Université Paris8

# gAgent
# Framework multi-agent en C++

*Auteur :*
Halim Djerroud
*hdd@ai.univ-paris8.fr*

18 mai 2018

# Table des matières

# 1 Introduction

**gAgent** est une framework écrit en C++ qui permet de développer un système multi-agent compatible FIPA [1] en C++ et en python.

La spécification FIPA

# 2 Architecture gAgent

Dans gAgent, chaque agent est un processus linux. Chaque agent (processus) est composé d'au moins deux threads :

— Listner : Ce thread est un thread détaché , il est implémenté sous forme de service, il permet de gérer les événements externes , des signaux Linux de `SIGRTMIN + 2` à `SIGRTMIN + 7`.

— Controler : Ce thread est un thread détaché aussi, il permet de gérer les verrous.

# 3 Premier pas

Le plus petit programme gAgent :

```
#include <gagent/AgentCore.hpp>
using namespace gagent;
int main(int argc, char* argv[]) {
        AgentCore::initAgentSystem();
        ...
        AgentCore::stopAgentSystem();
        return EXIT_SUCCESS;
}
```

`#include <gagentAgentCore.hpp>` : Inclure le moteur principale de gAgent.
`AgentCore::initAgentSystem()` : Permet d'initialiser **gAgent** (lire le fichier config).
`AgentCore::stopAgentSystem()` : Permet de libérer les ressources **gAgent**.

# 4 Créer un premier Agent

Les agents **gAgent** peuvent être dans d'un de ces états :

— Setup : À ce stade l'agent existe en mémoire mais sont état est inconnu (UNKNOWN). Pour activer l'agent il faut appeler la méthode `doInit()`

— Suspendu : La méthode `doSuspend()` permet de suspendre un agent qui est dans l'état Active, dans ce nouveau état (Suspendu) l'agent arrête tout exécution. Seule la méthode `doActivate()` permet le faire sortir de cet état pour revenir dans l'état active.

— En attente : La méthode `doWait()` permet de mettre l'agent en état d'attente d'un événement (généralement un message) puis il revient dans sont état active lors de la réception de l'événement ou-bien on peut le forcer son retour dans l'état active avec la méthode `doWake()`

— Active : Dans cet état l'agent est en cours d'exécution.

— Transit : La méthode `doMove()` permet la migration des agents d'une plateforme à une autre.

— Fin : La méthode `doDelete()` permet d'arrêter d'un agent en exécution et libérer les ressources.

```
#include <gagent/Agent.hpp>
#include <gagent/AgentCore.hpp>
using namespace gagent;

class myAgent: public Agent {
public:
  myAgent() :Agent() {};
  virtual ~myAgent() {}
  void setup() {
    ...
  }
};

int main(int argc, char* argv[]) {
    AgentCore::initAgentSystem();
    myAgent* g = new myAgent();
```

