Échanges de données pour un traitement distribué : communication par réseau ou entre langages

Jean-Michel Friedt, FEMTO-ST département temps-fréquence, Besançon, 5 septembre 2023

Comment tirer le meilleur parti des divers langages à notre disposition, entre vitesse de prototypage de Python ou GNU Octave et vitesse d'exécution de C ? Nous allons échanger des données entre des fonctions écrites dans ces langages, soit par socket du réseau soit par bibliothèques dynamiques.

Notre objectif dans cet exposé est de voir comment tirer le meilleur parti des trois langages d'acquisition et de traitement de signaux numériques que nous utilisons au quotidien : le **C** pour la vitesse et la compacité du langage compilé, **GNU Octave** pour un langage interprété implémentant les fonctions d'algèbre linéaire issu de Matlab, et **Python**. Alors que chaque langage s'utilise indépendamment, nous abordons l'échange d'informations entre ces langages dans un concept de traitement centralisé ou décentralisé par une communication par réseau informatique.

La communication entre systèmes numériques, embarqués ou non, semble être devenue la norme à en oublier les principes sous-jacents. Alors qu'historiquement Internet avait pour vocation d'uniformiser les communications entre les nombreux sous-réseaux qui fleurissaient dans les années 1960 et 1970 [1] selon une architecture rationnelle et hiérarchique avec une obsession de décentralisation liée à éviter toute dépendance en un unique nœud centralisant les échanges (penser "guerre froide" et "attaque nucléaire" pour un réseau financé par la DARPA), aujourd'hui pléthore de couches additionnelles viennent s'empiler au dessus de Internet Protocol (IP) pour amener de nouvelles "fonctionnalités". Nous avons vécu heureux avec IP, TCP et UDP [2], voir des *raw sockets* sur Ethernet quand nous n'avions pas besoin de routage de paquets dans une connexion point à point pour commander un système embarqué (`https://sourceforge.net/projects/proexgprcontrol/`), donc ces sur-couches applicatives semblent aussi superflues que gourmandes en ressources, voir instables dans le temps avec les changements constants d'API que nous avons une fois de plus vérifiés en rédigeant cette prose.

Néanmoins, nous allons ici explorer trois couches applicatives propageant des signaux au-dessus d'IP que sont XMLRPC, ZeroMQ et MQTT qui se chargent d'organiser les données lors des transactions. Au-delà des échanges au travers des *sockets* qui décrivent les interfaces de communication compatibles avec une liaison Internet, nombre de passerelles existent entre les langages de programmation pour échanger les données et tirer le meilleur parti de chaque langage – rapidité d'exécution du langage compilé (C), facilité de prototypage du langage interprété (Python et donc GNU Radio, GNU Octave). Même si cet exposé se veut agnostique de toute application, le traitement de signaux radiofréquences tel que proposé par GNU Radio fournit la ligne directrice de nos recherches et nous nous appuierons sur le générateur de code Python GNU Radio Companion pour sélectionner les technologies exposées. La nature des données que nous désirons échanger est un flux continu de données radiofréquences acquises par un récepteur matériel et dont les informations doivent être traitées, localement ou à distance, pour un système de calcul implémenté dans le langage le plus approprié.

L'exposé ne se veut pas être un traitement exhaustif de toutes les passerelles entre langages – ne maîtrisant à peu près que C, Python et GNU Octave que nous exploitons au quotidien pour le traitement numérique de signaux échantillonnés en temps discret – mais un guide pour présenter comment chaque langage peut contribuer à un système global complexe avec des parties de traitements simples et efficaces en vue de rapidité (C) et du prototypage en langage interprété plus permissif sur les typages au détriment de l'efficacité (Fig. 1). Néanmoins, ces passerelles introduisent une nouvelle dépendance envers les infrastructures de communication avec les risques inhérents de rupture des APIs et donc de pertes de fonctionnalités indépendantes de notre contrôle.

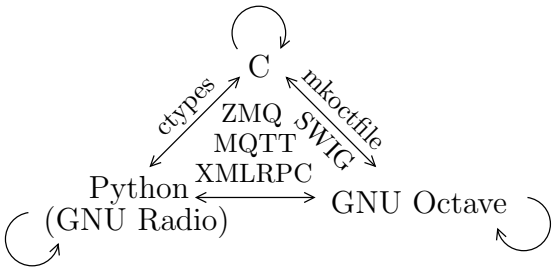


Figure 1: Interactions entre C, GNU Octave et Python natif ou produit par GNU Radio Companion, et outils associés qui seront abordés dans cet article.

