Deuxième rendu :
Étude des besoins

Salmane Bah & Tristan Braquelaire & Wassim Romdan & Timothée Sollaud

Université de Bordeaux

5 juillet 2014

Table des matières

Chapitre 1

Étude des besoins

Suite à nos entretiens avec le client, nous avons élaboré une liste d'exigences à laquelle la plateforme doit répondre :

1.1 Les besoins fonctionnels

1. Établir une connexion
   - Le serveur doit permettre aux client de se connecter.
   - Le serveur doit ouvrir et maintenir une session avec chaque client.
2. Transmettre les langages pris en charge
   - Le serveur doit permettre au client de connaître les langages gérés.
3. Compiler le code
   - Le serveur doit récupérer les paramètres de compilation ainsi que le code source (fichiers ou chaînes) et le langage transmis par un client.
   - Le serveur doit vérifier que la requête de compilation est bien valide.
   - Le serveur doit créer le(s) fichier(s) temporaire(s) qui contient le code source reçu.
   - Le serveur doit lancer une sandbox dans laquelle il va compiler le(s) fichier(s) temporaire(s).
   - Le serveur doit retourner le résultat de la compilation au client et le message d'erreur éventuellement associé.
4. Exécuter le code
   - Le serveur doit lancer dans une sandbox l'exécution d'un fichier déjà compilé.
   - Le serveur doit retourner le résultat de l'exécution au client ou une erreur.
5. Compiler-exécuter le code
   - Le serveur doit récupérer les paramètres de compilation, les paramètres d'exécution ainsi que le code source (fichiers ou chaînes) et le langage transmis par un client.
   - Le serveur doit vérifier que le langage est bien pris en charge.
   - Le serveur doit créer le(s) fichier(s) temporaire(s) qui contient le code source reçu.
   - Le serveur doit lancer une sandbox dans laquelle il va compiler le(s) fichier(s) temporaire(s).
   - Le serveur doit retourner le message d'erreur du compilateur si la compilation échoue.
   - Le serveur doit exécuter le/les fichier(s) compilé(s).
   - Le serveur doit retourner le résultat de l'exécution au client.
6. Fermer une connexion
   - Le serveur doit permettre aux client de se déconnecter.
   - Le serveur doit détruire l'environnement docker correspondant.
   - Le serveur doit supprimer le(s) fichier(s) temporaire(s) créé(s).
7. Gestion des services
   - Le serveur doit permettre à son administrateur de récuperer les langages qu'il (le serveur) peut compiler à tout moment
   - Le serveur doit permettre à son administrateur d'ajouter, supprimer, modifier les langages et les compilateurs pris en compte
   - Le serveur doit permettre à son administrateur de modifier les paramètres (limite mémoire, temps, ...) d'exécution/compilation des clients à chaud

1.2 Les besoins non fonctionnels

1. Sécurité
   - Le serveur doit gérer des sessions et assurer l'imperméabilité entres elles. (Un client ne doit pas être en mesure de se faire passer pour un autre client afin d'exécuter du code qui ne lui appartient pas.)
2. Disponibilité
   - Le serveur doit assurer un maximum de ressources disponibles pour de nouveaux clients potentiels. Il faut donc libérer toutes les ressources utilisées par un client dès lors qu'il est inactif depuis un temps donné ou qu'il se déconnecte. La durée d'inactivité avant déconnexion sera configurée par l'administrateur de la plate-forme.
   - Le serveur doit limiter le temps d'exécution afin de se prémunir des boucles infinies. Cette limite sera fixée par l'administrateur de la plate et sera fonction du langage.
   - Le serveur doit limiter l'espace mémoire disponible pour l'exécution du code (cas de bombe fork, etc..). Cette limite sera fixée par l'administrateur de la plate et sera fonction du langage.
3. Flexibilité & Fiabilité
   - Le système développé doit permettre de répartir le traitement des requêtes sur une multitude de serveurs traitant éventuellement un panel de langages différent selon les besoins.

Chapitre 2

Les diagrammes UML

Cette partie va nous permettre de voir les choses d'un point de vue plus conceptuel au travers de diagrammes UML.

2.1 Diagramme de cas d'utilisation

Les acteurs

- Utilisateur : Il s'agit de l'utilisateur de l'API de notre plate-forme de compilation et d'exécution.
- Administrateur : Il s'agit de l'administrateur responsable des serveurs de la plate-forme de compilation et d'exécution.

