# Python et l'agrégation d'outils

**Sur cet article**

Cet article est paru originellement dans le numéro 3/2007 de la revue francophone du Linux Developer Journal. La version présentée ici reprend globalement l'article paru, en y ajoutant des liens hypertext et des références.



*Utilisé autant dans le cadre de petits scripts répondant à des besoins immédiats, que pour des logiciels plus pérennes de toutes envergures, le langage Python excelle là où il faut interconnecter différentes technologies afin d'agréger des outils disparates utilisant des moyens de communication divers. Nous allons montrer ici l'utilisation de certaines techniques d'intégrations avec Python et donner des liens vers d'autres techniques et outils.*

**Notes aux lecteurs**

Sauf indication particulière, les modules (librairies) indiquées font parties de la distribution standard de Python et sont donc directement disponibles lorsque celle-ci est installée. Les sites où trouver les librairies tiers sont indiqués à la fin de l'article. Les exemples de scripts ont été testés avec la version 2.4.3 de Python, sur un Linux Mandriva 2007.0.

Le shebang est à adapter à votre distribution (`/usr/bin/env python`, `/bin/python`…), et l'encodage à adapter à celui utilisé par votre éditeur (on utilise souvent `latin1` et `ascii`, et maintenant de plus en plus `utf-8`).

Pour faciliter la compréhension on a préfixé explicitement le nom des modules lors de l'appel aux fonctions dans les exemples. Python permet des raccourcis pour éviter cette lourdeur d'écriture.

Pour l'installation de modules tiers, la première chose à regarder est s'ils ne sont pas disponibles dans le gestionnaire de packages de votre distribution. Si ce n'est pas le cas, la solution la plus simple est d'utiliser easy_install [EASYINSTALL], une application qui s'occupe de télécharger le module désiré (ainsi que les autres modules nécessaires s'il y a lieu) et de l'installer, à condition que le développeur ait créé un fichier .egg correspondant au module et déposé le tout sur le Python Package Index [PYPI]. Si l'on ne dispose que d'une archive, on la décompresse et si elle contient un fichier `setup.py`, on ouvre un shell dans le répertoire qui contient ce fichier puis on exécute python `setup.py install`. S'il n'y a pas de `setup.py`, il faut généralement copier les fichiers dans un des répertoires du **PYTHONPATH** – on utilisera le répertoire site-packages du répertoire des librairies de Python. Et sinon… il faut lire la documentation du module pour connaître sa procédure d'installation (certains passent par les habituels `./configure ; make ; make install`).

# Contrôle de processus

C'est le b.a.-ba de la réutilisation de logiciels existant, être capable de démarrer un exécutable tiers, de lui fournir les données en entrée s'il y a lieu, et de récupérer les résultats en sortie s'il en produit. On peut aussi vouloir un contrôle plus fin, avec une adaptation des entrées fournies au logiciel suivant des éléments extérieurs ou encore suivant les résultats qu'il produit.

## Fonctions des librairies C

Le module `os` fournit les fonctions habituelles des librairies C, `system()`, les familles `exec…()` et `spawn…()` (avec leurs variantes permettant d'indiquer l'utilisation ou non du **PATH**, le passage de variables d'environnement, et les façons de transmettre les arguments), `fork()`. Cf. Listing 1.

Listing 1. Fonctions de base des librairies C

**listing1.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import os
os.system('ls -l /')
exitcode = os.spawnlp(os.P_WAIT,'ls','ls','-l','/')
pid = os.spawnlp(os.P_NOWAIT,'ls','ls','-l','/')
```

L'utilisation de l'option **`P_NOWAIT`** permet de lancer l'exécutable tiers en parallèle au script Python en cours, le pid récupéré pouvant être utilisé pour connaître l'état du processus lancé, attendre sa terminaison, le tuer – le module os fournit les fonctions ad-hoc (`wait()`, `waitpid()`, `kill()`…). On notera dans les exemples `spawn` la répétition du nom de l'exécutable en première position des paramètre.

Le module `os`, ainsi que le module **`popen2`**, fournissent des fonctions et des classes permettant de lancer des processus en maintenant des pipes de communication avec leurs entrées/sorties standard (attention à l'ordre des pipes retournés, qui diffèrent entre les fonctions de **`os`** et celles de **`popen2`**). Pour les cas simples, le module commands définit des fonctions évitant d'avoir à manipuler les pipes. Cf. Listing 2.

Listing 2. Contrôle de base via des pipes

**listing2.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import popen2,commands
childout,childin = popen2.popen2('ls -l /')
res1 = childout.readlines()
res2 = commands.getoutput('ls -l /')
```

# Module subprocess

Les fonctions des librairies C fonctionnent bien, mais ne sont pas toujours pratiques à utiliser, par exemple lorsque l'on veut effectuer des pipes entre différents exécutables tiers, ni toujours très sécurisés (appel à `/bin/sh` de façon implicite). Depuis sa version 2.4, Python dispose d'un module **`subprocess`** qui remplace avantageusement ces outils dans la plupart des cas.

Il fonctionne via l'utilisation d'objets de la classe **`Popen`**, dont les nombreux paramètres de construction permettent de définir leur utilisation. Outre l'indication de l'exécutable et de ses paramètres, on peut spécifier le répertoire courant d'exécution, les variables d'environnement,

indiquer une fonction à appeler avant de créer le processus fils… et bien sûr spécifier comment on désire utiliser les flux en entrée et en sortie. Le Listing 3 donne un exemple réalisant l'équivalent d'une commande `shell ls -l / | sort -k 2 -n`.

