Object mARC\_Connector

Version 1.01

Date 14/01/2012

un connecteur se décompose en :

1 une socket cliente

2 un thread d’éxécution

3 un thread timer

4 un gestionnaire d’évènements

5 un gestionnaire de résultats en retour de serveur mARC

Connexion à un serveur mARC

Exemple :

marc = new mARC\_Connector ;

marc ->**Lock**(); //verrouillage optionnel si multi tâche

marc ->IP = “127.0.0.1”

marc ->Port = “1255”

marc ->**Connect**();

marc ->**UnLock**(); //déverrouillage

si la connexion a réussi

marc ->Connected() ; retourne **true**

l’évènement **mEvent\_onSocketConnected** est envoyé

si la connexion a échoué.

marc ->Connected() ; retourne **false**

l’évènement **mEvent\_onSocketDisconnected** est envoyé

Execution d’une commande mARC pas à pas

**Exemple 1 :**

marc = new mARC\_Connector ;

marc->**OpenScript** ();

marc->**Push** ("Contexts.Set");

marc->**Push** ("Knowledge");

marc->**Push** ("wikipedia");

marc->**AddFunction** ();

marc->ExecuteScript ();

une commande

Contexts.Set (<9 knowledge/>,<9 wikipedia/>);

Est générée puis exécutée par le serveur mARC connecté

marc->**Script** contient les lignes de commande envoyée

GPString \*s ;

**S** = (GPString \*) (\*marc->**Script)**[0] ;

**S** contient “Contexts.Set (<9 knowledge/>,<9 wikipedia/>);”

marc->Status.Error est positionné à **true** ou **false** selon la réussite de la commande

marc->State.**LastTime** contient la durée (en ms) de la dernière commande exécutée avec succès

A l’éxécution les évènements suivants peuvent être générés

mEvent\_onSocketDisconnected le connecteur n’est plus/pas connecté

mEvent\_onSocketError erreur TCP/IP ou socket

mEvent\_onSocketSending la socket envoie les données sur le réseau

mEvent\_onSocketSent la socket a terminé l’envoie des data

mEvent\_onMarcSending le connecteur prépare les data pou la socket

mEvent\_onMarcSent le connecteur a envoyé les données

mEvent\_onSocketReceiving la socket reçoit des données

mEvent\_onSocketReceived la socket a reçu toutes les data

mEvent\_onMarcReceiving le connecteur reçoit des résultats

mEvent\_onMarcReceived le connecteur a reçu tous les résultats

mEvent\_onMarcError erreur dans l’éxécution d’une commande

s’il y a erreur (marc->Status.Error == true)

marc->State.ErrorMessage = "le message d’erreur");

marc->Result.Count = 0;

et terminé

s’il n’y a pas d’erreur (marc->Status.Error == false)

si marc->State .LogReturn == false pas d’autres évènements émis

si marc->State .LogReturn == true émission de mEvent\_onMarcReturn

si marc->State . LogContext == true possible émission de mEvent\_onMarcContext

si marc->State . LogResult == true possible émission de mEvent\_onMarcResult

si marc-> State . LogTable == true possible émission de mEvent\_onMarcTable

si marc->State . LogKnowledge == true possible émission de mEvent\_onMarcKnowledge

marc->Result.Count = 1 . une table de résultat de la function est stockée, sous la forme d’un objet

**mResults**.

Un objet de type **mResults** modélise une table de données en format ligne colonne

mResults \* mr;

// mr -> Names contient les noms des colonnes une liste de string

// mr -> Types contient les types des colonnes une liste de string

// mr -> Sizes contient les tailles de certains type de données variables (0 en général)

// mr -> Data contient les data. Une liste de liste de string

// mr ->lines contient le nombre de ligne de la table

// mr ->columns contient le nombre de colonne de la table

mr = (mResults\*) Result[0];

Dans le cas de cet exemple, marc->Result.Count = 1, c’est-à-dire une seule instruction, une seule table de résultat.

mr-> lines = 0 et mr->Columns = 0, soit une table de retour vide, équivalente à un retour VOID.