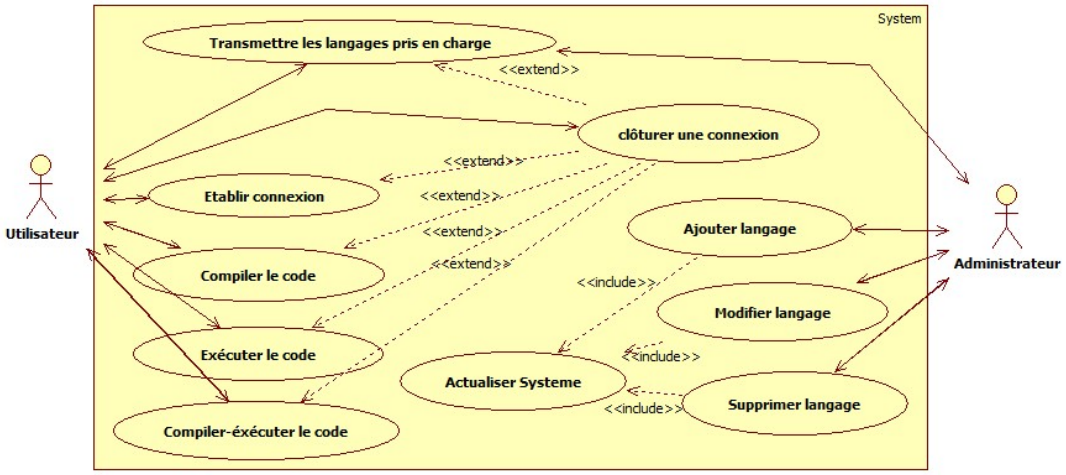

FIGURE 2.1 – Diagramme de cas d'utilisation

2.2 Diagrammes de séquences

2.2.1 Les fonctions de l'API utilisateur

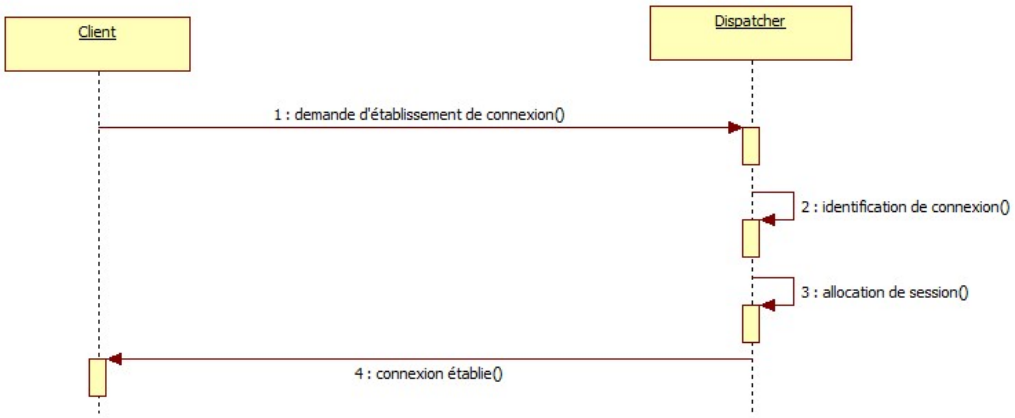

FIGURE 2.2 – Diagramme de séquence de connexion

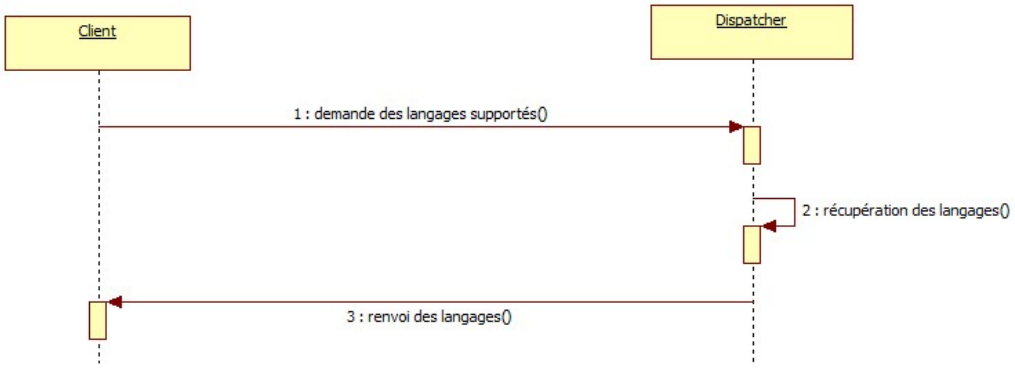

FIGURE 2.3 – Diagramme de séquence de la récupération les langages

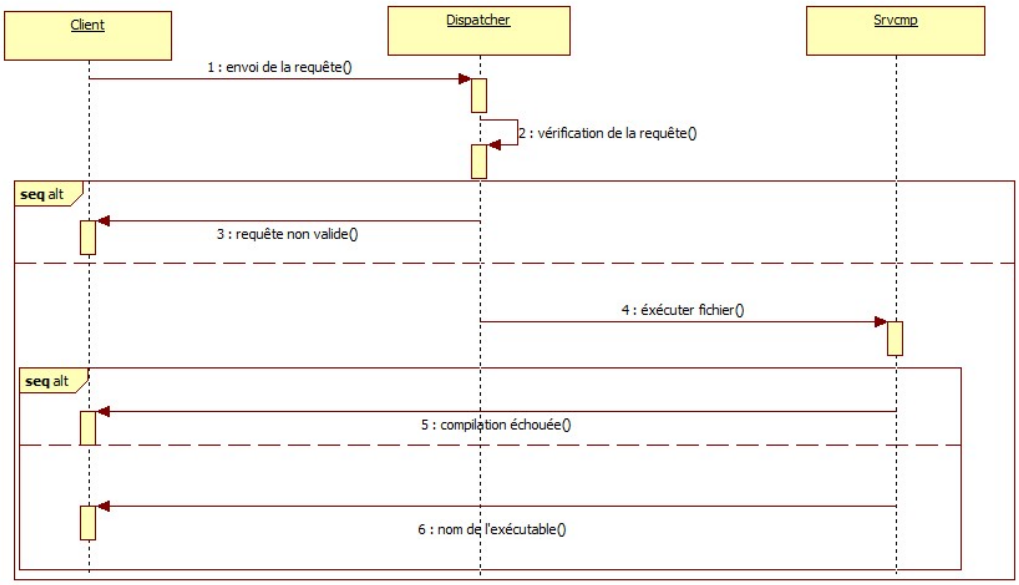

FIGURE 2.4 – Diagramme de séquence de compiler

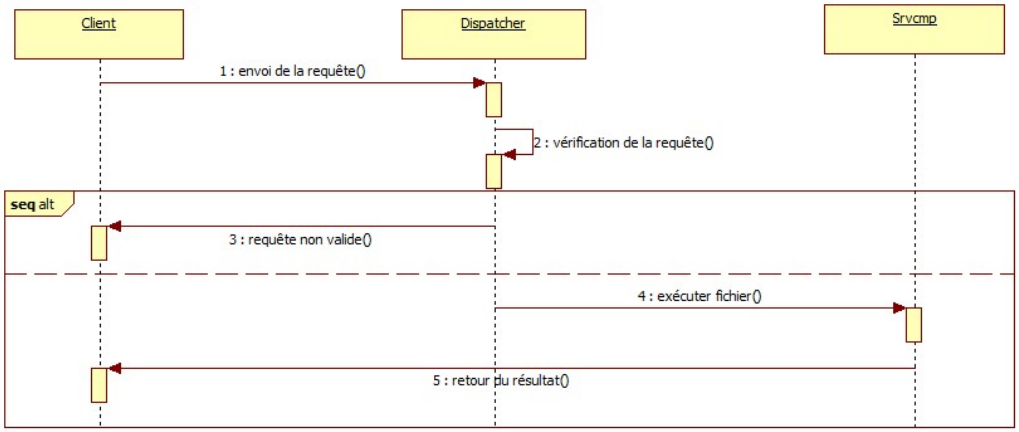

FIGURE 2.5 – Diagramme de séquence d'exécuter

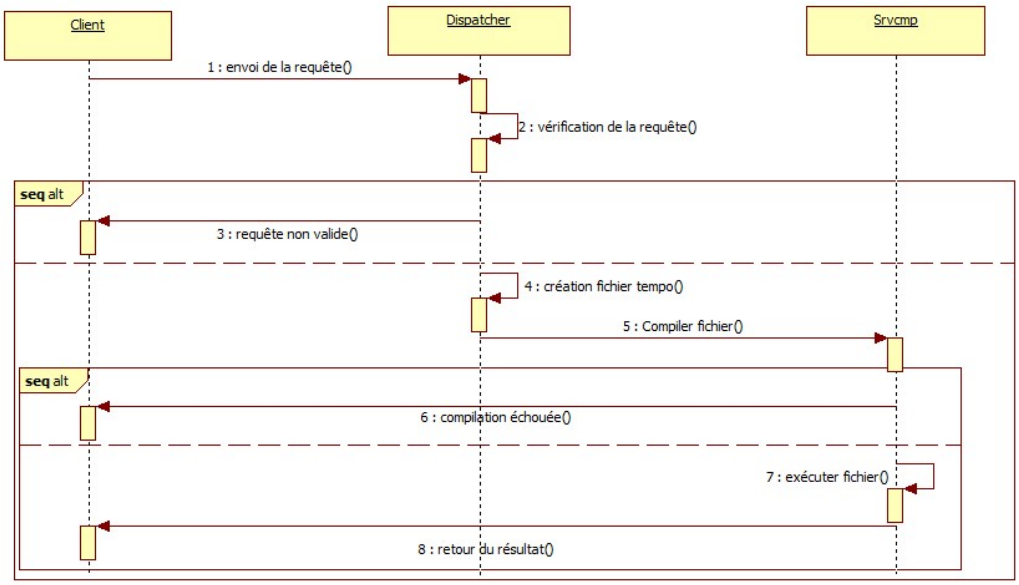

FIGURE 2.6 – Diagramme de séquence de compiler & exécuter

2.2.2 Les fonctions d'administration

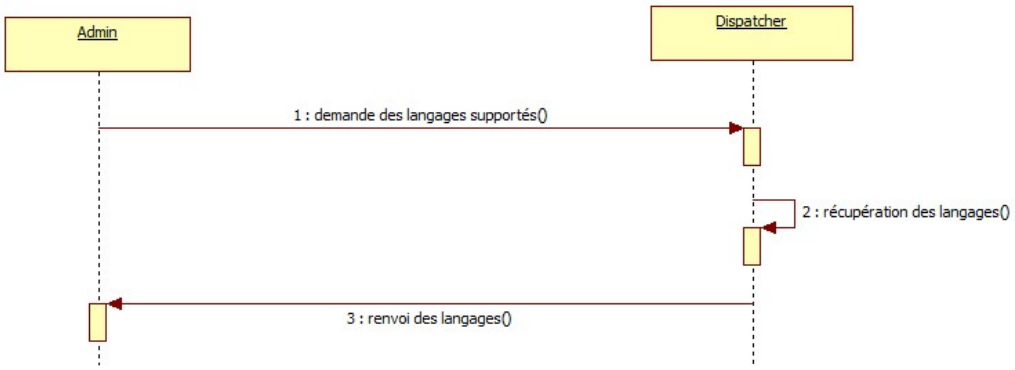

FIGURE 2.7 – Diagramme de séquence de la récupération des langages

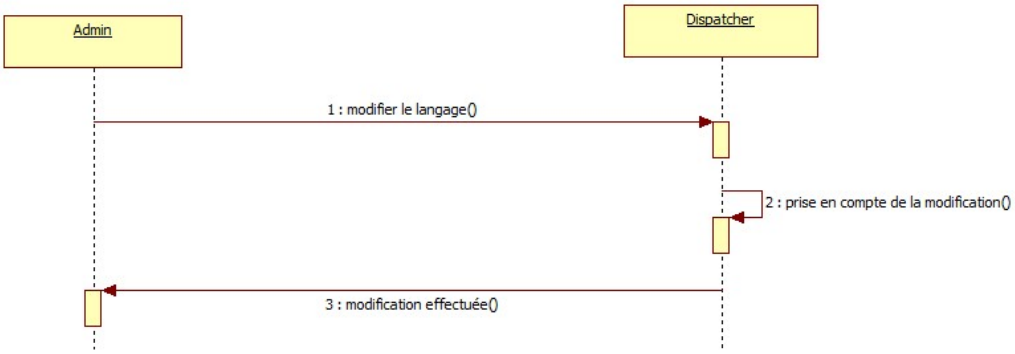

FIGURE 2.8 – Diagramme de séquence de l'ajout d'un langage