Listing 3. Utilisation de subprocess

**listing3.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import subprocess
pls = subprocess.Popen(['ls','-l','/'],
            stdout=subprocess.PIPE)
psort = subprocess.Popen(['sort', '-k','2','-n'],
            stdin=pls.stdout, stdout=subprocess.PIPE)
res = psort.communicate()[0]
print res
```

La documentation de Python contient des indications sur l'utilisation de ce module pour réaliser diverses opérations habituelles et remplacer les appels aux fonctions des librairies C.

Que ça soit avec **`os`**, **`popen2`** ou **`subprocess`**, lorsque l'exécutable doit fonctionner en étant alimenté régulièrement en données et/ou doit fournir un flux de résultats, il faut alors directement manipuler les pipes et utiliser leurs méthodes `write()`, `flush()`, `readline()`… éventuellement à partir de deux threads séparées si les flux doivent être simultanés. Cf. Listing 4.

Listing 4. Manipulation directe des pipes avec subprocess

**listing4.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import subprocess
s = subprocess.Popen(['bc','-q'],stdin = subprocess.PIPE,
            stdout = subprocess.PIPE, stderr = subprocess.STDOUT)
print dir(s.stdout)
while True :
    cmde = raw_input("Calcul? ")
    if not cmde : break
    s.stdin.write(cmde+'\n')
    s.stdin.flush()
    if '=' not in cmde :
        res = s.stdout.readline()
        print "Resultat:",res
```

```
s.stdin.close()
s.stdout.close()
```

## Module pexpect

Lors que l'on désire contrôler des exécutables interactifs qui n'utilisent pas les flux `stdin`/`stdout`/`stderr` et agissent directement avec le terminal, ou encore qui posent des problèmes lors de communications via des pipes (bufferisations gènantes…), il vaut mieux se tourner vers le module tiers **`pexpect`**.

Il permet d'envoyer des commandes vers l'exécutable, de définir des patterns possibles attendus en sortie, de récupérer le contenu des sorties, de donner la main à l'utilisateur… Il existe par ailleurs des extensions spécifiques destinées à certains cas particuliers, par exemple **`pxssh`** pour le support de `ssh`.

# Invocation de Procédures Distantes

Cela consiste à appeler une fonction/méthode comme si l'on effectuait un appel local à l'application courante, mais avec un transfert transparent de l'appel vers une autre application en cours de fonctionnement. Une couche intermédiaire prend en charge l'acheminement des paramètres, des valeurs de retour, et éventuellement des retours d'exceptions suite à des erreurs.

## Sun RPC

Les Sun Remote Procedure Call sont un grand standard, accessible avec l'installation du module tiers pyrpc [PYRPC]. Celui-ci nécessite l'installation préalable de **`Ply`** (une implémentation de Lex et Yacc pour Python).

La distribution de **`pyrpc`** vient avec un générateur de code, `pyrpcgen`, qui permet de générer le code client RPC à partir de la description au format `XDR` (eXternal Data Representation) de l'interface à laquelle on veut accéder.

En prenant un exemple simple [EX-RPC] (lancer `make` deux fois pour arriver à générer les binaires), qui fournit un service faisant une opération d'addition et une opération de soustraction. On commence par faire un pré-processing sur le fichier XDR (cf. Listing 5), `cpp -P simp.x -o simp.xdr`, on crée ensuite le répertoire destiné à recevoir le package Python correspondant, mkdir pysimp, et on utilise finalement pyrpcgen sur le fichier XDR préprocessé pour générer ce package, `pyrpcgen simp.xdr pysimp`.

Listing 5. Définition XDR de l'exemple

**exemple.xdr**

```
#define VERSION_NUMBER 1
struct operands {
    int x;
    int y;
};
```

```
program SIMP_PROG {
    version SIMP_VERSION {
        int ADD(operands) = 1;
        int SUB(operands) = 2;
    } = VERSION_NUMBER;
} = 555555555;
```

Il nous faut maintenant trouver le moyen de connaître le port sur lequel notre service RPC va être disponible. Démarrons ce service (exécutable server produit par les make sur l'exemple). Ce port est choisi au hasard, mais il est enregistré auprès du service standard portmapper, qui lui fonctionne sur le port connu et réservé `111` (voir `/etc/services`). On peut donc trouver à la main où notre service est accessible via une commande `/usr/sbin/rpcinfo -p`, mais ce qui est raisonnable pour quelques tests doit pouvoir se faire de façon automatique si l'on veut avoir un programme capable de rentrer en production. Qu'à cela ne tienne, le portmapper est simplement un service RPC, il suffit de trouver la définition de son interface XDR (une piste, la RFC 1833, cf. Listing 6), et de refaire les mêmes opérations que pour notre exemple. Note: on commentera la définition du type pmaplist ainsi que celle de la fonction `PMAPPROC_DUMP`, car elles ne passent pas la compilation par `pyrpcgen` à cause du symbole `*`. Cf. Listing 7.

