BUREAU D'ETUDE GYRO-PODE

ANNEXES AU SUJET DE TD & TP

DEPARTEMENT DE GENIE ÉLECTRIQUE ET INFORMATIQUE

Responsable pédagogique : Claude BARON
Support : Sébastien DI MERCURIO

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Table des matières

1. PRISE EN MAIN DE L'ENVIRONNEMENT DE TP

Toute la partie qui suit est décrite dans la vidéo sous MOODLE.

Ceci n'est qu'un mémo si la vidéo n'est pas accessible.

La première chose à faire est de récupérer le dépôt Git contenant le projet de base (vous pouvez également télécharger le zip sous MOODLE et décompresser le fichier dans votre répertoire de travail). Pour cela, connectez-vous sous Ubuntu et dans un terminal, créez un répertoire « gyropode » dans votre espace personnel :

```
mkdir gyropode
cd gyropode
```

A partir de là, il faut récupérer le dépôt git :

```
git clone https://github.com/INSA-GEI/segway.git
git checkout stable
```

Vous obtenez un répertoire « segway » contenant différents répertoires, notamment :

- docs : contient la doc technique ainsi que les sujet de TP et TD
- monitor: contient le projet de l'IHM. L'exécutable se nomme monitor-python.py et se trouve dans le répertoire monitor-python-gui, avec l'ensemble du code de l'IHM
- raspberry : contient le code du superviseur, sous forme d'un projet NetBeans, à compiler et exécuter sur la carte Raspberry

Pour lancer l'interface graphique, exécutez la commande suivante (dans le terminal sur le PC) :

```
./segway/monitor-python-gui/monitor-python.py &
```

Le « & » à la fin de la commande est important ! Ensuite, lancez NetBeans pour ouvrir le projet superviseur (toujours dans un terminal) :

```
netbeans &
```

Referez-vous à [] pour voir comment ouvrir un projet, ajouter une cible de compilation à distance (la carte Raspberry du simulateur) et compiler votre code.

Ouvrez le projet se trouvant dans « segway/raspberry ». C'est votre projet de base, se contentant de transférer les paramètres remontés par le STM32 directement sur l'IHM. Les fichiers « tasks.cpp » et « tasks.h » doivent être complétés. Vous n'avez pas besoin de modifier les autres fichiers mais vous pouvez les consulter. Accessoirement, vous pouvez modifier « parametres.cpp » et « parametres.h » mais ce n'est pas nécessaire.

Si l'édition du code et sa compilation se font bien à travers NetBeans, l'exécution du superviseur doit se faire à la main, la faute aux droits d'administration nécessaire à l'exécution d'un programme Xenomai. Pour cela, dans un autre terminal, connectez-vous à votre simulateur (pensez à mettre l'adresse IP de votre cible (qui se trouve indiquée sur la carte de droite. Elle est de la forme 10.105.0.XX)

```
ssh xenomai@<adresse IP de la cible>
```

Les informations de connexion sur le simulateur sont :
Login : xenomai
Mot de passe : xenomai

Une fois connecté, votre programme se trouve au bout d'une arborescence ayant la forme suivante :

```
cd .netbeans/remote/10.105.0.64/insa-<Nom de votre machine>-Linux-x86_64/<Chemin vers le depot git>/segway/raspberry/dist/Debug_Raspberry/GNU-Linux/
```

où :

- <Nom de votre machine> : nom de la machine de TP, sous la forme insa-xxxx, xxxx étant le numéro de la machine.
- <Chemin vers le dépôt git> : chemin complet (à partir de /) ou se trouve le dépôt git sur votre compte.

Une fois que vous êtes entré dans ce répertoire se trouve le programme « segway_supervisor ». Pour l'exécuter avec les droits administrateur (nécessaire pour Xenomai), il suffit de taper la commande :

```
sudo ./segway_supervisor
```

Dernier point : l'ensemble du code fourni utilise la convention de nommage du C#, à savoir un mélange de camelCase et de PascalCase. Le camelCase consiste à mettre en majuscule les premières lettres de chaque mot d'un nom, sauf le premier mot. Le PascalCase met en majuscule toutes les premières lettres de chaque mot. Ceci permet de rapidement différentier une variable et une méthode.

Pour info, voici un résumé rapide des différentes formes d'écriture rencontrées dans le code :

Nom de l'objet	Notation	Exemple
Nom de classe	PascalCase	MaClasse
Constructeur/Destructeur	PascalCase	MaClasse();
Méthode	PascalCase	void MaMethode();
Arguments de méthode	camelCase	void MaMethode(int unParametre);
Attributs, variable locale	camelCase	int compteurDeVitesse;
Constantes	MAJUSCULE	#define VALEUR_INITIALE 0
Type énuméré	PascalCase	typedef enum {…} MonEnumeration;

Tableau 1: Prise en main - Règles d'écriture

1.1 Analyse du code fourni

Une première version du superviseur vous est donnée. Son unique objectif est de vous montrer comment organiser du code C++ et comment manipuler des tâches, des sémaphores, des files de messages, etc. La réception de messages en provenance et vers le STM32, l'envoi de message vers le moniteur ainsi que la mise a jour des paramètres du système sont déjà fait, utilisant des méthodes de synchronisation (sémaphore, message queues) que vous pouvez modifier si vous le souhaitez. L'intérêt est de voir comment définir et utiliser une tache, un sémaphore, un message queue.

Il vous reste donc principalement à gérer les cas d'urgence et la régulation de couple.

Le programme initialement fourni se contente d'envoyer les messages en provenance du STM32 vers la tache d'asservissement qui retransmet directement à la tache d'affichage et donc, au final au moniteur.

Votre objectif dans ce TP sera :

- Assurer l'asservissement dans la tache d'asservissement
- Gérer les situations d'urgence (chute de l'utilisateur, batterie faible)

Le programme est composé de 3 tâches comme suit :

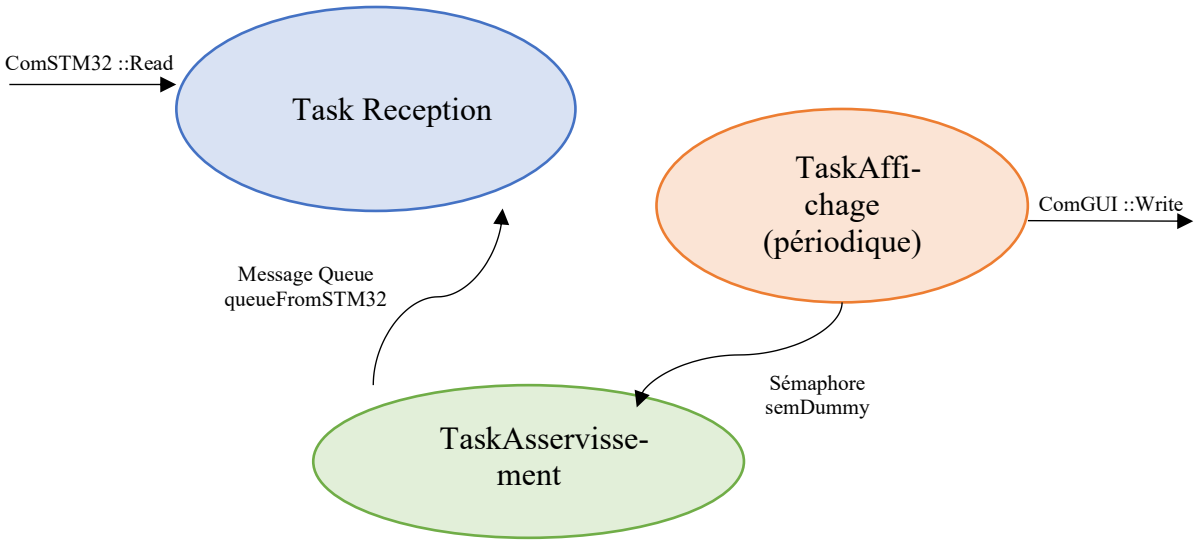


Figure : Prise en main - Organisation des tâches initiales

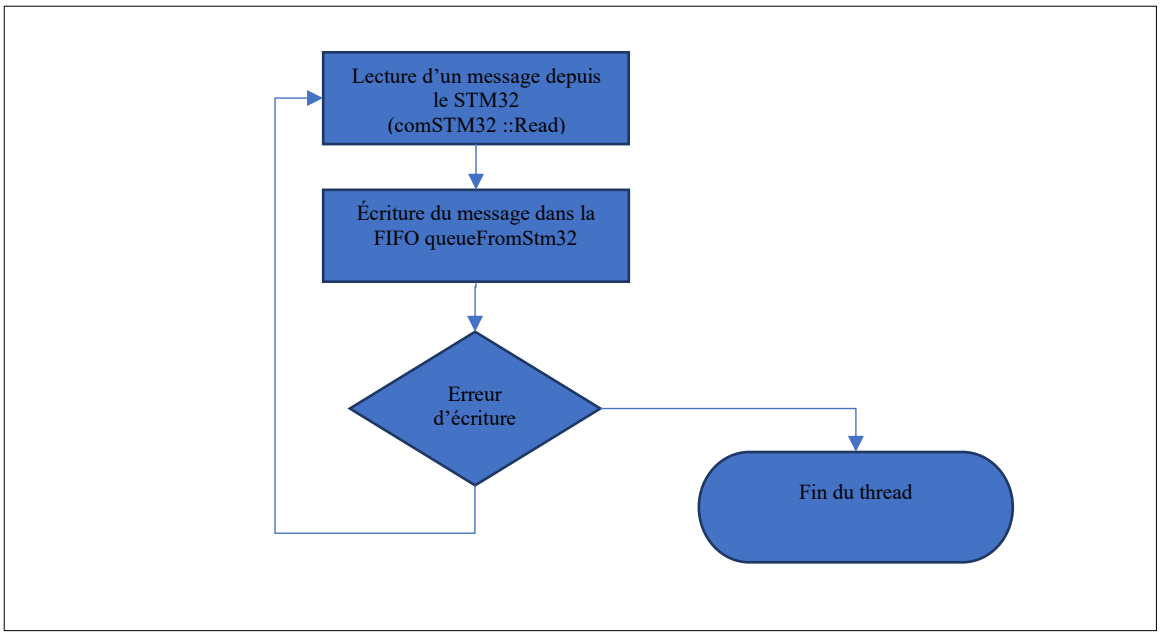
- Comportement de la tâche TaskReception

Le superviseur reçoit de manière régulière des données en provenance du STM32 (position, angle, vitesse, niveau de batterie, présence de l'utilisateur…). Il met ces données à disposition afin de permettre : leur affichage, l'asservissement, la surveillance de l'état du système, etc. Ce thread fonctionne à une fréquence de 50 Hz.