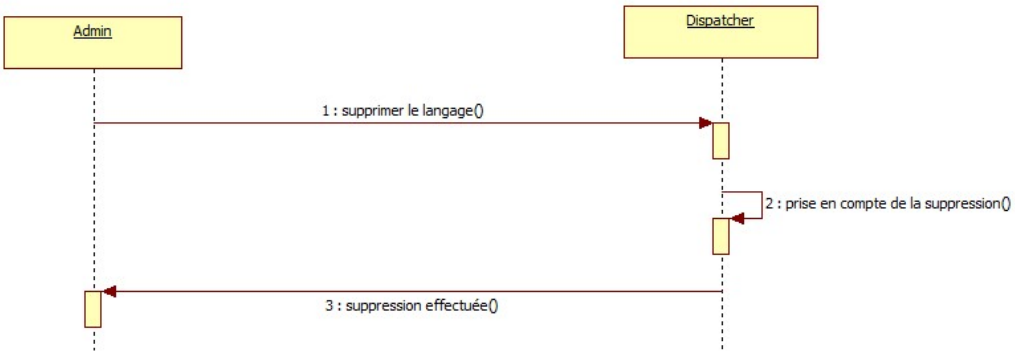

FIGURE 2.9 – Diagramme de séquence de la suppression d'un langage

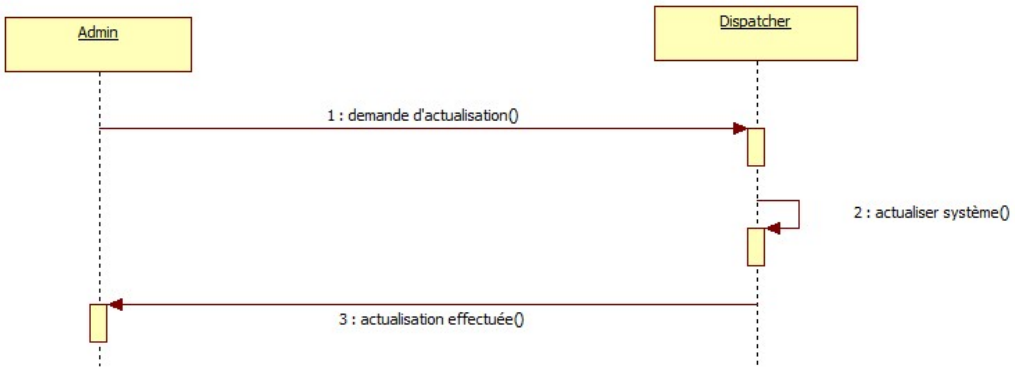

FIGURE 2.10 – Diagramme de séquence de l'actualisation du système

2.3 Diagramme de déploiement

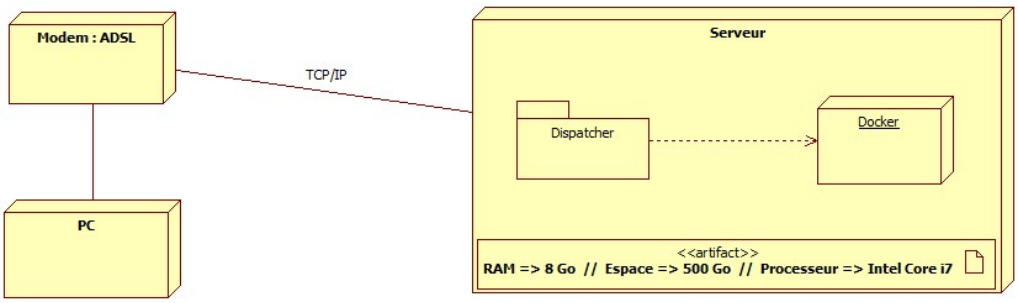

FIGURE 2.11 – Diagramme de déploiement

Chapitre 3

La spécification

Afin de répondre aux besoins du client nous avons défini une API au plus près de celle d'ideone tout en ajoutant la possibilité d'exécuter plusieurs fois un code déjà envoyé et compilé. Ceci nécessite d'inclure la gestion d'une session pour chaque client, obligeant ce dernier à transmettre sa clé de session pour la plupart des fonctions.

Pour plus de clarté nous avons établi une liste exhaustive des codes d'erreurs actuellement envisagés.

3.1 Liste des codes d'erreur (clé : retCode)

-1 erreur

0 succès

1 serveur surchargé