**Liste des services enregistrés auprès du portmapper**

```
/usr/sbin/rpcinfo -p
program no_version protocole no_port
   100000 2 tcp 111 portmapper
   100000 2 udp 111 portmapper
   100024 1 udp 32769 status
   100024 1 tcp 38070 status
555555555 1 udp 1003
555555555 1 tcp 1005
```

Listing 6. Définition XDR du portmapper (partie nous intéressant)

**portmapper.xdr**

```
const PMAP_PORT = 111;
struct mapping {
    unsigned int prog;
    unsigned int vers;
    unsigned int prot;
    unsigned int port;
};
const IPPROTO_TCP = 6;
const IPPROTO_UDP = 17;
program PMAP_PROG {
    version PMAP_VERS {
```

```
        void PMAPPROC_NULL(void) = 0;
        unsigned int PMAPPROC_GETPORT(mapping) = 3;
    } = 2;
} = 100000;
```

Listing 7. Utilisation du module client RPC

**listing7.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import pysimp
import portmapper
cxpmap = portmapper.client.PMAP_PROG.PMAP_VERS("localhost", portmapper.const.PMAP_PORT)
infoparams = portmapper.types.mapping(prog=pysimp.const.SIMP_PROG,
            vers=pysimp.const.SIMP_VERSION, prot=portmapper.const.IPPROTO_TCP, port=0)
port = cxpmap.PMAPPROC_GETPORT(infoparams)
cxserveur = pysimp.client.SIMP_PROG.SIMP_VERSION("localhost",port)
data = pysimp.types.operands(x=3,y=4)
res = cxserveur.ADD (data)
print res
```

Il suffit ensuite d'écrire un script client RPC qui utilise le package **pysimp** généré (penser à ce qu'il soit accessible dans le **PYTHONPATH** lors de l'exécution du script client).

# Objets Distribués

C'est l'étape suivante après les appels de procédures distantes, on ne s'adresse plus à une application dans son ensemble, mais à un objet résident dans une application. On n'appelle plus des fonctions mais des méthodes, et on accède à des attributs.

## Module Pyro

**Pyro** (Python Remote Object) est à Python ce que `RMI` (Remote Method Invocation) est à Java : une technologie d'appel de méthodes distantes dédié à un langage. Pyro n'est utilisable qu'avec un client et un serveur écrits en Python, mais il rend l'appel de procédure distante facile et quasi transparent.

Pour utiliser Pyro, nul besoin de définir une interface dans un langage tiers. Côté serveur, il suffit de créer une classe Python et soit de la faire hériter d'une classe spécifique de Pyro, soit

d'utiliser un mécanisme de délégation. Une fois le démon de Pyro démarré, il suffit d'y enregistrer l'objet qui peut alors recevoir des appels distants. Cf. Listing 8.

Listing 8. Un serveur utilisant Pyro

**listing8.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import Pyro.core
class MonServeur(Pyro.core.ObjBase):
    def __init__(self):
        Pyro.core.ObjBase.__init__(self)
    def doubler(self,x) :
        return x * 2
Pyro.core.initServer()
daemon = Pyro.core.Daemon(port=7766)
mon_serveur = MonServeur()
uri = daemon.connect(mon_serveur,"lenom")
daemon.requestLoop()
```

Côté client, il suffit de se connecter à l'objet serveur distant et de récupérer un objet proxy local permettant d'effectuer les appels au serveur. Cf. Listing 9.

Listing 9. Un client utilisant Pyro

**listing9.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import Pyro.core
serveur = Pyro.core.getProxyForURI("PYROLOC://localhost:7766/lenom")
print "Doublage chaine:",serveur.doubler("Une chaine")
print "Doublage entier:",serveur.doubler(21)
```

Si vous n'avez que des programmes Python à faire communiquer entre eux, c'est une solution à envisager sérieusement. Il s'agit d'un outil très complet, avec de nombreuses options permettant d'adapter son fonctionnement à certains besoins : logs, mécanisme d'URI, service de nommage (via des objets distants Pyro), migration de code, etc …

# CORBA

Défini et maintenu au sein de l'Object Management Group (OMG) qui rassemble des constructeurs et éditeurs majeurs, le Common Object Request Broker Architecture (CORBA) est

l'architecture logicielle standard industrielle pour les systèmes d'objets répartis multi-plateformes et multi-langages.

On utilise des définitions exprimées en Interface Definition Language (OMG/IDL) qui sont traitées pour produire des adaptateurs dans les langages de programmation cible – il existe des spécifications standards OMG/IDL pour différents langages dont Python.

Des middlewares dédiés, les Object Request Broker (ORB), se chargent d'implémenter les communications sur le bus à objets et d'adapter les requêtes pour que les objets puissent communiquer entre eux de façon transparente – ce sont ces implémentations qui fournissent les compilateurs d'OMG/IDL vers les langages cibles.

L'ORB préféré pour Python est OmniORBpy [OMNIORBPY], l'intégration dans Python de l'ORB OmniORB, il bénéficie d'une communauté très active d'utilisateurs et d'un développement suivi.

Avec cet ORB on génère les adaptateurs pour Python via la commande `omniidl -bpython monservice.idl`. Elle traite la définition d'interfaces en OMG/IDL (cf. Listing 10) et crée les packages Python (des répertoires) **`MonService`**, **`MonService__POA`**, ainsi que le module **`monservice_idl.py`**. Le premier est utilisé pour créer des clients, le second pour créer des serveurs, et le troisième est utilisé par les deux autres.

Listing 10. Définition OMG/IDL monservice.idl

**monservice.idl**