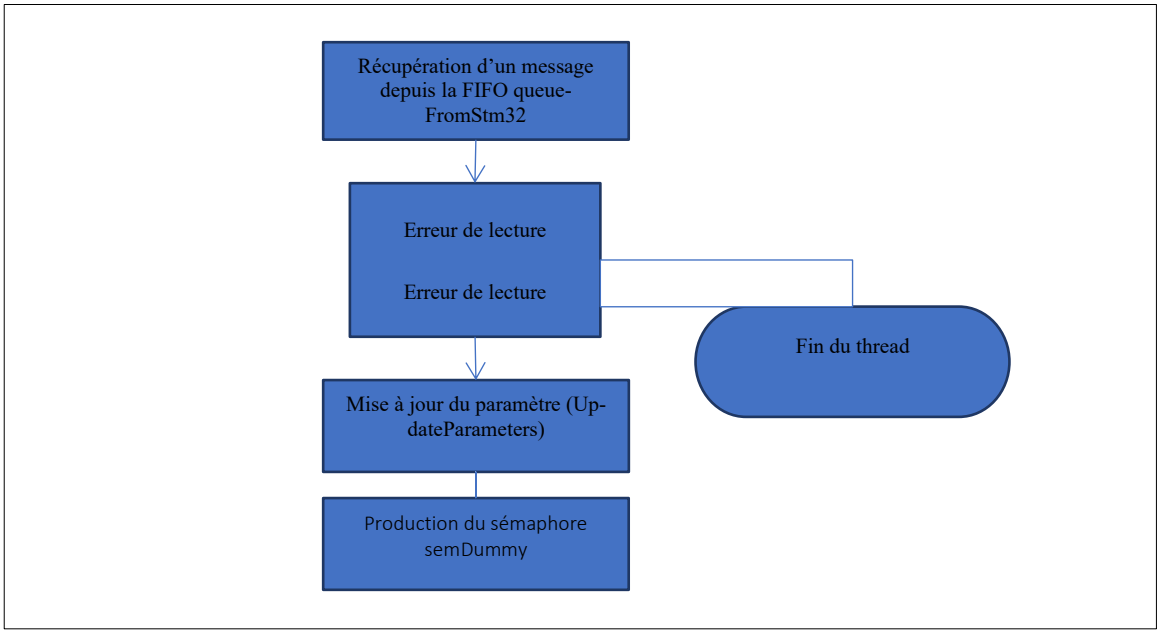
Donnée	Type	Unité
Position angulaire	float	rad
Vitesse angulaire	float	rad /s
Niveau batterie	integer	%
Vitesse linéaire	float	m/s
Présence utilisateur	integer	1 si présent, 0 sinon

Tableau 2: Prise en main - Liste des données et leurs types

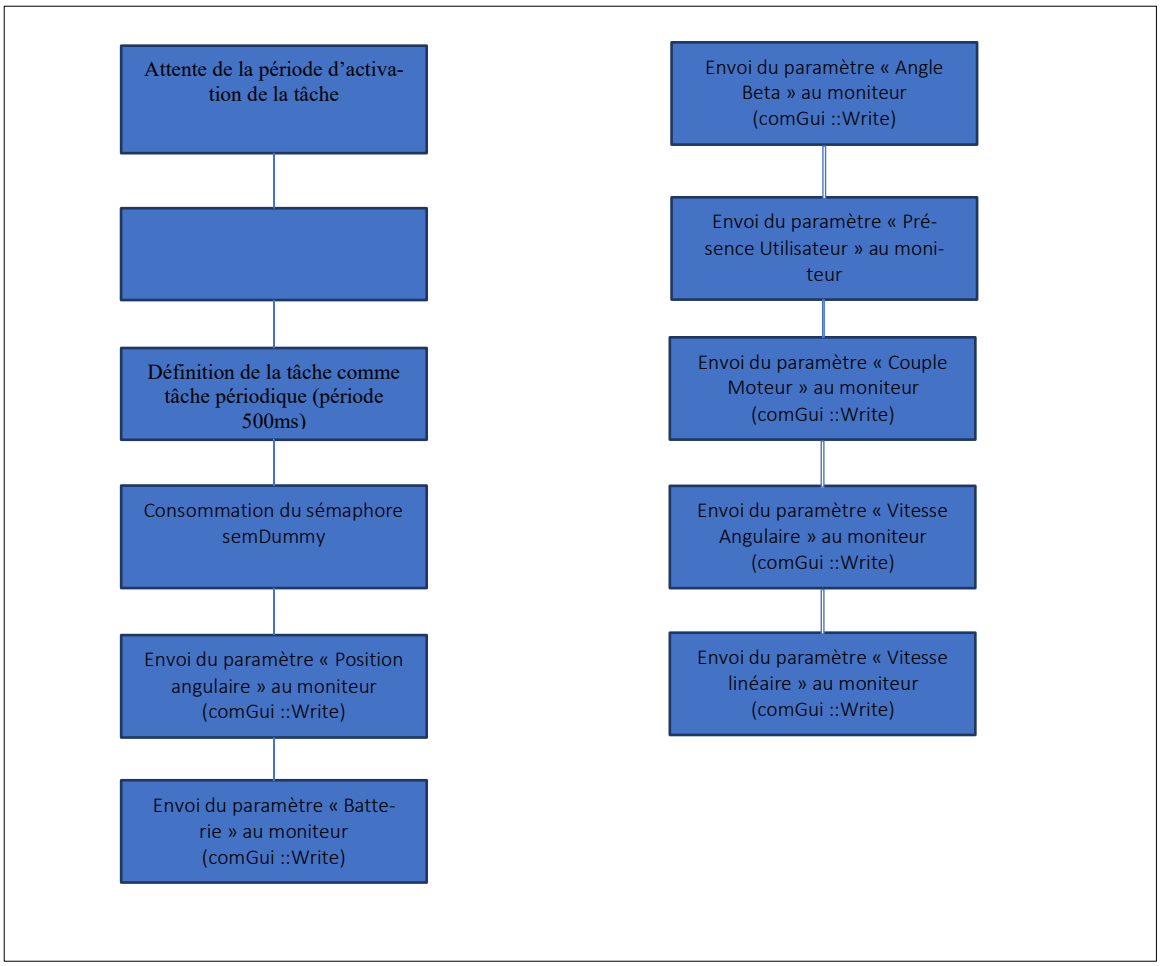
Plus d'information sur la structure des messages et leur format peut être trouvée en []



- Comportement de la tâche TaskAsservissement



- Comportement de la tâche TaskAffichage


Attente de la période d'activation de la tâche
Définition de la tâche comme tâche périodique (période 500ms)
Consommation du sémaphore semDummy
Envoi du paramètre « Position angulaire » au moniteur (comGui ::Write)
Envoi du paramètre « Batterie » au moniteur (comGui ::Write)
Envoi du paramètre « Angle Beta » au moniteur (comGui ::Write)
Envoi du paramètre « Présence Utilisateur » au moniteur
Envoi du paramètre « Couple Moteur » au moniteur (comGui ::Write)
Envoi du paramètre « Vitesse Angulaire » au moniteur (comGui ::Write)
Envoi du paramètre « Vitesse linéaire » au moniteur (comGui ::Write)