2 langage inconnu

3 référence inconnue

4 déjà connecté

5 sessionKey inconnue

3.2 Description des fonctions de l'API

Méthode	*connect*		
Description	Initialise une connexion avec le serveur		
Paramètres	Aucun		
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	-1	erreur
		0	succès
		4	déjà connecté
	sessionKey : int	45128545315*	Clé de session unique

*Valeur arbitraire (dépend de l'état du serveur). Si retCode est à -1, vaut -1.

Méthode	*disconnect*	
Description	Ferme une connexion avec le serveur	
Paramètres	sessionKey : int, clé de session (cf.connect())	
Type de retour	int	
Valeur de retour	**Valeur**	**Description**
	-1	erreur
	0	connexion fermée
	5	sessionKey inconnue

Méthode	*getLanguages*		
Description	Retourne les langages supportés		
Paramètres	Aucun		
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	-1	erreur
		0	succès
	languages : dico	{1 : "C gcc", 2 : "Java javac"}*	dictionnaire des langages idLang : int (id du langage) details : String (langage & compilateur)

*Valeur arbitraire (dépend de la configuration du serveur).

Méthode	*sendRequest*		
Description	Soumet une requête de compilation (et exécution) au serveur		
Paramètres	**Nom**	**Type**	**Description**
	sessionKey	int	clé de session (cf.connect())
	codeLang	int	code du langage (cf.getLanguage())
	srcCode	String	code source à soumettre
	compilOpt	String	option de compilation
	cmdArgs	String	arguments d'exécution
	stdin	String	chaînes d'entrée standard
	toRun	boolean	active l'exécution ou non
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	-1	erreur
		0	succès
		1	serveur surchargé
		2	langage inconnu
		5	sessionKey inconnue
	reference : int	1*	identifiant unique de la requête

*Valeur arbitraire relative au client. Vaut -1 si retCode n'est pas à 0.