GPString \*s ;

s= mr->**GetData** (0,0) // s = NULL, pas de data en retour

**Exemple 2**

marc = new mARC\_Connector ;

marc->**OpenScript** ();

marc->**Push** ("Table.Get");

marc->**Push** ("Instances");

marc->**AddFunction** ();

marc->ExecuteScript ();

si marc->Status.Error est **false** (pas d’erreur)

(\*marc->**Script)**[0]  contient “Table.Get (<9 Instances);”

marc->Result.Count = 1 une ligne de commande, un tableau de résultat

mr = (mResults\*) Result[0] est différent de NULL

mr-> lines = 2

mr-> Columns =1

mr->Names[0] = “Tables” le nom de la variable de retour

mr->Types[0] = “string” le type de la variable de retour

mr->Data (**num**, 0) la valeur de la variable « Tables » de la ligne **num**

GPString \*s ;

s= mr->**GetData** (0,0) // s contient “eumaster”

s= mr->**GetData** (1,0) // s contient “eudetail”

**Exemple 3**

marc = new mARC\_Connector ;

marc->**OpenScript** ();

marc->**Push** ("Contexts.Set");

marc->**Push** ("Knowledge");

marc->**Push** ("wikipedia");

marc->**Push** ("Table.Get");

marc->**Push** ("Instances");

marc->**AddFunction** ();

marc->ExecuteScript ();

marc->**Script->Count = 2** 2 instructions

(\*marc->**Script)**[0] est “Contexts.Set (<9 knowledge/>,<9 wikipedia/>);”

(\*marc->**Script)**[1] est “Table.Get (<9 Instances);”

marc->Result.Count = 2 2 tableaux de résultats

mr = (mResults\*) Result[0] est le tableau de résultat de la 1er instruction (cf plus haut)

mr = (mResults\*) Result[1] est le tableau de résultat de la 2e instruction (cf plus haut)

Execution d’une commande mARC encapsulée

Un certain nombre de commandes de l’API mARC sont encapsulées sous forme de méthodes C++ Standards

Liste des commandes encapsulées (non exhaustive, cf mARCconnector.h)

mResults\* KnowledgeGetInstances (void);

mResults\* TableGetInstances (void);

mResults\* GetTasks (void);

mResults\* GetAPI (void);

double GetLastTime (void);

GPString\* ServerGetName (void);

mResults\* ContextsGetStack (int max\_context, int max\_context\_size);

int ContextsGetCount (void);

bool ContextsGetSortOrder (int idx);

GPString\* ContextsGetSortedBy (int idx);

void ContextsSortByGenerality (bool order,int idx);

void ContextsSortByActivity (bool order,int idx);

int ContextsGetAct (int idx);

int ContextsGetSize (int idx);

mResults\* ContextsGetElements (int idx, int max\_atoms);

mResults\* ContextsNew (void);

mResults\* ContextsClear (void);

mResults\* ContextsSetKnowledge (GPString &know);

mResults\* ContextsAddElement (GPString &know);

mResults\* ContextsNewFromSem (bool relation ,int max\_act, int max\_number);

mResults\* KnowledgeGetProperties (GPString \*knw);

mResults\* TableGetProperties (GPString \*knw);

int ResultsGetCount (void);

*1 Exécution directe*

**Exemple** :

int nb;

marc->State.DirectExecute = true ;

nb = marc-> **ContextsGetCount**();

la commande est éxécutée directement comme n’importe quelle fonction C++

**nb** contient le nb de contextes sur la pile des contextes de la session serveur en cours

*2 Exécution scriptée*

int nb;

marc->State.DirectExecute = false ;

nb = marc-> **ContextsGetCount**();

nb = -1, c’est à dire non significatif.

Contexts.Get (<5 count/>) ; a été rajouté dans marc->Script.