2. DIAGRAMMES DE CLASSE

2.1 Classes principales (temps réelles)

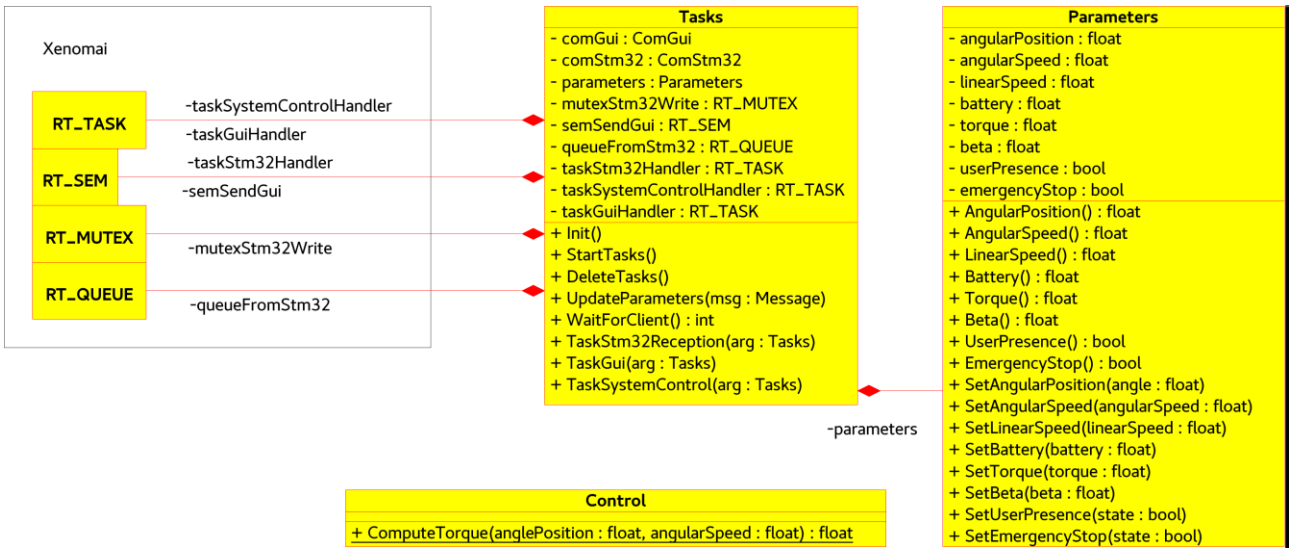


Figure 1 : Classes principales

2.2 Classes de messages

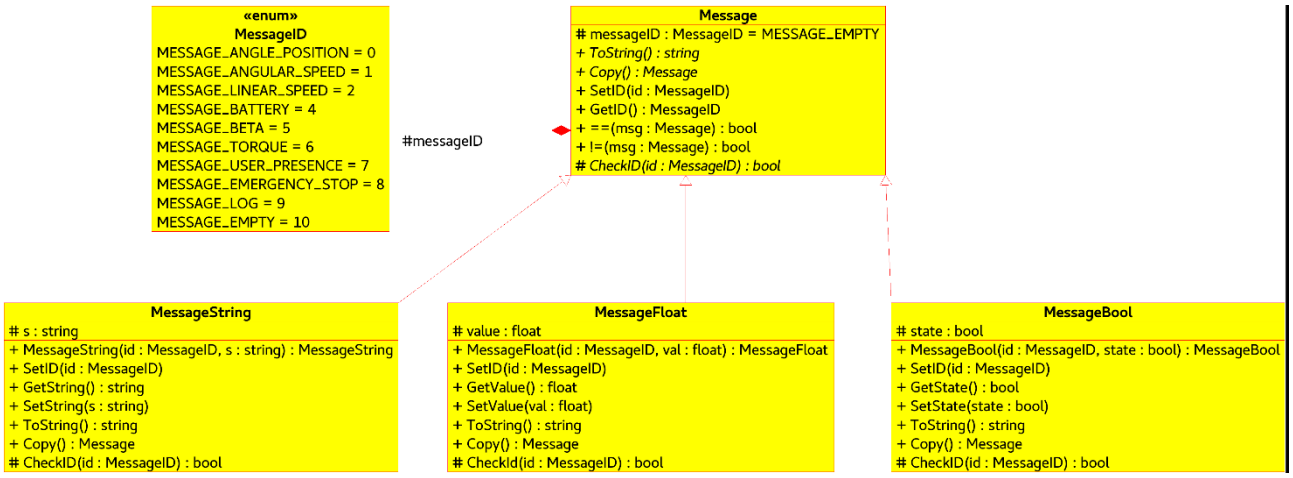


Figure 3 : Classes de messages

2.3 Classes de communication et de trace

La classe ComGui offre une méthode Write permettant d'envoyer un message vers le moniteur, message contenant soit un paramètre (angle, batterie, présence utilisateur, …), soit une information sur l'état d'un mutex, sémaphore ou tâche.


ComGui
socketFD : int = -1
clientFD : int = -1
+ Open(port : int) : int
+ Close()
+ AcceptClient() : int
+ Write(msg : Message)
+ Write_Pre()
+ Write_Post()
ComStm32
fd : int
+ Open() : int
+ Close() : int
+ GetComState() : bool
+ Read() : Message
+ Write(msg : Message) : int
+ Read_Pre()
+ Read_Post()
+ Write_Pre()
+ Write_Post()
# CharToFloat(bytes : unsigned char) : float
# CharToBool(bytes : unsigned char) : bool
# CharToInt(bytes : unsigned char) : unsigned int
# CharToMessage(bytes : unsigned char) : Message
# MessageToChar(msg : Message, inout buffer : unsigned char)
Trace
- beginTime : int
+ GetTimeUs() : int
+ GetTimeMs() : int
+ WaitForMutex(mut : RT_MUTEX) : Message
+ MutexAcquired(mut : RT_MUTEX) : Message
+ MutexReleased(mut : RT_MUTEX) : RT_MUTEX
+ WaitForSem(sem : RT_SEM) : Message
+ SemEntered(sem : RT_SEM) : Message
+ SemSignaled(sem : RT_SEM) : RT_SEM
+ TaskEntered() : Message
+ TaskNewIteration() : Message
+ TaskEnded() : Message
+ TaskDeleted(task : RT_TASK) : Message
- MutexGeneric(mut : RT_MUTEX, event : string) : Message
- SemGeneric(sem : RT_SEM, event : string) : Message
- TaskGeneric(task : RT_TASK, event : string) : Message

3. RAPPELS DE C++ ET PROGRAMMATION OBJET

Le C++ est un langage de programmation, dérivé du langage C, permettant une programmation orientée objet, basé sur le principe de classe (d'autres types de programmation objet existent). Les spécifications du langage continuent aujourd'hui encore d'évoluer pour lui rajouter des fonctionnalités.

En ce qui concerne les langages objets basés sur des classes, l'élément principal est la classe, sorte de patron (template) servant à décrire l'architecture des objets. De ce fait, on dit qu'un objet est une instance d'une classe, ou instancie une classe, autrement dit qu'il donne une existence réelle à une classe, notamment en lui affectant une zone en mémoire.

Ainsi, quand on écrit :

```
class MaClasse {                // Définition de la classe MaClasse
public:
  int var;                      // var est un attribut de MaClasse (variable)
  MaClasse() {var=0;}           // Constructeur, appelé lors de la création d'un                    // objet

  void Add(int x) {var=var+x;} // Méthode (fonction)
};

void main(void) {
  MaClasse maClasse;            // Création de l'objet maClasse à partir de la
                                // classe MaClasse (instanciation) => var vaut 0

  maClasse.Add(2);              // Ajoute 2 à maClasse.var => var vaut 2
}
```

maClasse est un objet basé sur MaClasse. maClasse est donc une instance de la classe MaClasse. On dit que maClasse est de type MaClasse. A partir de là, on peut accéder à ses attributs (variables) et méthodes (fonctions) publiques.

Il y a 2 façons de créer un objet et, du coup, 2 façons d'accéder à ses attributs et méthodes : soit déclarer directement l'objet, soit déclarer un pointeur sur l'objet. Dans le premier cas, on accédera aux éléments de l'objet par un point '.', dans l'autre cas, il faudra, avant de l'utiliser, lui allouer de la mémoire (via le mot clef 'new') et on accédera à ses éléments par le biais d'un flèche '→'

```
void main() {
  MaClasse monObjet;    // Déclaration d'un objet de type MaClasse
  MaClasse *ptrSurObjet; // Déclaration d'un pointeur sur un objet de
                         // type MaClasse

  monObjet.var=0;       // Accès à l'attribut var de monObjet
  ptrSurObjet = new MaClasse(); // Création d'un objet de type
                         // MaClasse et affectation du
                         // pointeur ptrSurObjet

  ptrSurObjet->var=0;           // Accès à l'attribut var de l'objet
                         // pointé par ptrSurObjet
}
```

3.1 Définition et déclaration

Une classe se compose d'une partie déclaration (qui se trouve dans un fichier .h) et d'une partie définition (qui se trouve dans un fichier .cpp).

La déclaration utilise le mot clef 'class'. Les éléments constitutifs de la classe sont rangés selon leur accessibilité : publique, privée ou protégée. La forme classique d'une déclaration est la suivante :

```
class MaClasse {
public:
   MaClasse();
   int UneMethode(int i);

private:
   int unAttributPrivee;
};
```

Pour les méthodes les plus triviales, on peut directement ajouter le corps de la méthode (sa définition) lors de sa déclaration dans le .h (par exemple ici pour la méthode UneMethode)

```
class MaClasse {
public:
   MaClasse();
   int UneMethode(int i) { unAttributPrivee=unAttributPrivee +i; }

private:
   int unAttributPrivee;
};
```

Pour les cas plus complexe, on sépare bien la déclaration et la définition. La définition des méthodes se trouve du coup dans le fichier .cpp et prend la forme suivante :

- dans le fichier maclasse.h

```
class MaClasse {
public:
   MaClasse();
   int UneMethode(int i);

private:
   int unAttributPrivee;
};
```

- dans le fichier maclasse.cpp

```
#include «maclasse.h»
MaClasse::MaClasse() {
   unAttributPrivee=0;
}
MaClasse::UneMethode(int i) {
   unAttributPrivee=unAttributPrivee +i;
}
```

Notez dans ce cas que l'on indique à quelle classe appartient la méthode (ou le constructeur) en rajoutant le nom de la classe avant le nom de la méthode séparé '::'

3.2 Visibilité

Avec les langages objets, on peut choisir si les éléments d'un objet sont visibles par l'appelant ou pas. Dans le cas du C++, on a le choix entre 3 niveaux : publique, privé ou protégé. Un élément publique est toujours directement accessible (via '.' ou '->') par l'appelant. Un élément privé est toujours directement inaccessible par l'appelant, seules les méthodes de l'objet y ont accès.