```
module MonService {
    interface Service {
        string doubler_str(in string s);
        long doubler_long(in long l);
    };
};
```

Il ne reste plus qu'à coder le client et le serveur, et à les faire communiquer.

Côté serveur (cf. Listing 11) il faut écrire une classe héritant d'un squelette créé par le compilateur IDL à partir des définitions, et créer un objet servant de cette classe, qui sera chargé de traiter effectivement les requêtes. Il faut ensuite activer ce servant auprès d'un adaptateur d'objets (Portable Object Adapter ou POA) qui s'occupera de créer un objet CORBA lié à notre servant – dans l'exemple on a utilisé de façon implicite le RootPOA qui est l'adaptateur d'objets par défaut. C'est le POA qui régit le fonctionnement de l'objet via différentes politiques de gestion : durée de vie, activation/désactivation, support multithreading… plus de détails sur la page décrivant les principes de base d'un ORB sur le site de l'OMG [ORBBASICS].

Listing 11. Un serveur utilisant CORBA

**listing11.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import sys
from omniORB import CORBA, PortableServer
import MonService, MonService__POA
```

```
class MonService_i (MonService__POA.Service):
    def doubler_str(self,s) :
        return s * 2
    def doubler_long(self,l) :
        return l * 2

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
poa = orb.resolve_initial_references("RootPOA")
service_implementation = MonService_i()
service_corba = service_implementation._this()
poaManager = poa._get_the_POAManager()
poaManager.activate()
open("id_service.txt","w").write(orb.object_to_string(service_corba))
orb.run()
```

Pour notre exemple on passe au client une information permettant de se connecter au serveur par l'intermédiaire d'un fichier texte contenant l'Interoperable Object Reference (IOR) du serveur.

Côté client (cf. Listing 12) il suffit de trouver une référence vers l'objet – dans notre exemple un IOR – puis de demander un accès à l'objet correspondant via un proxy, en s'assurant que l'on a bien un objet correspondant au service attendu. Après, il n'y a plus qu'à utiliser le proxy comme s'il s'agissait d'un objet local.

Listing 12. Un client utilisant CORBA

**listing12.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import sys
from omniORB import CORBA
import MonService

orb = CORBA.ORB_init(sys.argv, CORBA.ORB_ID)
ior = open("id_service.txt").read()
obj = orb.string_to_object(ior)
proxy_serveur = obj._narrow(MonService.Service)

if proxy_serveur is None:
    print "L´objet serveur n´est pas un MonService.Service!"
    sys.exit(1)
```

```
s = proxy_serveur.doubler_str("Drole.")
print s
l = proxy_serveur.doubler_long(21)
print l
```

L'exemple est volontairement simple, CORBA offre d'autres façons de gérer les servants et de les activer, de traiter le multithreading, de gérer la durée de vie des objets… et la norme définit de nombreux services optionnels. La documentation d'OmniORB offre d'autres exemples plus complets avec, par exemple, l'utilisation du service de nommage de CORBA, le Naming Service. Pour Python il existe aussi l'ORB Fnorb, antérieur à OmniORBpy, et dont le développement n'avance plus trop, mais qui a l'avantage d'être en pur Python.

**Transfert d'images par CORBA**

Si vous avez à transférer de gros tableaux d'entiers – typiquement des images – il peut être préférable de les définir en tant que `sequence<octet>` plutôt que `sequence<short/long>`. Les spécifications OMG/IDL-Python prévoient en effet de très coûteuses conversions en listes d'entiers pour les séquences de short/long, alors que les séquences d'octets sont simplement transférés via des chaînes Python (qui peuvent contenir des valeurs 0). Il vous faut par contre gérer vous-même les problèmes d'ordre des octets suivant les plateformes, mais cela permet de manipuler directement des chaînes Python et de les utiliser par exemple avec la Python Imaging Library.

Et pour les utilisateurs de Windows, le module d'extension pywin32 offre le support de la technologie d'objets distribués propriétaire COM/DCOM de MicroSoft, ce qui permet de piloter la plupart des applications sur cette plateforme ainsi que de créer des serveurs.

## DBUS

DBUS est un projet FreeDesktop de technologie de communication inter-applications et système-applications, permettant de piloter les applications Gnome et KDE 4 (KDE 3 utilisant DCOP), ainsi que de s'informer sur le système ou d'être notifié de certains évènements via des signaux DBUS.

Il existe un binding Python via un module tiers dbus et un tutoriel pour son utilisation.

# Programmation Sockets

Il s'agit là de la mise en oeuvre de l'accès le plus simple au réseau, l'accès direct aux sockets et l'envoi/réception de paquets ou de flux.

## Accès UDP

Le User Datagram Protocol (UDP) permet de transmettre des paquets sur un réseau IP en mode non connecté (pas de garantie de délivrance ni d'ordre d'arrivée). En cours réseau, on le compare généralement à l'envoi de courrier postal. Son utilisation est très simple et peut permettre la mise en place rapide d'échanges entre diverses applications.

Pour réaliser un serveur pouvant recevoir des paquets UDP, il suffit de créer un objet `socket` via le module `socket` pour de l'UDP (`SOCK_DGRAM`) sur IP (`AF_INET`), de lui associer une adresse, puis de lire les paquets qui arrivent sur le socket. Cf. Listing 13.

Note suite à un courriel de Stéphane Bortzmeyer : Il est possible d'utiliser `AF_UNSPEC` à la place de `AF_INET`, ce qui permet d'utiliser le socket sans préjuger de la couche réseau utilisée (entre autre cela permet de supporter IPv4 et IPv6).

Listing 13. Un serveur UDP

**listing13.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import socket

PORT = 17834
HOTE = "localhost"
TAILLE_MAX_PAQUET = 10240
s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
s.bind((HOTE,PORT))

while True :
    paquet,source = s.recvfrom(TAILLE_MAX_PAQUET)
    print "En provenance de",source,":"
    print repr(paquet)
    if paquet == "FIN" : break
s.close()
```

La partie cliente est encore plus simple (cf. Listing 14), il suffit simplement d'écrire sur le socket créé.

