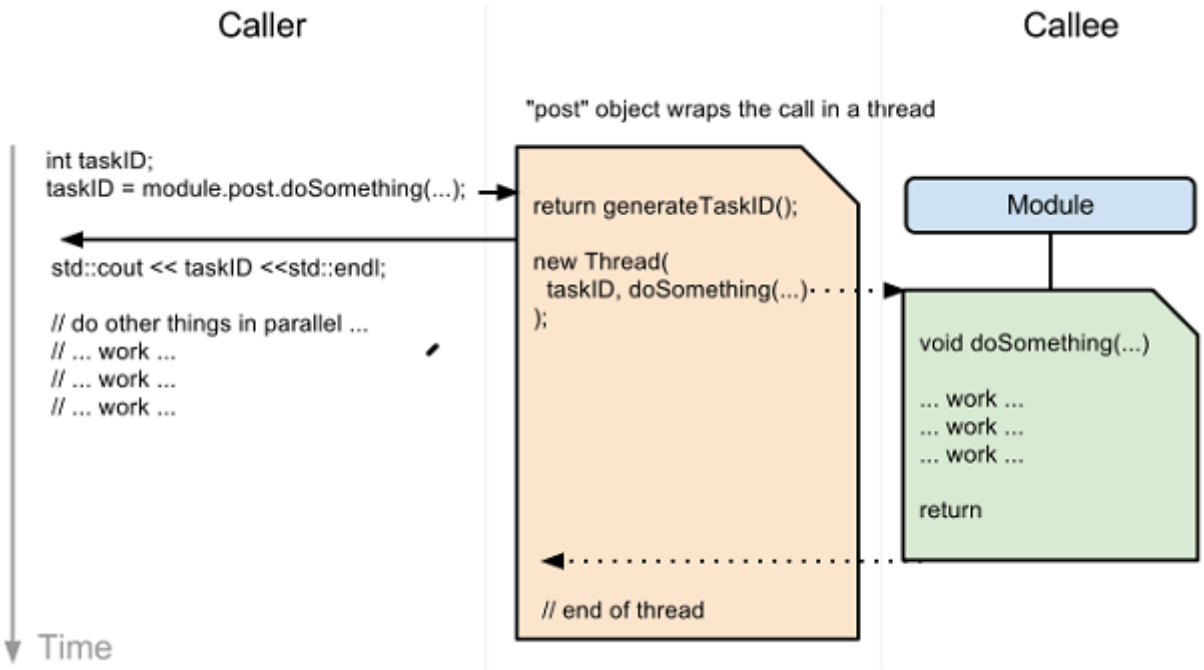
- **Appels bloquants**

En fonctionnement normal, les appels de méthode sont bloquants - La prochaine instruction sera exécutée après la fin de l'appel précédent. Tous les appels peuvent déclencher une exception et doivent être encapsulés dans un bloc **try-catch**. Les appels peuvent avoir des valeurs de retour.



- **Appels non bloquants**

En utilisant *post objet* d'un proxy, une tâche est créée dans un thread parallèle. Cela vous permet d'effectuer d'autres tâches en même temps (par exemple, marcher pendant que vous parlez). Chaque post-appel génère un identifiant de tâche (task id). Vous pouvez utiliser cet ID de tâche pour vérifier si une tâche est en cours d'exécution ou attendre que la tâche soit terminée.



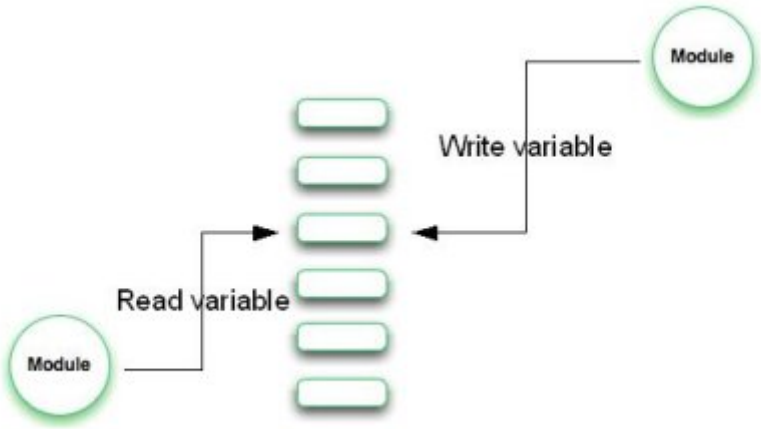
Mémoire

ALMemory est la mémoire du robot. Tous les modules peuvent lire ou écrire des données, s'abonner à des événements pour être appelés lorsque des événements sont déclenchés.

Sachez que **ALMemory** n'est pas un outil de synchronisation en temps réel. Limite l'abonnement à DCM / time ou motion / synchro ou variable en temps réel.

ALMemory

ALMemory est un tableau d'**ALValues** (pour plus de détails, voir: Bibliothèque ALValue). L'accès à une variable est thread-safe. Nous utilisons des sections critiques en lecture / écriture pour éviter les mauvaises performances lors de la lecture de la mémoire.



ALMemory contient trois types de données:

- Les données des actionneurs et capteurs,
- Les événements,
- Les micro-événement.

Réagir aux événements

Quelques modules exposent également certains événements.

Vous devez vous abonner à l'événement à partir d'un autre module, en utilisant un rappel qui doit être une méthode de votre abonné.

Par exemple, vous pouvez avoir un module appelé *FaceReaction* contenant une méthode *onFaceDetected*.

Vous pouvez abonner le module *FaceReaction* à la méthode *FaceDetected* du module *ALFaceRecognition* avec le rappel *onFaceDetected*.

Cela provoquera l'exécution de l'algorithme de détection de visage, et chaque fois qu'un visage est détecté, le rappel *onFaceDetected* sera appelé.

Pour voir comment cela se fait en Python, veuillez consulter la section Réagir aux événements.