Le cas de protégé ('protected') est plus subtil. Par rapport à l'appelant, c'est comme un élément privé, inaccessible directement. Par contre, dans le cas d'une classe dérivée (héritage) d'une autre, les éléments protégés sont accessibles, tndis que les éléments privés reste privés, même pour une classe dérivée. La différence entre privé et protégé prend son sens dans l'héritage de classe.

Le tableau si dessous résume les visibilités de chaque mot-clefs

	Public Members	Protected Members	Private Members
Public Inheritance	public	protected	private
Protected inheritance	protected	protected	private
Private inheritance	private	private	private

Tableau 3: Rappel C++ - Visibilité des membres d'une classe

3.3 Méthodes statiques

Une méthode d'une classe peut être déclarée statique (mot-clef 'static'). Dans ce cas, plutôt que d'être une méthode appartenant à l'objet instancié (et n'existant qu'à ce moment là), la méthode est utilisable telle qu'elle, sans avoir besoin préalablement d'instancier un objet pour y accéder. Et inversement, la méthode ne fait pas partie des méthodes appartenant à un objet. Elle a une existence propre, indépendante des objets.

```
class MaClasse {
public:
 MaClasse();
 int UneMethode(int i);
 static int UneMethodeStatique(int i);

private:
 int unAttributPrivee;
};

void main() {
 MaClasse monObjet;

 monObjet.UneMethode(5);        // Ok, ca marche
 monObjet.UneMethodeStatique(3);  // Ne compile pas: UneMethodeStatique
 // n'appartient pas à l'objet monObjet
 MaClasse::UneMethodeStatique(3); // Ok, ca marche
}
```

Étant donné que les méthodes statiques appartiennent à une classe mais pas aux objets qui l'instancient, les méthodes statiques ne peuvent pas accéder aux éléments non statiques d'une classe (méthodes ou attributs). Elles doivent se contenter des paramètres qui leur sont passé.

3.4 Méthodes virtuelles et polymorphisme

Une méthode peut être déclarée virtuelle ('virtual'). Elle existe belle et bien, mais l'intérêt de ce qualificateur est de permettre le polymorphisme lors de l'exécution du programme. Pour cela, il faut que les objets soient manipulés via un pointeur ou une référence sur le type de base.

Lorsqu'une classe dérive d'une classe de base, les méthodes virtuelles qui seront surchargées par cette classe fille pourront, à l'exécution, être correctement appelées, même si le type du pointeur qui pointe sur l'objet est du type de la classe de base.

Concrètement si l'on considère les classes ClasseDeBase et ClasseDerive décrites si dessous

```
#include <iostream>

using namespace std;

class ClasseDeBase {
public:
    ClasseDeBase() { compteur=0; }
    void Add(int i) { compteur+=i; }
    virtual void ToString() { cout<<"ClasseDeBase: "<<compteur<<endl; }

protected:
    int compteur;
};

class ClasseDerive : public ClasseDeBase {
public:
ClasseDerive() {compteurSpecifique=0;}
void Add(int i) { compteurSpecifique+=i; }
void ToString() {
    cout<<"ClasseDerive: "<<compteurSpecifique<<" Compteur de base=" <<
                compteur<<endl; }

protected:
    int compteurSpecifique;
};
int main(int argc, char** argv) {
    ClasseDeBase base;
    ClasseDeBase *ptrBase;
    ClasseDerive derive;

    ptrBase = new ClasseDerive();

    base.Add(1);
    derive.Add(2);
    ptrBase->Add(3);

    base.ToString();
    derive.ToString();
    ptrBase->ToString();
    return 0;
}
```

On voit que les deux classes possèdent une méthode Add et une méthode ToString. ClassDerive surcharge les deux méthodes, mais ToString est une méthode virtuelle dans la classe de base, pas Add.

Dans le main, on affecte à ptrBase (qui est de type pointeur sur un objet de type ClasseDeBase) un pointeur de type ClasseDerive. On a le droit vu que ClasseDerive possède ClasseDeBase comme classe de base.

Lorsque l'on appelle les méthodes Add, vu qu'elles ne sont pas virtuelles, ce sont les méthodes Add de leurs classes respective qui sont appelées (dans le cas de ptrBase, c'est la méthode Add de la classe ClasseDeBase qui est appelé, ClasseDebase étant le type du pointeur). Comme Add n'est pas virtuelle, on ne remonte pas à la classe à l'origine du pointeur.

Lorsque l'on appelle les méthodes ToString, qui sont virtuelles, c'est bien la méthode de la classe à l'origine du pointeur qui sera appelé, peu importe le type du pointeur. Du coup, dans le cas de ptrBase->ToString(), c'est bien la méthode de la classe ClasseDerive qui sera appelé, pas celle de ClasseDeBase, pourtant type du pointeur.

Le résultat à l'écran sera celui-ci :

```
ClasseDeBase: 1                ← resultat de base.ToString();
ClasseDerive: 2 Compteur de base=0  ← resultat de derive.ToString();
ClasseDerive: 0 Compteur de base=3  ← resultat de ptrBase→ToString();
```

Ce mécanisme est très largement utilisé dans la classe Message et ses dérivées.

4. COMMUNICATION SERIE (STM32)

4.1 Envoi de données

L'envoi de données (par le STM32) est effectué dans la gestion d'interruption externe de l'accéléromètre, qui déclenche à 94 Hz. Afin de simplifier les envois, toutes les données sont envoyées dans une seule trame, qui mesure 37 octets en incluant les caractères de contrôle.

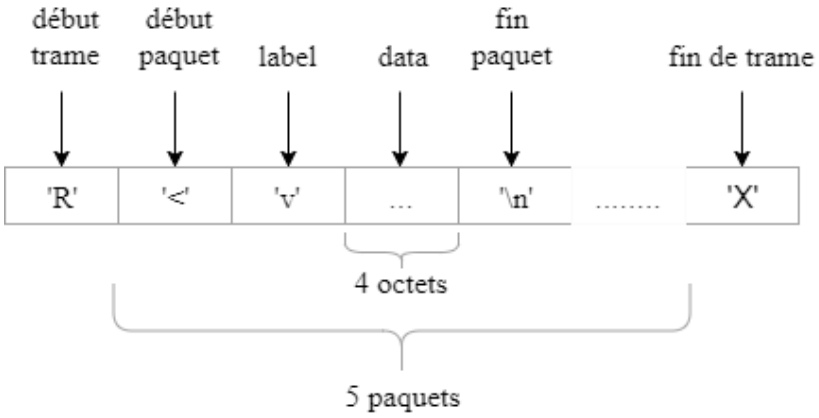
4.2 Envoi des données vers le STM32

Le superviseur envoie le couple moteur et l'ordre d'arrêt d'urgence au STM32.

Donnée	Type	Unité
Consigne de couple	float	N.m
Arrêt	int	1 si arrêt d'urgence, 0 sinon

Tableau 4: Communication STM32 - Liste des données envoyées au STM32

L'envoi de données de STM32 se fait grâce à l'envoi de trames par la méthode de division du nombre flottant en 4 octets sur le port série. Le décodage demande moins de calcul que pour des trames en ASCII. Pour éviter que les octets puissent prendre n'importe quelle valeur, les octets des données sont entourés par des caractères de contrôle.



Ces trames sont composées des champs suivants:

• *début de trame*: contient le caractère 'R', permet de reconnaître le début d'une trame

• *fin de trame*: contient le caractère 'X', permet de reconnaître le fin d'une trame

• *paquet de donnée:*

- *début paquet*: contient le caractère '<' qui indique le début d'un paquet
- *label*: contient un caractère qui permet d'identifier la grandeur associée aux informations du champ data
- *data* : contient l'information envoyée
- *fin paquet*: contient le caractère '\n' qui indique la fin d'un paquet

- *fin de trame*: contient le caractère 'X', permet de reconnaître la fin de la trame

4.3 Réception de données

Les données sont reçues sous le format d'un paquet (7 octets), ce qui diffère du cas précédent. La détection d'un paquet complet se fait au début et à la fin de paquet (caractères '<' et '\n'). Ce choix vient du fait que l'envoi de consignes de Raspberry Pi se fait à la vitesse de 100 Hz, le temps de traitement des messages est donc largement suffisamment. De plus, comme il existe d'autres tâches temps réel qui peuvent envoyer une trame d'urgence pour arrêter le gyropode, il est nécessaire de recevoir les informations paquet par paquet. Au niveau du STM32, la consigne de couple reçue du Raspberry Pi est convertie directement en courant (A). La conversion est faite en divisant par la valeur 0.80435, qui est le produit de K_m (le constant de couple de moteur DC(Nm/A)) et le rapport de réduction de moteur à la roueK_g.

4.4 Liste des labels utilisés dans les trames

Donnée	Type	Unité	Label
position angulaire	float	rad	'p'
vitesse angulaire	float	rad /s	's'
niveau batterie	integer	%	'b'
vitesse linéaire	float	m/s	'v'
présence utilisateur	integer	1 si présent, 0 sinon	'u'
Consigne de couple	float	N.m	'c'
Arrêt	int	1 si arrêt d'urgence, 0 sinon	'a'

Tableau 5: Communication STM32 - Liste des données reçues du STM32

5. MISE EN PLACE D'UNE TRACE

Pour mieux comprendre l'exécution des tâches dans le programme de supervision, une classe Trace permet de générer des messages relatifs à l'état d'élément de synchronisation, qui peuvent être alors affichés sur la console (terminal) du poste de travail ou dans la fenêtre de log du moniteur.