Listing 14. Un client UDP

**listing14.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import socket

PORT = 17834
HOTE = "localhost"
s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

```
while True :
    paquet = raw_input("Message (FIN pour terminer):")
    s.sendto(paquet, (HOTE,PORT))
    if paquet == "FIN" : break
s.close()
```

# Accès TCP

Le Transmission Control Protocol (TCP) est plus évolué, il assure un transport fiable en mode connecté. En cours réseau, on le compare généralement à une communication téléphonique. C'est sur ce protocole que sont réalisées la plupart des transmissions d'informations du Web.

Un serveur TCP écoute sur un socket A associé à un port spécifique connu. Lorsqu'une connexion est mise en place un autre socket B lui est dédié, qui sera utilisé pour toute la durée de cette connexion – le socket A peut alors recevoir d'autres demandes de connexion. Cf. Listing 15.

Listing 15. Un serveur TCP

**listing15.py**

```
import socket

PORT = 17834
HOTE = "localhost"
TAILLE_MAX_PAQUET = 10240
TAILLE_FILE_ATTENTE = 10

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((HOTE,PORT))
s.listen(TAILLE_FILE_ATTENTE)

donnees = ""
while True :
    connexion,source = s.accept()
    print "Connexion avec",source
    while True:
        print '-'*40
        donnees = connexion.recv(TAILLE_MAX_PAQUET)
        print "Recu:",repr(donnees)
```

```
        if "FIN" in donnees : break
        connexion.send('Coucou '+donnees)
    connexion.close()
    if "QUIT" in donnees : break
s.close()
```

Côté client (cf. Listing 16), il suffit d'établir la connexion puis de lire/écrire sur le socket.

Listing 16. Un client TCP

**listing16.py**

```
import socket

PORT = 17834
HOTE = "localhost"
TAILLE_MAX_PAQUET = 10240

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect((HOTE,PORT))

while True :
    paquet = raw_input("Message (FIN ou FIN,QUIT pour terminer):")
    s.send(paquet)
    if "FIN" in paquet : break
    retour = s.recv(TAILLE_MAX_PAQUET)
    print "Retour:",repr(retour)
s.close()
```

Pour l'exemple de client/serveur TCP, saisir `FIN` côté client pour terminer le client en cours, et `FIN,QUIT` pour terminer le client et le serveur.

# Surcouches légères

Au dessus de TCP ou UDP de nombreuses personnes ont développé des couches logicielles pour répondre à des besoins spécifiques à un domaine en définissant des protocoles ad-hoc. Et dans bien des cas il existe un module tiers Python qui permet d'utiliser ces protocoles. En voici quelques exemples.

Open Sound Control (OSC) est un protocole de communication utilisé dans le domaine du spectacle vivant, qui permet de transmettre des messages contenant des données typées, des ordres de contrôle pour des instruments, ou encore des échantillons numériques. Ce protocole est implémenté en respectant des contraintes de temps (rendu temps réel) et de facilité d'accès par des personnes dont le métier de base n'est pas nécessairement l'informatique. Les paquets OSC peuvent être transportés de différentes façons, liaison série, bus USB, et liaisons réseau

TCP ou UDP. Il existe une implémentation en Python SimpleOSC pour l'encapsulation et l'extraction de données dans des paquets OSC.

Ivy est un bus logiciel qui utilise la diffusion (broadcast) de paquets UDP pour transmettre des informations entre divers logiciels faiblement couplés. Chaque client met en place des règles de filtrage basées sur des expressions régulières afin de ne recevoir que ce qui l'intéresse. Il existe une implémentation en pur Python ainsi qu'un module d'extension basé sur la librairie C d'Ivy.

# Protocoles de l'Internet

Par les protocoles de l'Internet, on entend les façons d'accéder aux informations qui sont arrivées avec l'Hypertext Transfer Protocol (HTTP), les Uniform Resource Locators (URLs), et les différentes couches logicielles qui y ont été ajoutées.

## HTTP

Un grand nombre d'échanges sur l'Internet se font via HTTP, le protocole de base du Web. Python est très (trop) bien fourni au niveau des applications pour le Web [WEBPROG], que ça soit pour les serveurs, les frameworks applicatifs ou les systèmes de template. Il dispose aussi de nombreux outils pour le développement côté client.

Les modules standards de Python fournissent le nécessaire pour la mise en place de serveurs, ce qui peut être pratique pour rendre facilement accessible des informations via un butineur Web. Voici un exemple très simple, cf. Listing 17.

Listing 17. Un serveur HTTP très simple

**listing17.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer

class MonHandler(BaseHTTPRequestHandler) :
    def do_GET(self) :
        self.send_response(200)
        self.send_header("Content-type","text.html; charset=iso-8859-1")
        self.end_headers()
        self.wfile.write(self.contenu().encode("iso-8859-1"))
    def contenu(self) :
        return u"""<html><body>Mon contenu&hellip; qui peut être dynamique.
</body></html>"""

try :
    server = HTTPServer(('',8083),MonHandler)
```

```
    server.serve_forever()
except KeyboardInterrupt :
        server.socket.close()
```

Une fois ce script en fonctionnement, il suffit de saisir l'URL http://localhost:8083/ dans un browser Web pour récupérer le contenu de notre page servie par notre serveur.

À noter, le module **cgi**, prévu à la base pour l'utilisation de Python en tant que service Common Gateway Interface (CGI), comporte des fonctions permettant de manipuler des URLs et des en-têtes HTTP, de convertir les entités caractères (là dessus voir aussi le module htmlentitydefs)… qui sont tout à fait utilisables dans le cadre d'un serveur Web autonome. Ainsi que le module **Cookie** pour la gestion des cookies côté serveur, et le module **urlparse** pour l'analyse des URLs. Côté client, le module **urllib2** fournit les fonctions nécessaires pour pouvoir ouvrir des URLs (se faire passer pour un client Web par exemple) et récupérer ou envoyer facilement des données via HTTP, FTP… Cf. Listing 18.

Listing 18. Un client HTTP simple

**listing18.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import urllib2
reponse = urllib2.urlopen('http://www.kernel.org/')
xhtmldata = reponse.read()
for num,ligne in enumerate(xhtmldata.splitlines()) :
    print "%04d - %s"%(num,ligne)
```