Il est ainsi possible de mixer des méthodes encapsulées et des instructions de script directes

**Exemple**

marc = new mARC\_Connector ;

marc->State->DirectExecute = false ;

marc->**OpenScript** ();

marc-> **ContextsGetCount**();

marc->**Push** ("Table.Get");

marc->**Push** ("Instances");

marc->**AddFunction** ();

marc->ExecuteScript ();

est équivalent à :

marc->**OpenScript** ();

marc->**Push** ("Contexts.Get");

marc->**Push** ("Count");

marc->**AddFunction** ();

marc->**Push** ("Table.Get");

marc->**Push** ("Instances");

marc->**AddFunction** ();

marc->ExecuteScript ();

NB : l’éxécution de ce script générera en supplément les évènements

mEvent\_onMarcTable //appel à une métode de type Table

mEvent\_onMarcContext //appel à une métode de typeContext

Execution d’une commande mARC non blocante

Dans les exemple précédents, l’éxécution des commandes et scriptes était bloquante.

Ceci correspond à marc->Status.Blocking = **true** ;

Il est possible d’éxécuter un script ou une instruction de manière non bloquante.

Auquel cas il ne faut pas en attendre un valeur de retour directe. Seules les routines branchées sur un évènement, en particulier mEvent\_onMarcReturn , seront activées et pourront ainsi exploiter le tableau marc->Result.

Le script est éxécuté au sein d’un thread indépendant de celui du thread appelant.

Installation d’un gestionnaire d’évènement

Elle se fait par l’intermédiaire des méthodes **Plug** et **UnPlug**

**Exemple**  :

.hpp

class **Texemple**

{

static void (\***onReturn**) (mARC\_Connector \*marc, void \*object);

void **ToDoOnReturn** (mARC\_Connector \*marc);

};

.cpp

void **Texemple**::onReturn(mARC\_Connector \*marc,void \*object)

{

if (marc == NULL) return;

if (object == NULL) return;

//en général, mais pas obligatoirement

**Texemple** \*exemple\_object ;

exemple\_object  = (**Texemple**\*) object ; //cast vers le type d’objet attendu

exemple\_object -> **ToDoOnReturn** (marc);

}

void **Texemple**:: ToDoOnReturn (mARC\_Connector \*marc)

{

//code de gestion de l’évènement

// généralement pour ce genre d’évènement

// l’examen de marc->Result

}

mARC\_Connector  \*marc;

Texemple \*object;

marc = new **mARC\_Connector** ;

object = new **Texemple**;

marc->**Plug** (**Texemple**::onReturn, object);

Tout retour de méthode dans le connecteur marc appelera la routine object->**ToDoOnReturn**

marc->UnPlug (object);

éliminera object du gestionnaire d’évènement du connecteur marc, pour tous les évènements possibles.

Inhibition / activation d’évènements

Il est possible d’inhiber ou d’activer certains évènements  en jouant sur les états suivants :

marc->State .LogReturn = true; mEvent\_onMarcReturn actif

marc->State . LogContext = true ; mEvent\_onMarcContext actif

marc->State . LogResult = true; mEvent\_onMarcResult actif

marc-> State . LogTable = true ; mEvent\_onMarcTable actif

marc->State . LogKnowledge = true; mEvent\_onMarcKnowledge actif

marc->State .LogReturn = false ; mEvent\_onMarcReturn inactif

marc->State . LogContext = false ; mEvent\_onMarcContext inactif

marc->State . LogResult = false; mEvent\_onMarcResult inactif

marc-> State . LogTable = false ; mEvent\_onMarcTable inactif

marc->State . LogKnowledge = false; mEvent\_onMarcKnowledge inactif

NB : si State .LogReturn == false les 4 autres évènements seront également implicitement desactivés.

Sauvegarde du contexte du connecteur

Méthodes

void PushContext (void); //sauvegarde le contexte du connecteur

void PopContext (void); //restaure le contexte du connecteur

**PushContext** sauvegarde Result et State sur une pile interne

**PopContext** restaure Result et State depuis la pile interne

On peut ainsi partager un connecteur, et restaurer son état d’origine correspondant à celui attendu par une autre routine appelante.

**Exemple** :

int nb;

marc-> PushContext() ; //on sauve le contexte connecteur

marc->State.DirectExecute = true;

nb = marc-> **ContextsGetCount**();

marc-> PopContext() ;

on a récupéré le nombre de contexte sur la pile des contextes de la session serveur, mais en gardant intacts les résultats et états précédents du connecteur.