Elle permet le suivi des éléments de synchronisation (mutex, sémaphore, taches) du programme. La classe fournit des méthodes permettant d'étiqueter une attente, ou la prise (d'un sémaphore, d'un mutex) et de tagger cette étiquette avec l'heure depuis le lancement de la trace.

Les méthodes renvoient toutes un message, qui doit être ensuite envoyé vers l'interface graphique. Peu importe si ces messages ne sont pas envoyés immédiatement, ou par salve, le fait qu'ils soient horodatés suffit.

Ces messages n'apparaissent pas directement sur l'IHM, mais se rajoutent dans la fenêtre de log, pour une analyse après coup.

La classe Trace fourni les méthodes suivantes :

Nom de la méthode	Rôle	A utiliser
Message* WaitForMutex(RT_MUTEX* mut);	Indique que l'on attend la prise du mutex mut	Avant la prise du mutex
Message* MutexAcquired(RT_MUTEX* mut);	Indique que l'on a passé la prise du mutex mut	Après la prise du mutex
Message* MutexReleased(RT_MUTEX* mut);	Indique que l'on vient de libérer le mutex mut	Après la libération du mutex
Message* WaitForSem(RT_SEM* sem);	Indique que l'on attend la prise du sémaphore sem	Avant la prise du sémaphore
Message* SemEntered(RT_SEM* sem);	Indique que l'on a passé la prise du sémaphore sem	Après la prise du sémaphore
Message* SemSignaled(RT_SEM* sem);	Indique que l'on vient de libérer le sémaphore sem	Après la libération du sémaphore
Message* TaskEntered();	Indique que l'on vient de lancer la tâche courante	Au début de la tâche
Message* TaskNewIteration();	Indique que la tache periodique vient de réitérer	Au début de la boucle de tâche
Message* TaskEnded();	Indique que l'a tâche courante va se terminer	A la fin de la tâche
Message* TaskDeleted(RT_TASK* task);	Permet de savoir si la tâche task est détruite	N'importe où, ailleurs que dans la tâche task

Tableau 6: Trace - Liste des méthodes

6. PRISE EN MAIN DE NETBEANS

Netbeans est un environnement de développement intégré (IDE), conçu initialement pour développer des programmes écrits en Java. Mais il se prête bien à la programmation en C++ et surtout, possède une fonction de compilation à distance permettant, dans notre cas, de compiler le code directement sur la Raspberry.

Le lancement de l'environnement se fait en ligne de commande (dans un terminal) en tapant :

```
netbeans
```

L'IDE apparaît alors comme suit

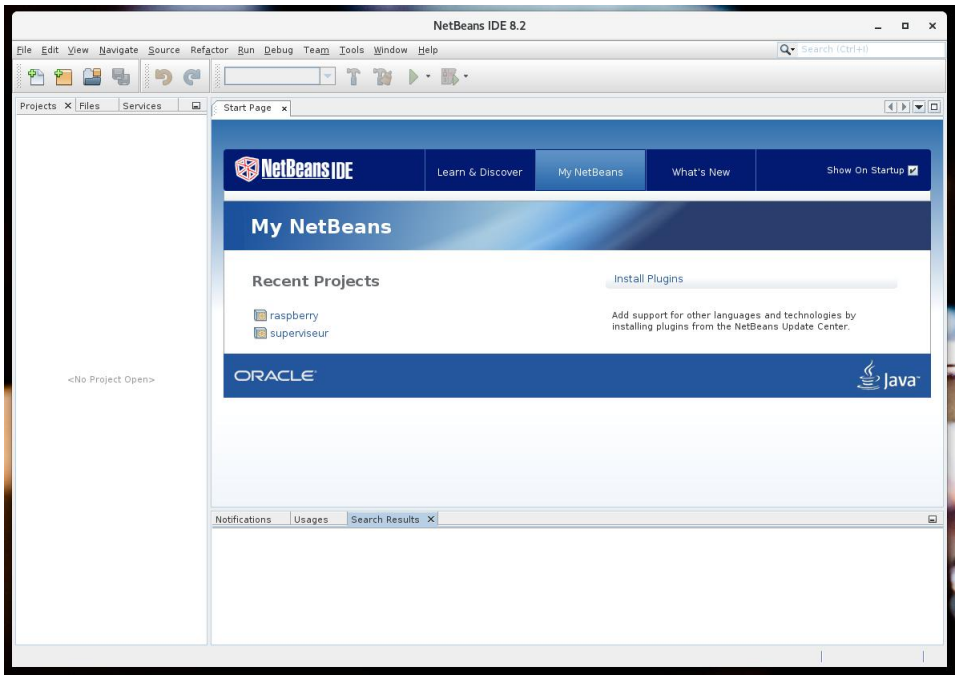


Figure 4:Netbeans - Écran principal de Netbeans

Sur les nouvelles versions de Netbeans, le support des projets compilable sur serveurs distants n'est plus intégré de base dans Netbeans. Il faut donc rajouter un plugin manuellement avant d'ouvrir votre projet (à ne faire que la première fois).

Pour cela, allez dans le menu « Tools → Plugins » et dans la fenêtre qui s'ouvre, sélectionnez l'onglet « Settings ». Dans cet onglet, cliquez (activez) Netbeans 8.2 Plugin Portal.

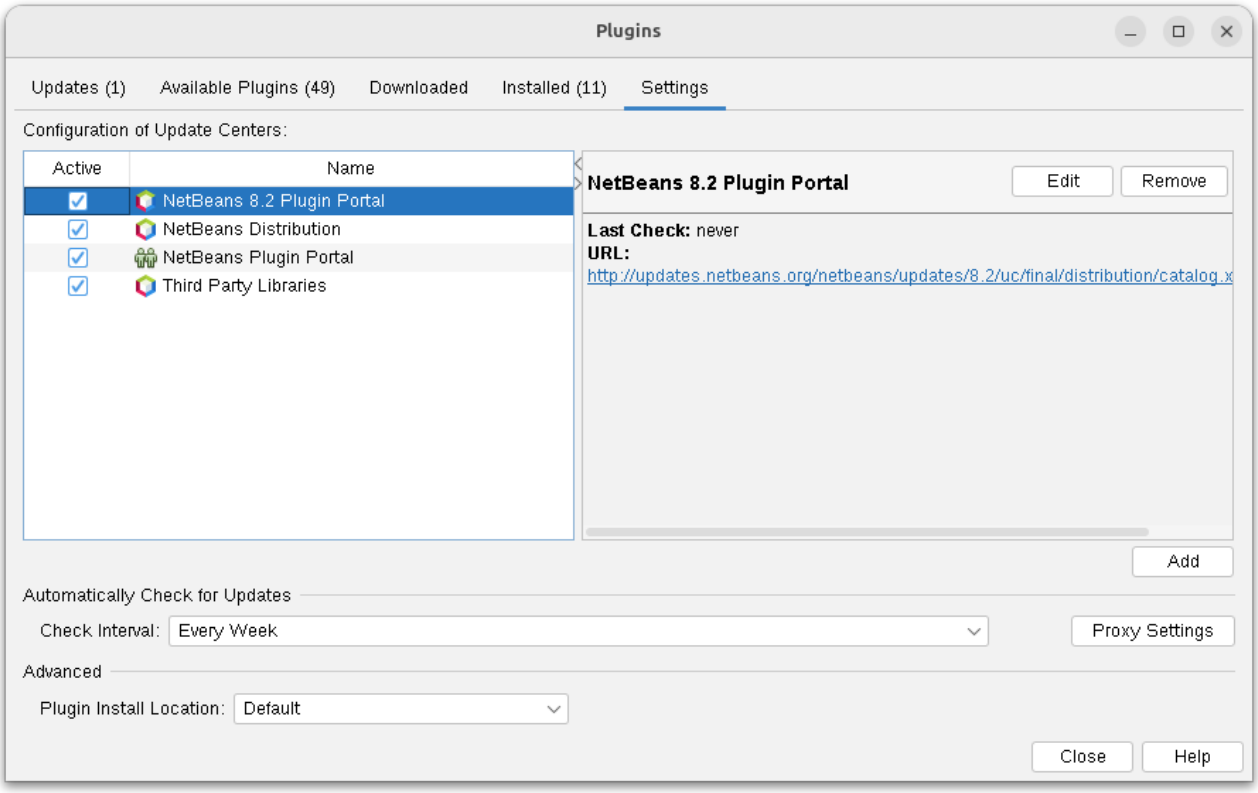


Figure 5: Netbeans - Ajout de la source de plugin Netbeans 8.2

Ensuite, sélectionnez l'onglet « Available Plugins » et cliquez (installez) le plugin « C/C++ » de la source « Certified Plugin » puis validez en cliquant en bas sur « Install ».