En plus de la section sur urllib2 dans la documentation des librairies Python, il faut signaler le document Urllib2 - Le Manuel manquant [URLLIB2MM], qui contient de nombreux exemples décrivant l'utilisation avancée de ce module (transmission d'en-têtes HTML particuliers, simulation de formulaires, authentification…).

D'autres modules fournissent des services utiles, comme **cookielib** pour la gestion des cookies côté client, ou **webbrowser** pour directement démarrer le browser Web de l'utilisateur.

# Manipulation du HTML/XHTML

Si vous avez à traiter des documents HTML/XHTML, il existe le module **HTMLParser**, mais celui-ci peut avoir du mal à analyser les documents mal formés. Le module tiers BeautifulSoup utilise, dans ce cas, des heuristiques permettant de récupérer tout de même une structure arborescente correspondant au document.

Si vous avez à traiter des documents XML (par exemple du XHTML), correctement formés, les outils dédiés à ce langage peuvent s'avérer plus adaptés. Il existe en standard des modules xml.sax implémentant la Simple API for XML (SAX) pour un traitement par flux, et des modules xml.dom implémentant le Document Object Model (DOM) permettant de manipuler une arborescence XML directement en mémoire. Ces interfaces de programmation sont issues du monde Java / C/C++, et leur utilisation est parfois un peu lourde par rapport à d'autres modules Python, d'autres boites à outil ont donc été naturellement développés pour pythoniser la manipulation de documents XML. Il faut citer principalement ElementTree qui a été intégré en

standard dans Python 2.5. Il en existe d'autres, certaines ciblant l'adaptation de l'interface de programmation à des besoins ou à des dialectes XML spécifiques, d'autres ayant pour objectif d'intégrer des librairies externes (généralement en C/C++) pour pouvoir les utiliser directement à partir de scripts Python.

# XML-RPC

De HTTP et XML, on arrive naturellement à XML-Remote Procedure Call (XML-RPC) [XMLRPC]. C'est un protocole d'appel de procédure distante utilisant HTTP pour le transport et XML pour la représentation des données, en essayant de conserver un fonctionnement simple pour transmettre et recevoir des informations.

Python supporte XML-RPC en standard avec le module xmlrpclib, ainsi que des modules outils comme SimpleXMLRPCServer ou DocXMLRPCServer. L'association de procédures avec un serveur peut se faire soit en définissant un objet dont les méthodes seront les procédures (il est aussi possible de créer une méthode _dispatch() de l'objet qui sera chargée de router les appels), soit en effectuant des associations nom/fonction individuellement. Cf. Listing 19.

Listing 19. Un serveur XML-RPC

**listing19.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import SimpleXMLRPCServer,time

class MonService:
    def quelleHeure(self):
        return time.asctime()
    def quellePrevisionMeteo(self,jours) :
        if int(jours)%2 :
           return u"Météo: pluie"
        else :
             return u"Météo: beau temps"

serveur = SimpleXMLRPCServer.SimpleXMLRPCServer(("localhost",5123))
serveur.register_instance(MonService())
serveur.serve_forever()
```

La mise en place d'un client est des plus triviales : il suffit de créer un proxy d'accès au serveur distant, via un appel à xmlrpclib.ServerProxy(), puis d'appeler les procédures distantes sur ce proxy. Cf. Listing 20.

Listing 20. Un client XML-RPC

**listing20.py**

```
#!/bin/env python
```

```
# -*- coding: utf-8 -*-


import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:5123/")

temps = proxy.quelleHeure()

previsions = proxy.quellePrevisionMeteo(3)