Méthode	*run*		
Description	Exécute le code compilé de la tache référencée		
Paramètres	**Nom**	**Type**	**Description**
	sessionKey	int	clé de session (cf.connect())
	reference	int	reference de la tache (cf.sendRequest)
	cmdArgs	String	arguments d'exécution
	stdin	String	chaînes d'entrée standard
Type de retour	int		
Valeur de retour	**Valeur**	**Description**	
	-1	erreur	
	0	succès	
	1	serveur surchargé	
	3	référence inconnue	
	5	sessionKey inconnue	

Méthode	*getDetails*		
Description	retourne les informations associées à une requête déjà soumise		
Paramètres	**Nom**	**Type**	**Description**
	sessionKey	int	clé de session (cf.connect())
	reference	int	reference de la tache (cf.sendRequest)
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	-1	erreur
		0	succès
		3	référence inconnue
		5	sessionKey inconnue
	status : int	0	traité
		1	en cours de compilation
		2	compilé
		3	en cours d'exécution
		4	en attente
	result : int	0	succès
		1	erreur de compilation
		2	erreur d'exécution
	compLog : String		Log produit par le compilateur
	stdOut : String		Sortie produite par l'exécution
	stdErr : String		Sortie d'erreur produite par l'exécution
	timeInfo : float		info sur la durée d'exécution
	memInfo : float		info sur la mémoire utilisée

Méthode	*getStatus*		
Description	retourne l'état de la requête référencée		
Paramètres	**Nom**	**Type**	**Description**
	sessionKey	int	clé de session (cf.connect())
	reference	int	reference de la tache (cf.sendRequest)
Type de retour	dictionnaire		
Valeur de retour	**Clé : Type**	**Valeur**	**Description**
	retCode : int	-1	erreur
		0	succès
		3	référence inconnue
		5	sessionKey inconnue
	status : int	0	traité
		1	en cours de compilation
		2	compilé
		3	en cours d'exécution
		4	en attente
	result : int	0	succès
		1	erreur de compilation
		2	erreur d'exécution

Chapitre 4

Scénario & Tests

Pour l'étape de validation, nous avons mis on place les scénarios possibles ainsi que les tests prévus sur notre plateforme.

4.1 Scénario

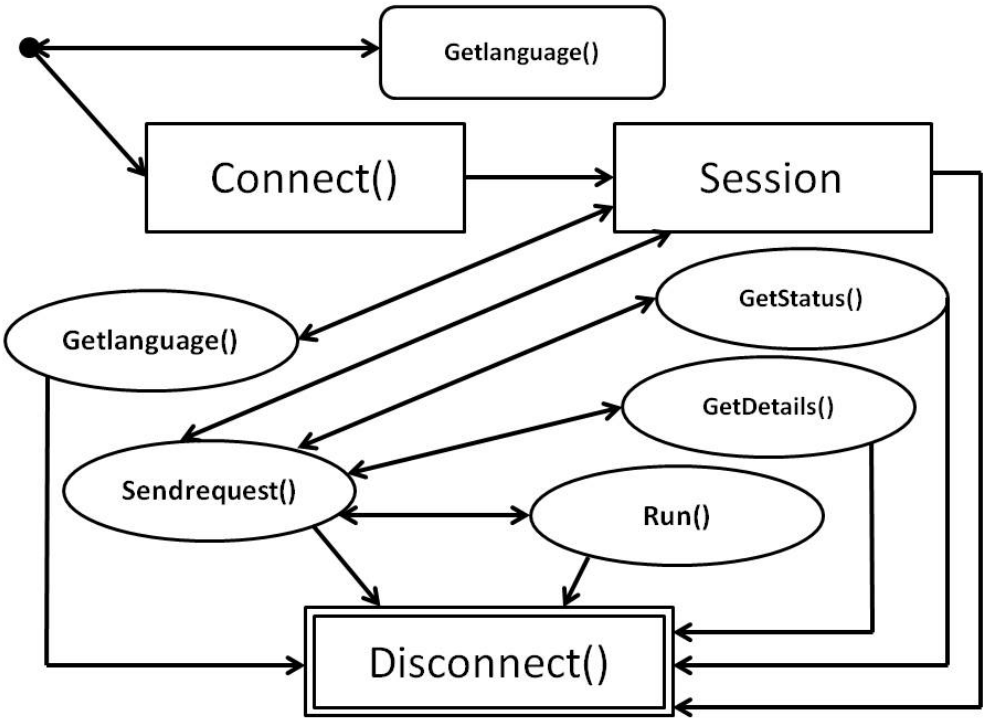

FIGURE 4.1 – Scénario des possibles

L'utilisateur de l'API peut a tout moment obtenir la liste des langages gérés par le serveur. Toute autre action reposant sur la session du client nécessite sa connexion.