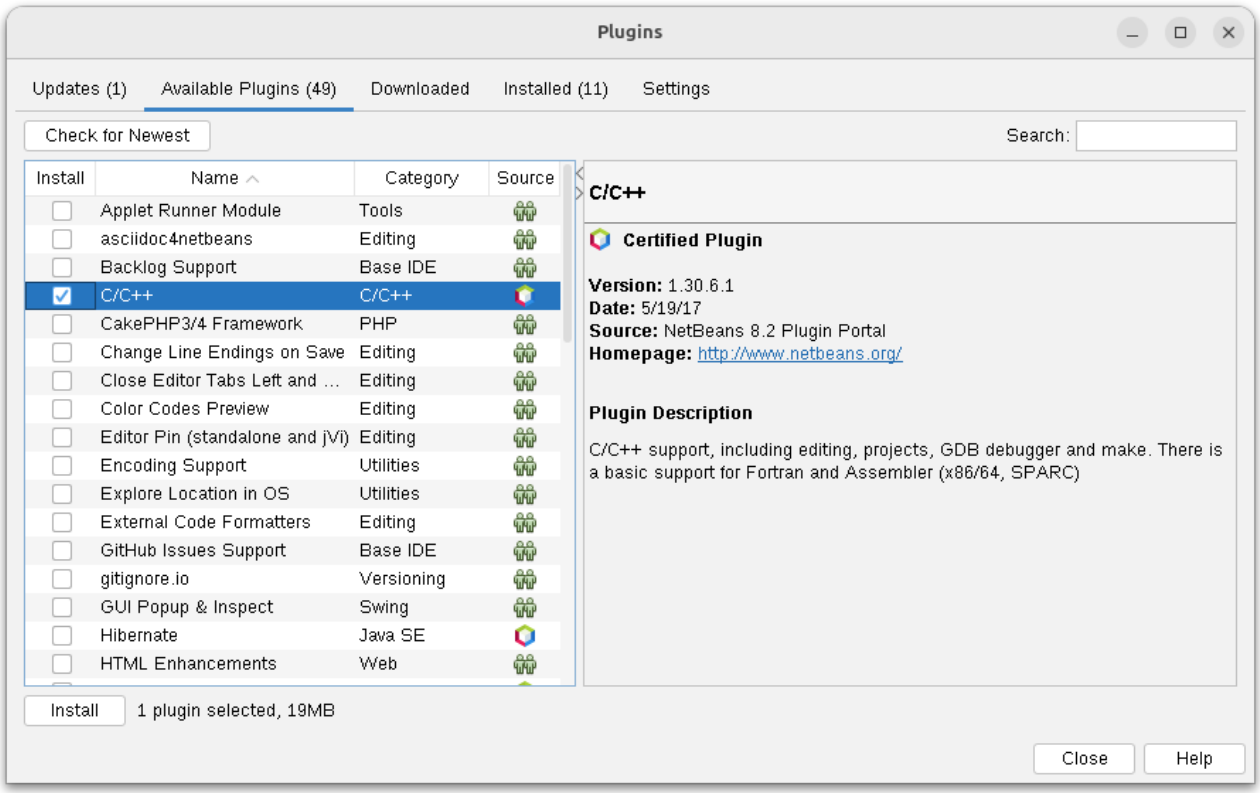


Figure 6:Netbeans - Ajout du plugin C/C++

Acceptez les licences, et continuez.

Si le programme vous dit qu'il ne trouve pas le fichier « unpack200 », vous pouvez en trouver une version dans « /usr/local/insa/src/stm32cube/jre/bin).

Ensuite, le programme d'installation demandera de vérifier les certificats. Validez et continuez

Ensuite, pour ouvrir un projet, allez dans « File / Open Project » et sélectionnez le projet de votre choix (dans notre cas, il se trouve dans le dépôt git dans segway/raspberry). La fenêtre suivante s'ouvre :

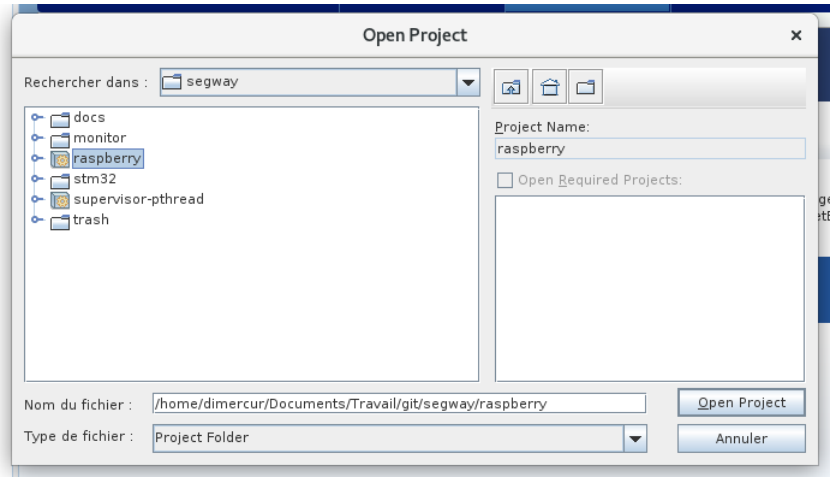


Figure : Netbeans - Dialogue "Open Project"

Cliquez sur « Open Project » et l'environnement se charge avec le projet :

En cliquant (comme fait ici) en face de « Header Files » et de « Source Files », les fichiers constituant votre projet sont accessibles. Double cliquez dessus pour les ouvrir dans l'environnement.

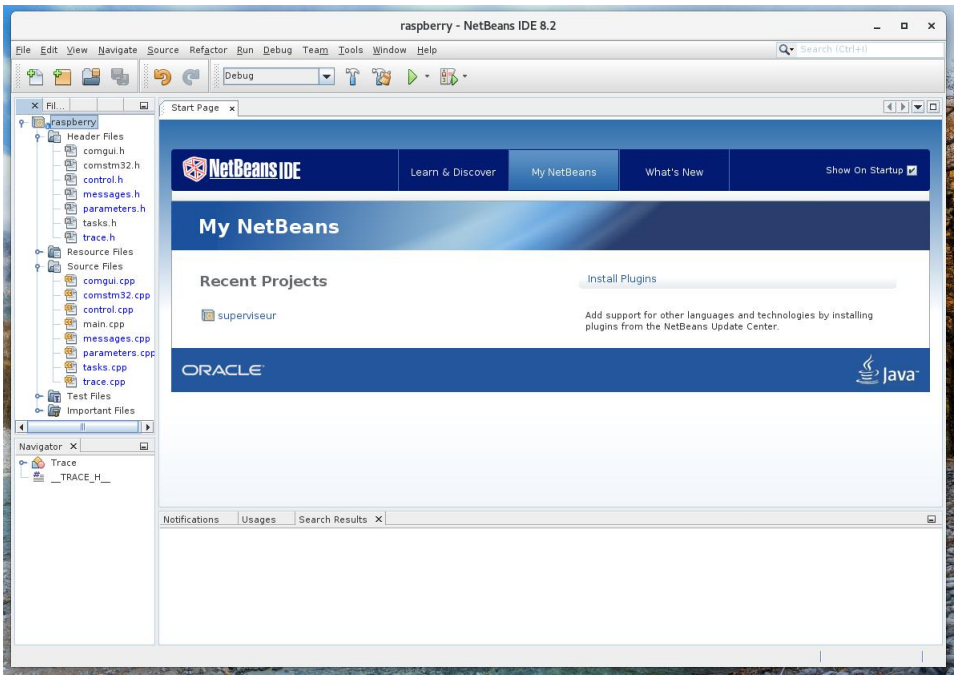


Figure 7: Netbeans - IDE avec le projet chargé

Pour pouvoir compiler sur Raspberry, il faut lui rajouter une cible de compilation. Dans le bandeau de gauche (ou se trouve l'arborescence de votre projet) se trouve en haut des onglets. Le troisième se nomme « Services » (agrandissez le bandeau) et contient un champ nommé « C/C++ Build Hosts ». Ouvrez cette entrée :

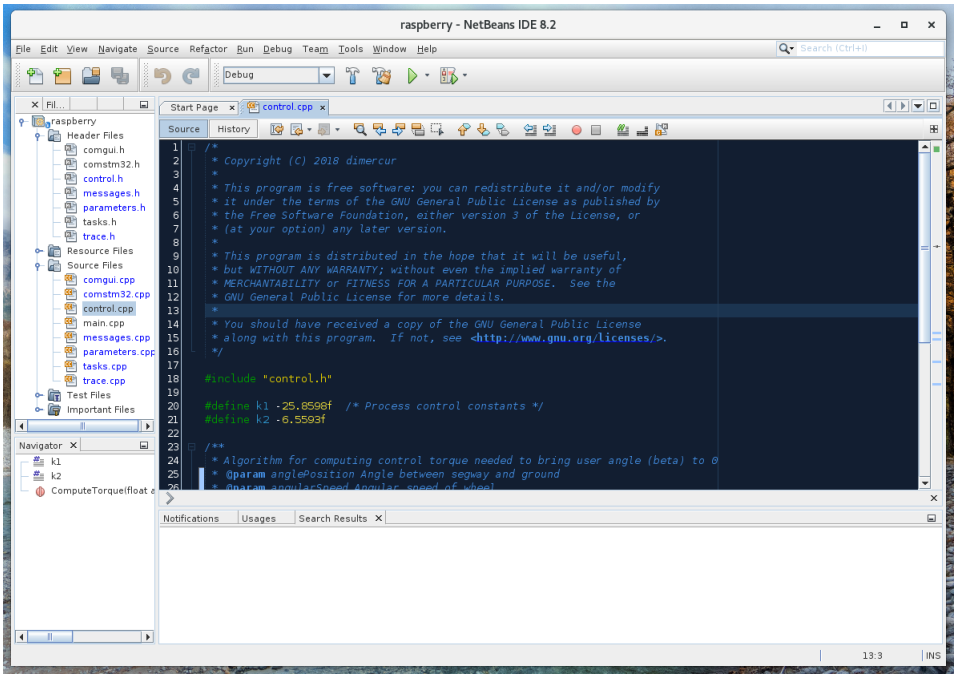


Figure 9: Netbeans - Affichage de fichier dans l'IDE

« localhost » correspond à votre machine de TP : c'est la cible par défaut . Les autres présentes dans cet exemple sont des Raspberry déjà rajoutées. Pour en rajouter une de plus (votre simulateur), cliquez sur « C/C++ Build Hosts » avec le bouton droit : le menu « Add new host » apparaît :