Création de connecteurs enfants

Méthodes

mARC\_Connector\* GetNewConnector (void);

on peut créer des connecteurs enfants qui partageront certaines caractérisques du connecteur parent.

**Exemple** :

mARC\_Connector  \*marc;

mARC\_Connector  \*childmarc;

marc = new mARC\_Connector ;

childmarc = marc-> GetNewConnector () ;

…..

marc->Connect() ; // marc **et tous ses enfants** se connecteront au même serveur

……

**delete** marc ; //marc, et tout ses enfants seront détruits

Les connecteurs enfants héritent du **Port** et de l’adresse **IP** du serveur Parent, mais pas des gestionnaires d’évènements.

Référence des méthodes :

bool Lock (void);

bool UnLock (void);

void Connect (void); //connexion à un serveur

bool Connected (void);

bool Valid (void);

void PushContext (void); //sauvegarde le contexte du connecteur

void PopContext (void); //restaure le contexte du connecteur

void OpenScript (void);

void Push (GPString s); //empile une commande ou un param

void AddFunction (void);

bool ExecuteScript (void);

void SetTimer (double time);

void ClearResult (void);

void ClearScript (void);

bool Event (int eventnum); //génération d'évènements

bool Plug (void (\*f) (mARC\_Connector \*,

void \*object),

void\* object, int event

);

bool UnPlug (void\* object);

mARC\_Connector\* GetNewConnector (void);

Référence des propriétés :

GPString IP; //adresse IP defaut 127.0.0.1

GPString Port; // Port défaut 1254

GPString SessionId;

GPString Name;

GPList Result; //une list de mResults

GPList Script; //des lignes de script (liste de GPString)

bool Stop; //arret des process asynchrones si **true**

mARC\_State **State**;

mARC\_Status **Status**;

**State**

bool LogReturn; //activation/inhibition évènement mEvent\_onMarcReturn

bool LogContext; //activation/inhibition évènement mEvent\_onMarcContext

bool LogResult; //activation/inhibition évènement mEvent\_onMarcResult

bool LogTable; //activation/inhibition évènement mEvent\_onMarcTable

bool LogKnowledge; //activation/inhibition évènement mEvent\_onMarcKnowledge

bool LogServer; //activation / inhibition évènement mEvent\_onMarcServer

bool DirectExecute;

bool Blocking; //appel bloquant ou non

bool ContextCall; //evènement détecté mEvent\_onMarcContext

bool ResultCall; //evènement détecté mEvent\_onMarcResult

bool TableCall; //evènement détecté mEvent\_onMarcTable

bool KnowledgeCall; //evènement détecté mEvent\_onMarcKnowledge

bool ServerCall; //evènement détecté mEvent\_onMarcServer

int ScriptSize;

double LastTime; //durée du dernier script/commandet éxécuté

GPString ErrorMessage; //dernier message d'erreur

**Status**

bool Error;

bool Busy;

bool Sending;

bool Sent;

bool Receiving;

bool Received;

**Evènements**

mEvent\_onSocketConnected

mEvent\_onSocketDisconnected

mEvent\_onSocketError

mEvent\_onSocketReceiving

mEvent\_onSocketReceived

mEvent\_onSocketSending

mEvent\_onSocketSent

mEvent\_onSocketIdle

mEvent\_onSocketActive

mEvent\_onMarcReturn

mEvent\_onMarcContext

mEvent\_onMarcTable

mEvent\_onMarcKnowledge

mEvent\_onMarcServer

mEvent\_onMarcBusy

mEvent\_onMarcTimer

mEvent\_onMarcSending

mEvent\_onMarcSent

mEvent\_onMarcReceiving

mEvent\_onMarcReceived

mEvent\_onMarcResult

mEvent\_onMarcError

mEvent\_onServerEvent

**Résultats (mResult)**

int lines;

int columns;

GPList Names ;

GPList Types ;

GPList Sizes ;

GPList Data ; //liste de liste de string

GPString\* GetData (int line,int col);