1. http://www.fipa.org

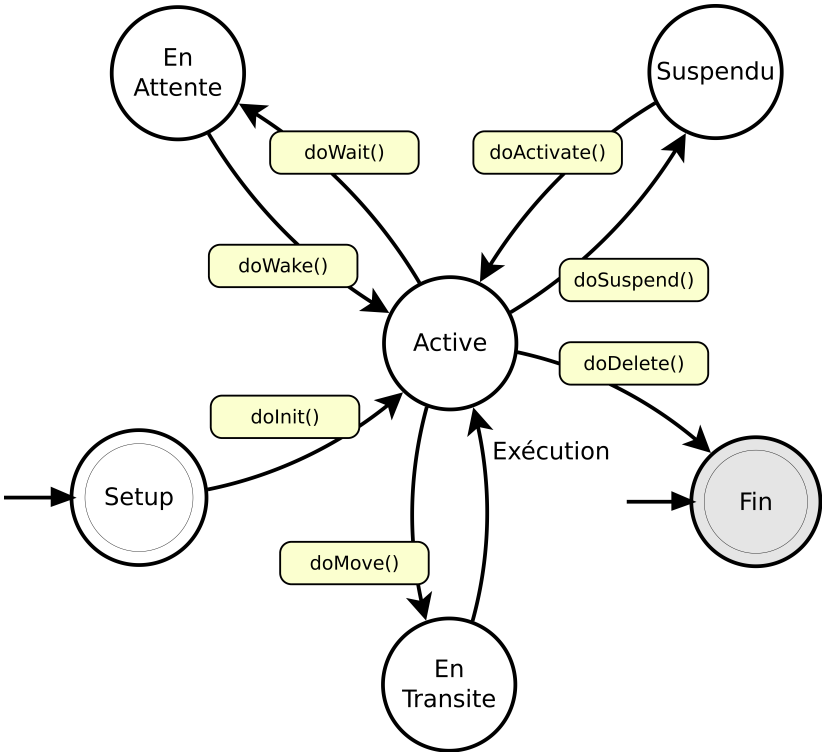


FIGURE 1 – Cycle de vie d'un agent gAgent.

```
        g->init();

        sleep(3);
        g->doWait();

        sleep(3);
        g->doActivate();

        sleep(3);
        g->doMove();

        sleep(3);
        g->doSuspend();

        sleep(3);
        g->doActivate();

        sleep(3);
        g->doWait();

        sleep(3);
        g->doWake();

        sleep(3);
        g->doDelete();

        delete g;

        AgentCore::stopAgentSystem();
        return EXIT_SUCCESS;
    }
```

Les méthodes :
— `setup()` : invoqué lors de l'initialisation de l'agent.
— `takedown()` : invoquée avant qu'un agent ne quitte la plateforme.

# 5 Les comportements d'un agent gAgent

Pour qu'un agent exécute une tâche, ou plusieurs, if faut auparavant définir ces tâches. Les tâches sont appelées **behaviours** (ou comportements en français) sont des instances de la classe **Behaviour** ou une des ses sous classes.

Pour qu'un agent exécute une tâche il doit lui l'attribuer un ou plusieurs **Behaviour** par la méthode `addBehaviour(Behaviour b)` de la classe **Agent** (voir l'exemple qui suit).

**gAgent** implémente chaque **Behaviour** dans un thread, si plusieurs **Behaviour** sont implémenté par l'agent, alors ces **Behaviours** sont exécutés en parallèles.

Chaque **Behaviour** doit obligatoirement implémenter les deux méthodes suivantes :

— `action()` : l'utilisateur doit implémenter les opérations à exécuter par le **Behaviour** ;

— `done()` : qui indique si le **Behaviour** en question a terminé son exécution ou pas.

Les deux autres méthodes suivantes, dont l'implémentation n'est pas obligatoire mais qui peuvent s'avérer utiles :

— `onStart()` : appelée juste avant l'exécution de ma méthode `action()` ;

— `onEnd()` : appelée juste après la retournement de *true* par la méthode `done()`.

Si besoin de savoir quel est le propriétaire d'un **Behaviour**, et cela peut être connu par le membre `this_agent` du **Behaviour** en question.

Exemple d'implémentation d'un Behaviour :

```
#include <iostream>
#include <stdio.h>

#include <gagent/Behaviour.hpp>
#include <gagent/Agent.hpp>
#include <gagent/AgentCore.hpp>

using namespace gagent;

class myCycle: public CyclicBehaviour {
public:
        myCycle(Agent* ag) :CyclicBehaviour(ag) {
        }

        void action() {
                std::cout << "Coin " << std::flush;
                sleep(1);
        }
};

class myAgent: public Agent {
public:
        myAgent() :Agent() {};

        virtual ~myAgent() {}

        void setup() {
                myCycle* bb = new myCycle(this);
                addBehaviour(bb);
        }
};

int main() {

        AgentCore::initAgentSystem();
        myAgent* g = new myAgent();
        g->init();

        AgentCore::syncAgentSystem();
        AgentCore::stopAgentSystem();

        return 0;
}
```

Résultat :

```
Coin Coin Coin Coin Coin Coin Coin
```

gAgent propose les Behaviours suivants :