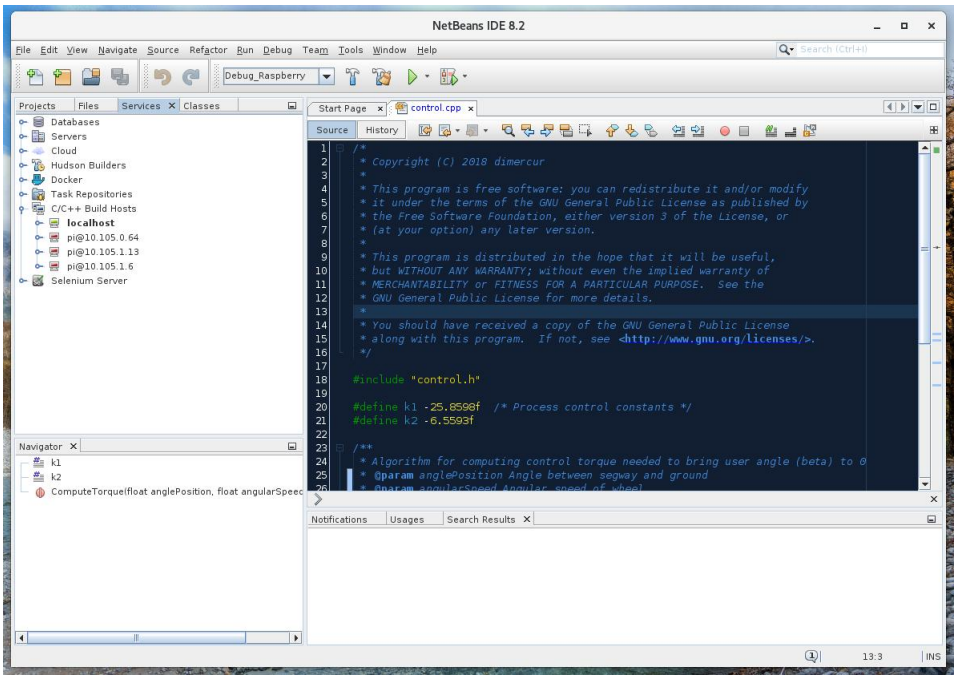


Figure 11: Netbeans - Affichage des serveurs de compilation enregistrés

Cliquez dessus (bouton gauche) et la fenêtre suivante s'ouvre :

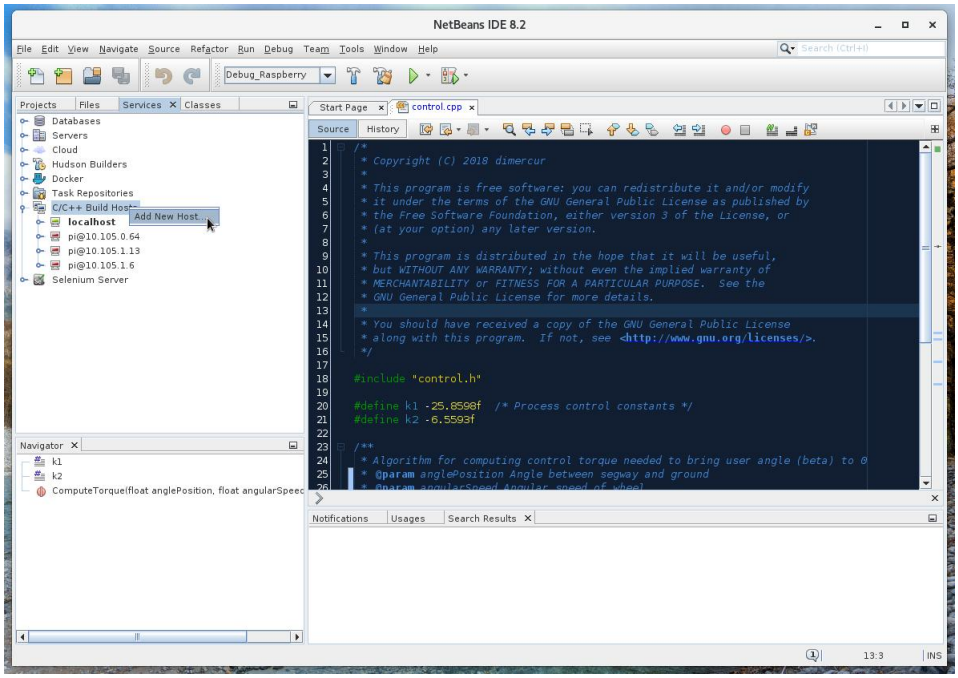


Figure 13:Netbeans - Ajout d'un serveur de compilation

Dans le champ « Hostname » (en dessous de « Select Host ») saisissez l'adresse IP de votre simulateur (étiquette collée sur la Raspberry » puis cliquez sur le bouton « Next » en bas de la fenêtre.

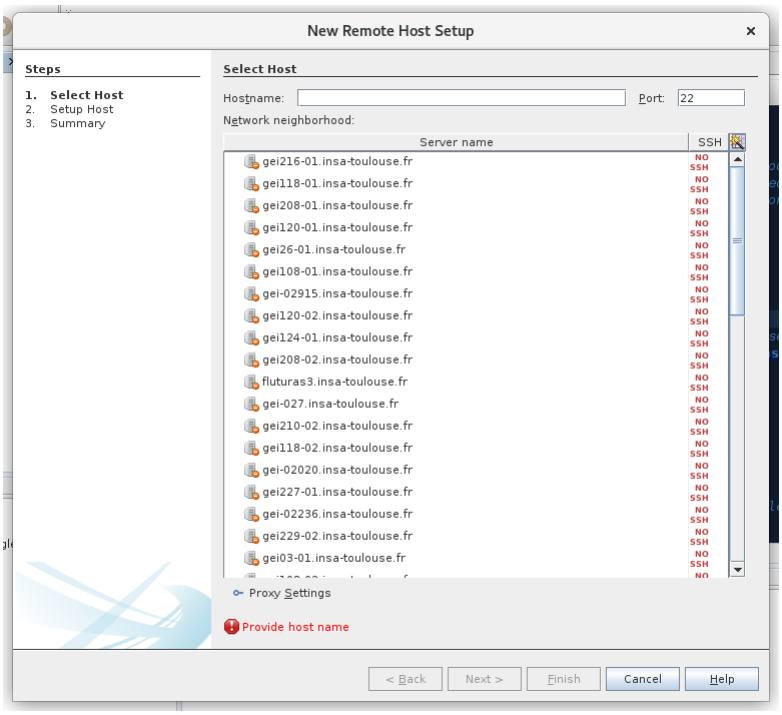


Figure : Netbeans - Choix d'un serveur de compilation

Si la cible est démarrée et accessible, vous devriez obtenir ceci :

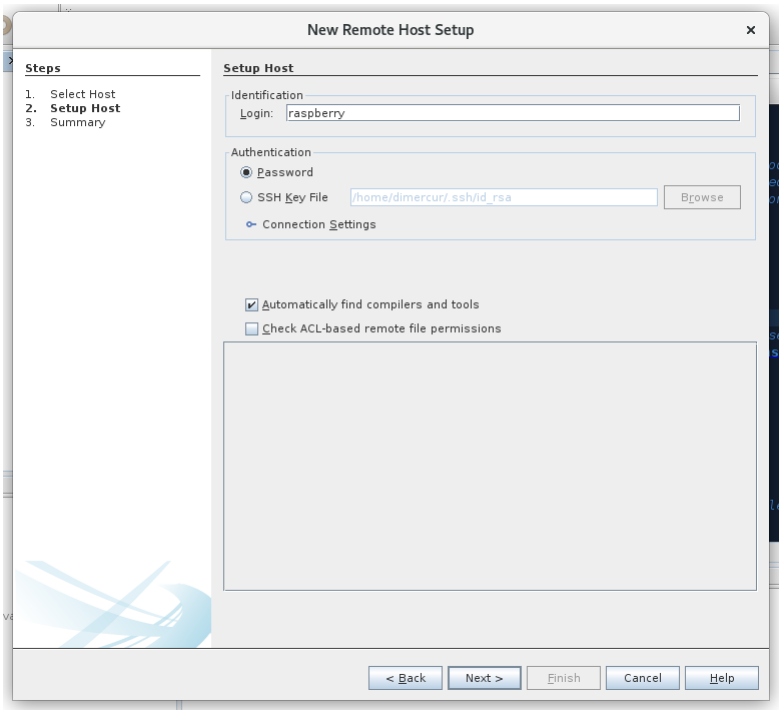


Figure 14: Netbeans - Configuration du serveur - Identifiant

Changez le login par « xenomai» dans le champ identification (et pas pi comme sur l'image). Netbeans demandera l'ouverture d'un portefeuille kdewallet pour mémoriser le mot de passe de connections : indiquez ici le mot de passe de votre compte INSA. Une fenêtre d'authentification sur la cible distante s'ouvre alors . Elle a cette forme là

Le mot de passe est «xenomai». Cliquez ensuite sur OK.