print "Heure:",temps

print "Previsions meteo:",previsions
```

Si votre client est constitué d'un script JavaScript tournant dans un browser Web, il peut être intéressant de se tourner vers la représentation des données avec le JavaScript Object Notation (JSON) et d'utiliser simplement les JSON-RPC.

## SOAP

L'étape suivante est le Simple Object Access Protocol (SOAP). Normalisé par le W3C [SOAP-W3C], SOAP utilise un protocole (éventuellement HTTP) afin de transférer les données nécessaires à l'appel de procédure distante, et un représentation XML pour ces données et le résultat en retour. Il utilise abondamment les espaces de noms XML, permet de définir de nouveaux types de données construits ; c'est une version plus puissante de XML-RPC, mais aussi beaucoup plus lourde à mettre en oeuvre si l'on n'a pas des outils automatiques.

SOAP est à la base des Web Services disponibles sur le Net, où des applications décrivent leurs interfaces dans des documents Web Services Description Language (WSDL) et communiquent effectivement via SOAP.

Pour faire court (voir le site du W3C pour faire plus long), avec SOAP on a des services et des ports, des messages échangés pour réaliser des opérations. Par analogie, on peut voir les services comme des objets et les ports comme des méthodes sur ces objets, les envois de messages correspondant à des appels de méthodes distantes.

Il existe divers modules tiers pour le support de SOAP dans Python, citons le Python Web Services [PYWEBSVCS] et son module SOAPpy, le package SOAPy [SOAPY] ainsi que l'utilisation possible de ElementTree.

**La boîte à outils : twisted**

On ne peut aborder la programmation de services réseau en Python sans citer le framework de développement d'applications réseau twisted [TWISTED]. Il permet de mettre en place des serveurs pilotés par évènements en utilisant divers protocoles de transport de base : TCP et UDP, les connexions sécurisées SSL/TLS, la diffusion sur des groupes d'adresses en multicast, les sockets Unix ; ainsi que des protocoles de plus haut niveau : HTTP, NNTP, IMAP, SSH, IRC, FTP…

# Intégration de bibliothèques

À côté des exécutables tiers autonomes avec lesquels on veut communiquer il y a aussi des librairies de code compilé que l'on voudrait pouvoir utiliser directement à partir de Python.

# Codes en C/C++

Pour les librairies C/C++ il existe différentes solutions pour réaliser cette intégration, le choix dépendant en partie du nombre de fonctions auxquelles on veut pouvoir accéder à partir de Python, et en partie du couplage que l'on veut si l'on a du C++ avec des classes.

Si l'on a juste quelques fonctions d'une librairie C, disponible sous forme de module chargeable, que l'on veut pouvoir appeler, le plus simple est d'utiliser le module **ctypes**. Le Listing 21 donne un exemple d'accès direct aux fonctions fopen, fputs, fclose de la libC, où l'on spécifie explicitement les arguments en entrée et en sortie de ces fonctions. Il est possible d'utiliser les conversions automatiques automatiques que réalise ctypes entre les types C et certains types Python, mais le risque est grand de sortir alors en core-dump.

Listing 21. Utilisation de ctypes

**listing21.py**

```
#!/bin/env python
# -*- coding: utf-8 -*-

import ctypes

# Pré-chargement de la librairie.
libc = ctypes.cdll.LoadLibrary("libc.so.6")

# FILE* fopen(const char* filename, const char* mode) ;
fopen = libc.fopen
fopen.argtypes = [ctypes.c_char_p,ctypes.c_char_p]
fopen.restype = ctypes.c_void_p

# int fclose(FILE* stream) ;
fclose = libc.fclose
fclose.argtypes = [ctypes.c_void_p]
fclose.restype = ctypes.c_int

# int fputs(const char* s, FILE* stream) ;
fputs = libc.fputs
fputs.argtypes = [ctypes.c_char_p,ctypes.c_void_p]
fputs.restype = ctypes.c_int