— OneShotBehaviour
— SimpleBehaviour
— CompositeBehaviour
— CyclicBehaviour
— WakerBehaviour
— ParallelBehaviour
— SequentialBehaviour
— FSMBehaviour

## 5.1 Les Behaviours simples

C'est un Behaviour qui donner au
Ils sont composés de Behaviours : OneShotBehaviour, CyclicBehaviour.

```
...
class mySimpleBehavior: public SimpleBehaviour {
public:
        mySimpleBehavior(Agent* ag) : SimpleBehaviour(ag) {}

        unsigned int i = 0;

        void onStart(){
                std::cout << "Je␣sais␣compter␣de␣1␣a␣10␣"
                          << std::endl << std::flush;
                i=0;
        }

        void action() {
                i++;
                std::cout <<  i  << std::endl << std::flush;
        }

        bool done(){
                if (i==10){
                        std::cout << "J␣ai␣fini␣de␣compter␣:)␣."
                                  << std::endl << std::flush;
                        return true;
                }
                return false;
        }
};

class myAgent: public Agent {
public:
        myAgent() :Agent() {};

        virtual ~myAgent() {}

        void setup() {
                mySimpleBehavior* b1 = new mySimpleBehavior(this);
                addBehaviour(b1);

        }
};
```

Résultat :

```
Je sais compter de 1 a 10
1
2
3
4
5
6
7
8
9
10
J ai fini de compter :) .
```

### 5.1.1 OneShotBehaviour

Le Behaviour **OneShotBehaviour** permet d'exécuter une tâche une et une seule fois puis il se termine. La classe OneShotBehaviour implémente la méthode `done()` et elle retourne toujours *true*.

```
...
using namespace gagent;

class myBehavior: public OneShotBehaviour {
public:
        myBehavior(Agent* ag) : OneShotBehaviour(ag) {
        }

        void onStart() {
                std::cout << "␣--␣start␣--␣" << std::endl << std::flush;
        }

        void action() {
                std::cout << "Coin␣" << std::endl << std::flush;
        }

        void onEnd() {
                std::cout << "␣--␣end␣--␣"   << std::endl << std::flush;
        }
};

class myAgent: public Agent {
public:
        ...
        void setup() {
                myBehavior* bb = new myBehavior(this);
                addBehaviour(bb);
        }
};

int main() {
        ...
        myAgent* g = new myAgent();
        g->init();
        ...
}
```

Résultat :

```
-- start --
Coin
-- end --
```

Un exemple avec deux Behaviours :

```
...
class myBehavior1: public OneShotBehaviour {
public:
        myBehavior1(Agent* ag) : OneShotBehaviour(ag) {}
        void action() {
                std::cout << "Comportement␣1␣" << std::endl << std::flush;
        }
};

class myBehavior2: public OneShotBehaviour {
public:
        myBehavior2(Agent* ag) : OneShotBehaviour(ag) {}
        void action() {
                std::cout << "Comportement␣2␣" << std::endl << std::flush;
        }
};

class myAgent: public Agent {
public:
       ...
        void setup() {
                myBehavior1* b1 = new myBehavior1(this);
                addBehaviour(b1);

                myBehavior2* b2 = new myBehavior2(this);
```

```
                addBehaviour(b2);
            }
    };

    int main() {...
```

Résultat :

```
Comportement 1
Comportement 2
```

### 5.1.2 Cyclic Behaviour

Le **Cyclic Behaviour** permet d'exécuter la méthode **action()** de façon continue indéfiniment.

```
...
class myCycle: public CyclicBehaviour {
public:
        myCycle(Agent* ag) :CyclicBehaviour(ag) {
        }

        void action() {
                std::cout << "Coin " << std::flush;
                sleep(1);
        }
};

class myAgent: public Agent {
public:
        myAgent() :Agent() {};

        virtual ~myAgent() {}

        void setup() {
                myCycle* bb = new myCycle(this);
                addBehaviour(bb);
        }
};

int main() {

        AgentCore::initAgentSystem();
        myAgent* g = new myAgent();
        g->init();
        AgentCore::syncAgentSystem();
        AgentCore::stopAgentSystem();
        return 0;
}
```