Figure 15:Netbeans - Configuration du serveur - Mot de passe

La fenêtre de configuration de la cible distante cherche alors le compilateur et change comme ceci :

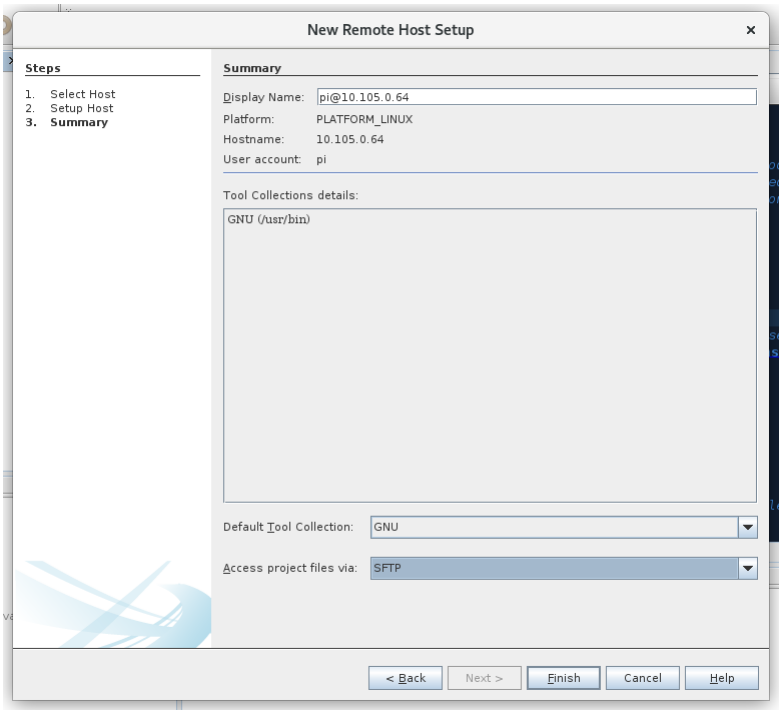


Figure 16: Netbeans - Configuration du serveur - Mode d'accès

Sélectionnez alors « SFTP » dans le champ « Access project file via : » puis cliquez sur le bouton « Finish ». Votre cible est ajoutée !

Il faut ensuite configurer la cible « Debug_Raspberry » de votre projet. Pour cela, cliquez sur la liste déroulante (en dessous des menus « Run » et « Debug » et à côté du marteau bleu) et sélectionnez « Debug_Raspberry »

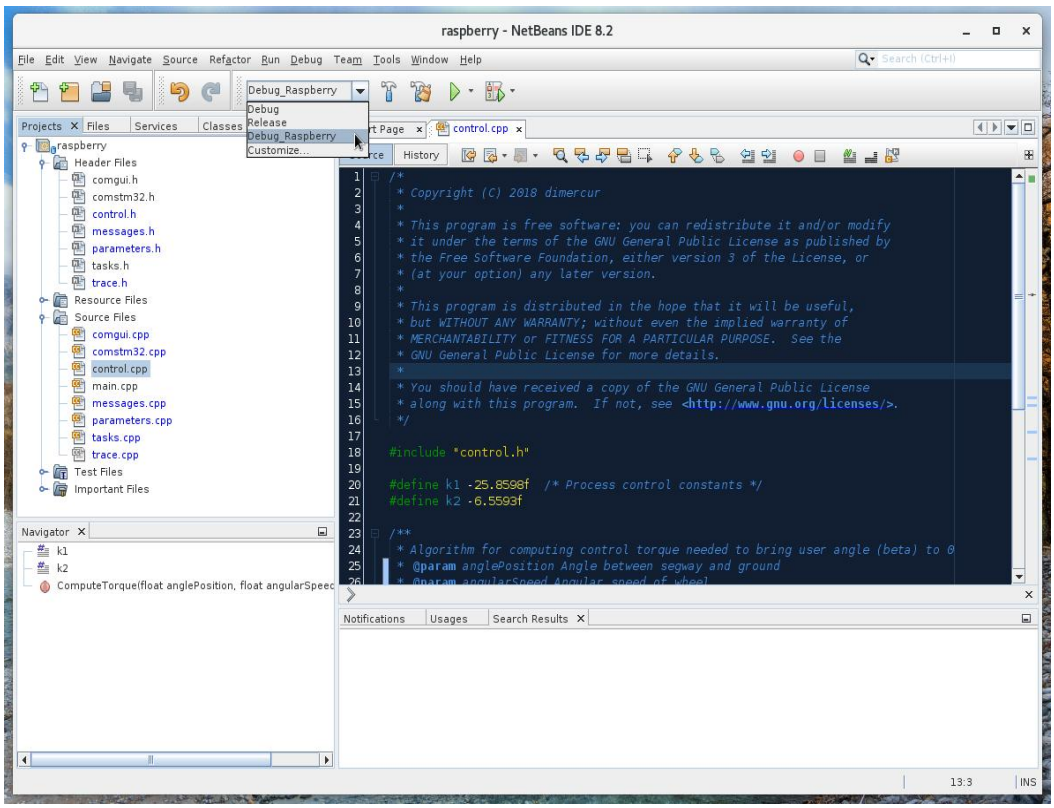


Figure 17: Netbeans - Sélection de la cible « Debug »

Ensuite, cliquez (bouton droit) sur « Raspberry » dans l'arborescence de votre projet (tout en haut) et dans le menu, choisissez « Properties »

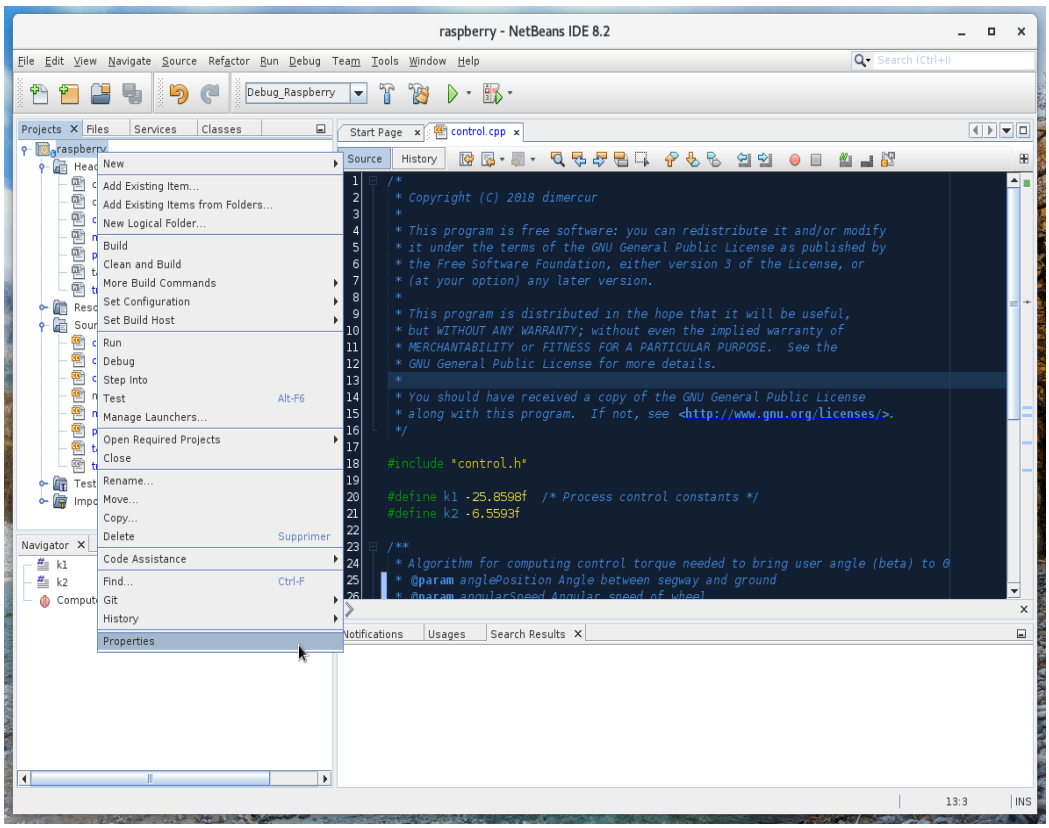


Figure 18:Netbeans - Configuration du projet

La fenêtre de propriétés du projet s'ouvre. Dans l'arborescence de gauche cliquez sur « Build »

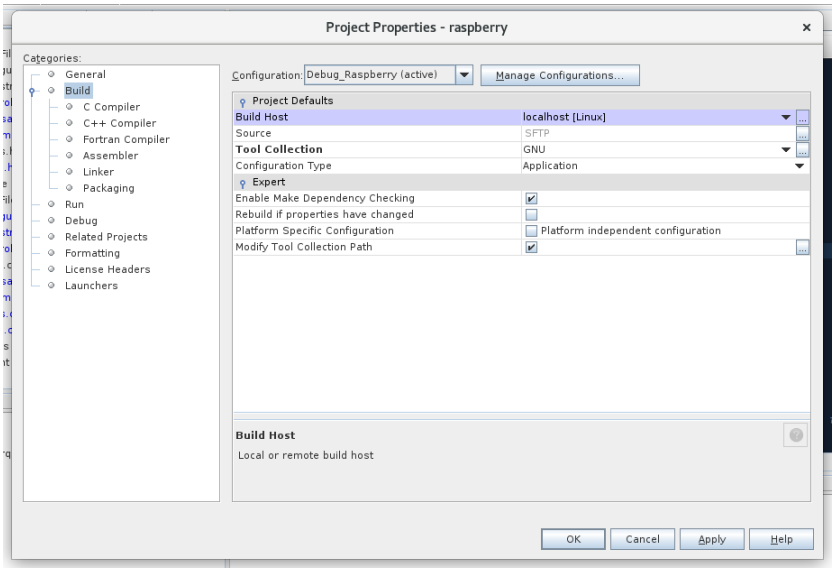


Figure 19: Netbeans - Configuration du projet - Choix du serveur de compilation

Sélectionnez la ligne (en violet ici) « Build Host » et déroulez la liste déroulante à droite (contenant initialement « localhost »). Choisissez votre cible distante et cliquez sur le bouton « Ok » en bas de la fenêtre.

Voilà, votre projet peut maintenant compiler sur votre cible distante.