Un client connecté peut demander la liste des langages gérés par la plate-forme de compilation & d'exécution, envoyer une requête ou se déconnecter.

Après l'envoi d'une requête, le client reçoit la référence de cette dernière. Il peut alors utiliser de nouvelles fonctions lui permettant de récupérer les informations détaillées sur sa requête, de récupérer le statut de la requête, ou encore de demander l'exécution du code lié à cette requête.

4.2 Les tests

Pour l'ensemble des tests suivants, nous considérons -1 comme étant un numero de requête et une sessionKey invalide.

4.2.1 fonction *connect()*

1. **appel :** connect(1)
   retour attendu : message erreur de SOAP : connect() ne prend pas de paramètre.
2. **appel :** connect()
   retour attendu : 0 ; <sessionKey>
   appel : connect()
   retour attendu : 4 ; <sessionKey>

4.2.2 fonction *disconnect(sessionKey)*

1. **appel :** disconnect()
   retour attendu : message erreur de SOAP : connect() requiert un paramètre.

1. **appel :** disconnect(-1)
   retour attendu : 5

4.2.3 fonction *getLangages()*

1. **appel :** getLangages(1)
   retour attendu : message erreur de SOAP : getLangages ne prend pas de paramètre.

4.2.4 fonction *sendRequest(sessionKey, codeLang, srcCode, compilOpt, cmdArgs, stdin, toRun)*

On attribuera aux entiers 1 et -999 respectivement le langage C et un langage non attribué.
- code_bon_C : code en langage C ne posant pas de problème de compilation
- code_faux : code en langage C ne compilant pas
- code_1arg_bon : code qui requiert un argument ne posant pas de problème de compilation
- code_stdin_bon : code qui requiert du contenu provenant de l'entrée standard ne posant pas de problème de compilation
- code_bon_JAVA : code en langage JAVA ne posant pas de problème de compilation

1. **appel :** sendRequest()
   retour attendu : message erreur de SOAP : Test sendRequest requiert 7 paramètres.
2. **appel :** sendRequest(<sessionKey>, 1,code_bon_C, "", "", "", 0)
   retour attendu : 0 ; <numéro requete>

3. **appel :** sendRequest(<sessionKey>, -999, code_bon_C, "", "", "", 0)
   retour attendu : 2 ; -1

4. **appel :** sendRequest(-1, 1, code_bon_JAVA, "", "", "", 0)
   retour attendu : 5 ; -1

5. **appel :** sendRequest(<sessionKey>, 1, code_faux, "", "", "", 0)
   retour attendu : 0 ; <numéro requete>

6. **appel :** sendRequest(<sessionKey>, 1, code_bon_C, "mauvaise option", "", "", 0)
   retour attendu : 0 ; <numéro requete>

7. **appel :** sendRequest(<sessionKey>, 1, code_1arg_bon, "", "", "", 0)
   retour attendu : 0 ; <numéro requete>

8. **appel :** sendRequest(<sessionKey>, 1, code_1arg_bon, "", "1 2", "", 0)
   retour attendu : 0 ; <numéro requete>

9. **appel :** sendRequest(<sessionKey>, 1, code_stdin_bon, "", "", "c", 0)
   retour attendu : 0 ; <numéro requete>

10. **appel :** sendRequest(<sessionKey>, 1, code_stdin_bon, "", "", "", 0)
    retour attendu : 0 ; <numéro requete>

11. **appel :** sendRequest(<sessionKey>, 1, code_bon_C, "", "1", "1", 0)
    retour attendu : 0 ; <numéro requete>

12. **appel :** sendRequest(<sessionKey>, 1, code_bon_JAVA, "", "", "", 0)
    retour attendu : 0 ; <numéro requete>

Les tests précédents doivent renvoyer les même résultat avec le booléen d'éxécution à 1.

4.2.5 fonction *run(int sessionKey, int reference, String cmdArgs, String stdin))*

- requête 1 : code sans paramètre, sans stdin
- requête 2 : code avec 1 paramètre, sans stdin
- requête 3 : code sans paramètre, avec stdin
- requête 4 : code avec 1 paramètre, avec stdin

1. **appel :** run()
   retour attendu : message erreur de SOAP : Test sendRequest requiert 4 paramètres.
2. **appel :** run(-1, 1, “”, “”)
   retour attendu : 5
3. **appel :** run(<sessionKey>, -1, “”, “”)
   retour attendu : 3
4. **appel :** run(<sessionKey>, 1, “1”, “stdin”)
   retour attendu :0
5. **appel :** run(<sessionKey>, 1, “”, “stdin”)
   retour attendu :0
6. **appel :** run(<sessionKey>, 1, “1”, “”)
   retour attendu :0
7. **appel :** run(<sessionKey>, 1, “”, “”)
   retour attendu :0

Les quatres derniers tests appliqués respectivement aux requetes 2, 3 et 4 doivent également retourner 0.