Résultat :

```
Coin Coin Coin Coin Coin Coin Coin Coin Coin Coin ...
```

## 5.2 Les Behaviours planifiés

### 5.2.1 WakerBehaviour

Le **WakerBehaviour** permet d'exécuter la méthode `onWake()` après un délai passé en paramètre en millisecondes.

```
...
using namespace gagent;

class myWakerBehaviour: public WakerBehaviour {
public:
        myWakerBehaviour(Agent* ag) : WakerBehaviour(ag,50000) { }

        void onWake(){
```

```
                std::cout << "Termine " << std::endl << std::flush;
        }
};

class myAgent: public Agent {
public:
        myAgent() :
                        Agent() {
        };

        virtual ~myAgent() {
        }

        void setup() {
                myWakerBehaviour* b = new myWakerBehaviour(this);
                addBehaviour(b);
        }
};

int main() {
        ...
        g->init();
        ...
}
```

Résultat :

```
//apres une attente de 5 secondes.
Termine
```

### 5.2.2 TickerBehaviour

Le **TickerBehaviour** est implémenté pour qu'il exécute sa tâche périodiquement par la méthode onTick(). La durée de la période est passée comme argument au constructeur en millisecondes.

```
...
class myWakerBehaviour: public WakerBehaviour {
public:
        myWakerBehaviour(Agent* ag) : WakerBehaviour(ag,5000) { }

        void onWake(){
                std::cout << " Termine " << std::endl << std::flush;
                this_agent->doDelete();
        }

};

class myTickerBehaviour: public TickerBehaviour {
public:
        myTickerBehaviour(Agent* ag) : TickerBehaviour(ag,1000) { }

        void onTick(){
                std::cout << " tictac " << std::endl << std::flush;
        }
};

class myAgent: public Agent {
public:
        myAgent() :
                Agent() {
        };
        void setup() {
                myWakerBehaviour* b3 = new myWakerBehaviour(this);
                addBehaviour(b3);
                myTickerBehaviour* b4 = new myTickerBehaviour(this);
                addBehaviour(b4);
        }
};

int main() {

        AgentCore::initAgentSystem();
```

```
        myAgent* g = new myAgent();
        g->init();

        AgentCore::syncAgentSystem();
        AgentCore::stopAgentSystem();
        return 0;
}
```

Résultat :

```
tictac
tictac
tictac
tictac
tictac
Termine
```

# 6 La communication entre les agents

# 7 Le plateforme gAgent

# 8 Monitoring

**gAgent** propose un outil de monitoring sur une console. pour lancer le monitoring il faut appeler le programme `agentmonitor`. C'est un service qui permet de recevoir des messages des agents en utilisant le protocole UDP.

L'adresse IP et le PORT sont définis en paramètres sinon le programme contente de lire le fichier de configuration `config.cfg` dans lequel sont ces paramètres sont définis.

```
$agentmonitor --help
Allowed options:
--help                produce help message
--port arg            port
--ip arg              Ip adress
```

On peut tester cet outil avec la commande Unix `nc` comme suite `echo 'hello' | nc -u <ip> <port>`.
exemple :
`echo 'hello' | nc -u 127.0.0.1 40013`

```
000001 : hello
```

L'objectif de cet outil est permettre à l'utilisateur de surveiller le système durant le son fonctionnement. Les agent peuvent envoyer les messages (en C++) sur le monitor comme suite :

```
this_agent->sendMsgMonitor("␣Hello␣");
```

Résultat :

```
000001 : EbcrjZlp -> Start agent PID : 13029
000002 : EbcrjZlp ->  tictac
000003 : EbcrjZlp ->  tictac
000004 : EbcrjZlp ->  tictac
000005 : EbcrjZlp ->  tictac
000006 : EbcrjZlp ->  tictac
000007 : EbcrjZlp -> Stop agent PID : 13029
```