# Utilisation
f = fopen("test-ctypes.txt","w")
fputs("Juste une chaine pour faire un essai",f)
```

```
fclose(f)
```

Il est possible de spécifier d'autres genres de paramètres avec ctypes : indiquer un passage par référence, définir des structures ou des unions, définir des tableaux, gérer des zones mémoire alloués dynamiquement (et modifiables), définir des fonctions callback (voir le tutoriel), accéder à des variables exportées par la librairie…

C'est un outil très pratique lorsque l'on n'a qu'un nombre limité fonctions à définir, ou lorsque l'on n'a pas nécessairement de compilateur C installé sur la machine. Mais dès que l'on a des besoins plus poussés : interfaçage avec du C++, création d'adaptateur pour Python et pour d'autres langages… l'utilisation de swig, de boost-python ou de sip deviens incontournable. Ceux-ci permettent de réaliser des adaptateurs pour des librairies complètes, en spécifiant des façons de procéder pour les types définis, pour les traitements d'erreurs, pour les callbacks, etc …

Il faut aussi citer Pyrex, qui permet générer du code C à partir d'un sous-ensemble de code source Python (pouvant intégrer des appels à des codes C existant), et de créer des modules d'extension compilés. Cela peut être par ailleurs un bon moyen d'accélérer certains traitements sans avoir à écrire un module complet en C.

Attention, même si les outils cités facilitent le travail, la création d'adaptateurs Python pour des bibliothèques n'est pas toujours triviale, aussi avant d'essayer de le faire vous-même, vérifiez bien que quelqu'un n'a pas déjà fait le travail. L'article Scripting C with Python [SCRIPTINGC] peut vous permettre de faire un choix sur la technologie la plus adaptée à vos besoins.

## Autres langages

Il existe des outils pour lier Python avec d'autres langages que le C. Pour Java : Jython (interpréteur Python écrit en Java), le Java to Python Integration (JPYPE) et le Java Embedded Python (JEPP) ainsi que le projet java2python permettant de traduire du code Java vers Python. Pour les amateurs de C# il existe IronPython, un interpréteur Python basé sur le Common Language Runtime (CLR) de Microsoft. Pour Fortran, il existe une présentation [FORTRAN] donnant un apperçu des trois solutions disponibles : Pyfort, f2py et Forthon.

Et il existe des outils d'interfaçage avec d'autres langages encore : Lisp, Ocaml, Delphi, etc …

# Lecture/écriture de fichiers

Un dernier problème que nous aborderons dans cet article, la manipulation des différents formats des fichiers générés par les processus informatiques. Nous ne donnerons ici que des pistes vers des outils dédiés qui évitent d'avoir à ré-inventer la roue et l'engrenage.

Outre l'accès simple en mode texte brut, Python peut rendre transparent l'encodage des fichiers avec la fonction `codecs.open()` qui retourne un objet fichier manipulant des chaînes unicodes au niveau de l'utilisateur et les convertissant de/vers l'encodage désiré au niveau du fichier.

Pour les utilisateurs de données textuelles formatées en tableaux (avec séparateurs de colonnes et de lignes), le module **`csv`** permet d'automatiser l'extraction des données.

Si l'on désire accéder à des fichiers binaires spécifiques, il est possible d'ouvrir les fichiers en mode binaire et d'utiliser les modules **`struct`**, **`array`**, et **`ctypes`** (ce dernier a été intégré dans Python 2.5) qui permettent de spécifier les structures de données et de manipuler les informations correspondantes.

Pour les fichiers d'images bitmap (et pour la manipulation de ces images), la Python Imaging Library [PIL] supporte la lecture/écriture d'un certain nombre de formats de fichiers et peut être étendue pour en gérer d'autres. Concernant les fichiers SVG, les outils XML sont normalement suffisants.

Pour les documents Open Office, ce sont des archives compressées dont il est possible d'extraire les fichier contenus avec le module zipfile. À partir de là on peut accéder aux données qu'ils contiennent (du XML pour les documents principaux). Il est possible de manipuler le XML directement avec les outils cités précédemment, d'utiliser les py-odftools de Google, ou encore d'utiliser le module tiers oopy. On peut aussi scripter directement OpenOffice en utilisant PyUNO.

Pour les documents Microsoft Excel, il existe divers projets : pyExcelerator, xlrd et pyXLWriter. Et pour la génération de documents Rich Text Format (RTF), il y a PyRTF. Pour le scripting des applications Microsoft Office, cela passe par COM/DCOM avec le module tiers pywin32.

Pour de nombreux formats issus du monde scientifique il existe des modules tiers, par exemple pour le Hierarchical Data Format il y a le module pyhdf et l'outil pytables, et certaines communautés ont développés des packages complets contenant entre autres les outils de manipulations de leurs formats de données habituels, par exemple biopython, SciPy et ScientificPython.

# Conclusion

Cet article a essayé de faire un tour non exhaustif des différentes façons par lesquelles Python peut s'interfacer avec d'autres outils. Il existe bien d'autres protocoles, d'autres façons de faire communiquer des applications, d'autres modules dédiés disponibles sur l'Internet. Nous n'avons pas traité par exemple des accès aux bases de données, du pilotage d'interfaces graphiques par du scripting (DBUS, COM/DCOM, pywinauto…), du pilotage de liaisons série.

J'espère que cette lecture vous aura fait découvrir les possibilités de Python, la facilité et la rapidité avec laquelle on peut réaliser des développements à-façon, et vous aura donné l'envie d'essayer ce langage.

Et si vous achoppez sur certains problèmes, vous pouvez compter sur la communauté des Pythoneurs toujours prête à répondre aux questions pertinentes (usenet anglophone [CLP] et francophone [FCLP], liste de diffusion python-fr [PYTHONFR], Association Francophone Python [AFPY]…), sans oublier les ressources que l'on peut trouver sur le Net avec les moteurs de recherche.

**Sur l'auteur**

Laurent Pointal L'auteur est informaticien pour deux groupes de recherche au sein du laboratoire LIMSI/CNRS. Il y exerce depuis trois ans dans le domaine de la communication homme-machine. Il a travaillé précédemment pendant un peu plus de dix ans au pilotage d'expériences scientifiques sur les installations du très grand équipement du laboratoire LURE. Ses activités l'ont amené à aborder certaines techniques permettant d'interfacer des logiciels issus de diverses sources, afin de réaliser des systèmes communiquants complexes.

**Liens sur Internet**

Liens pour certains modules et documents cités dans l'article :

- [AFPY] – http://afpy.org/
- [CLP] – http://groups.google.fr/group/comp.lang.python/
- [EASYINSTALL] – http://peak.telecommunity.com/DevCenter/EasyInstall
- [EX-RPC] – http://www.cs.rpi.edu/~hollingd/netprog/code/rpc/simple/

- [FCLP] – http://groups.google.fr/group/fr.comp.lang.python/
- [FORTRAN] – http://calcul.math.cnrs.fr/IMG/pdf/python-fortran.pdf
- [OMNIORBPY] – http://omniorb.sourceforge.net/
- [ORBBASICS] – http://www.omg.org/gettingstarted/orb_basics.htm
- [PIL] – http://www.pythonware.com/products/pil/
- [PYPI] – http://www.python.org/pypi
- [PYRPC] – http://thomas.enix.org/Blog-20051221211433-Libre et http://thomas.enix.org/pub/pyrpc/
- [PYTHON] – http://www.python.org/
- [PYTHONFR] – https://www.aful.org/wws/info/python
- [PYWEBSVCS] – http://pywebsvcs.sourceforge.net/
- [SCRIPTINGC] – http://www.suttoncourtenay.org.uk/duncan/accu/ScriptingC.html
- [SOAP-W3C] – http://www.w3.org/2002/07/soap-translation/soap12-part0.html
- [SOAPY] – http://soapy.sourceforge.net/
- [TWISTED] – http://twistedmatrix.com/trac/
- [URLLIB2MM] – http://www.voidspace.org.uk/python/articles/urllib2.shtml et en VF http://www.voidspace.org.uk/python/articles/urllib2_francais.shtml
- [WEBPROG] – http://wiki.python.org/moin/WebProgramming
- [XMLRPC] – http://www.xmlrpc.com/spec

Et pour aller plus loin :

- Le site de Python (téléchargement, documentations, wiki, dépôt de modules…) – http://www.python.org/
- Un livre en ligne pour apprendre Python à partir d'exemples (traduit dans différentes langues) – http://diveintopython.org/
- La page de liens sur Python de l'auteur – http://www.limsi.fr/Individu/pointal/python.html

**Liste des listings**