4.2.6 fonction *getDetails(int sessionKey, int reference))*

1. **appel :** getDetails()
   retour attendu : message erreur de SOAP : Test getDetails requiert 1 paramètre.
2. **appel :** getDetails(-1, 1)
   retour attendu : 5
3. **appel :** getDetails(-1, -1)
   retour attendu : 3

4.2.7 fonction *getStatus(int sessionKey, int reference))*

1. **appel :** getStatus()
   retour attendu : message erreur de SOAP : Test getStatus requiert 1 paramètre.

2. **appel :** getStatus(-1, 1)
   retour attendu : 5
3. **appel :** getStatus(-1, -1)
   retour attendu : 3

Chapitre 5

Prototype python

Ce prototype montre la possibilité d'utiliser docker comme moyen de créer des sandbox à la volée et SOAP comme moyen d'effectuer des appels de fonction à distance.
Il compile et éxecute du code écris en C par un client distant connecté à la plateforme. Voici quelques extraits de code de ce prototype (le code source complet se trouve sur le dépôt savanne du projet dans le répertoire trunk/src/prototype).
Les services offerts par la plateforme :

- comp : pour compiler du code C passé sous forme de chaîne de caractère
- execute : pour éxecuter du code précedemment compilé sur la plateforme

```
def comp(srcString) :
"""Compile␣the␣C␣source␣string␣in␣paramater␣and␣returns
␣␣␣the␣compilation␣log␣and␣executable␣name"""

    (result , execName) = DH.doCompile(hocker , srcString)
    return {'compilationMsg' : result , 'executableName' : execName}
```

```
def execute(filename) :
"""␣Execute␣a␣previously␣compiled␣file␣with␣name
␣␣␣␣in␣parameter␣and␣returns␣the␣result␣"""
    result = DH.doExecute(hocker , filename)
    return result
```

Enregistrement des services *comp* et *execute* de la plateforme sur le serveur avec comme nom de procédure distante respectivement *Compile* et *Execute*

```
dockerDispatcher.register_function
('Compile',comp,returns={'compilationMsg':str,'executableName':str},
                args={'srcString':str})
```

```
dockerDispatcher.register_function('Execute',execute,
                                   returns={'executionResult':str} ,
                                   args ={'filename' : str})
```

Les services *Compile* et *Execute* utilisent respectivement les fonctions *doCompile* et *doExecute* qui lancent dans un processus fils la commande docker pour la compilation et l'execution dans une sandbox. Ces fonctions capturent la sortie standard du processus fils et la retourne au client.
doCompile retourne le log de compilation en cas d'erreur ou la chaîne "success" et le nom de l'éxecutable crée.

```
def doCompile(docker , instruction = None , srcName = None) :
    if instruction :
        fileName = _createFileWith(instruction)
        return _compileHelper(docker , fileName)
    elif srcName :
        return _compileHelper(docker , srcName)
    else :
        print 'Give␣a␣parameter'
        sys.exit(-1)
```

doExecute retourne le resultat d'execution (stdout) du programme passé en paramètre puis supprime les fichiers source et éxecutable genérés par la requête du client.

```
def doExecute(docker , srcName) :
    cmdOpt = docker.execString() + './' + srcName
    out=subprocess.Popen(cmdOpt,shell=True,stdout=subprocess.PIPE)
    out.wait()
    #remove src and exec file if exist
    if os.path.exists('./' + srcName) :
        os.remove('./' + srcName)

    if os.path.exists(('./' + srcName)[:-4]) :
        os.remove(('./' + srcName)[:-4])
    return out.stdout.read()
```

Voici à titre d'exemple un client se connectant à la plateforme et soumettant une requête de compilation et d'éxecution.

```
instructions = """␣#include␣<stdio.h>
int␣main(int␣argc␣,␣char*␣argv[])
{
␣␣␣␣␣␣printf("Hello␣World␣from␣Docker");
␣␣␣␣␣␣return␣0;
}
"""

client = SoapClient(wsdl="http://127.0.0.1:8009")
res = client.Compile(srcString = instructions)
if res['compilationMsg'] != 'success' :
    print 'Error:␣' + res['compilationMsg']
else :
    val = client.Execute(res['executableName'])
    print 'Result␣:␣' + val['executionResult']
```