1 TCP/IP et UDP/IP

Commençons par le commencement : l'interopérabilité des ordinateurs connectés sur un réseau selon les préceptes d'Internet.

La hiérarchie des couches d'abstraction est formalisée par le standard OSI (*Open Systems Interconnection*) qui peut paraître arbitraire jusqu'à ce qu'on tente de l'implémenter dans un cas pratique [3] pour découvrir que chaque couche implique une expertise et des connaissances techniques différentes. Ainsi, la couche la plus basse – matérielle – sera facilement abordée par un électronicien quand la couche applicative – la plus élevée – implique nombre de concepts informatiques abstraits. Entre les deux, les informations doivent être assemblées en paquets, routées d'une machine à une autre afin que des sauts de puce permettent d'acheminer les informations de la source à la destination, les interlocuteurs doivent s'accorder sur la représentation des informations et sur les divers services susceptibles de traiter les informations (port des sockets). Le principe de la hiérarchie OSI est que chaque couche supérieure suppose que les couches inférieures ont été implémentées et sont fonctionnelles. Ainsi, pas de routage de paquets par TCP en mode connecté qui garantit l'intégrité des échanges ou UDP dans lequel un serveur jette en pâture des informations aux clients susceptibles, ou non, de les recevoir, sans le contrôle d'accès et conversion d'adresse physique en adresse logicielle par ARP (Fig. 2).

7 Applicative (HTTP, SMTP, NTP)
6 Présentation (ASCII, HTML, SSL, MQTT, XML)
5 Session (RPC, socket)
4 Transport (TCP,UDP)
3 Réseau (ARP, IP)
2 Liaison (Ethernet)
1 Physique (10BASE-T)

Figure 2: Hiérarchie des couches OSI décrivant les services nécessaires à une communication par réseau informatique. Nous nous intéresserons ici aux couches les plus élevées.

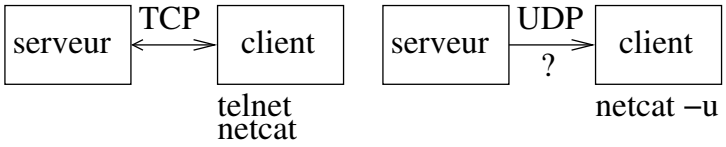


FIGURE 3 – Notion de serveur – celui en attente de fournir un service – et de client – celui qui requiert un service, échangeant des données soit en mode connecté pour garantir les échanges (TCP) ou en mode datagramme sans validation des transactions (UDP).

Au-dessus de IP qui traduit les adresses physiques en adresses logiques, deux modes de communication sont TCP qui garantit les transactions (mode connecté) et dans lequel le serveur bloque ses échanges tant qu'ils ne sont pas acquittés avec le client, et UDP (*datagram*) dans lequel le serveur émet des données qui sont ou non reçues par un client dans un ordre qui n'est pas garanti en fonction du routage des paquets le long du trajet entre le serveur et le client. Dans ce second cas, le serveur exécute ses opérations indépendamment de toute connexion d'un client pour recevoir ou non les données acquises : nous serons friands de ce mode de communication pour mettre en œuvre un RADAR par exemple, qui peut librement commander un récepteur de radio logicielle et déplacer les antennes tandis qu'un client reçoit les données lorsque les conditions sont favorables. Des clients universels permettant de facilement tester les serveurs sont `telnet` et `netcat` pour TCP, le second avec son option `-u` pour recevoir des informations en UDP. On pourrait se demander pourquoi ne pas toujours utiliser TCP qui garantit les échanges de données ? Un échange TCP nécessite de mémoriser les paquets transitant qui pourraient être corrompus ou dont l'ordre a été échangé par un changement des règles de routage sur le réseau au cours de la communication : une pile TCP/IP est très lourde à implémenter et gourmande en ressources mémoires, quand un échange UDP tient en quelques lignes en l'absence de tout acquittement.

Le concept de socket est au cœur d'un système unix qui ne saurait fonctionner sans, tel que le définit la norme POSIX [4]. Il suffit de lancer la commande `ss` (*socket statistics*) pour constater les plusieurs centaines de tuyaux de communication ouverts sur un système GNU Linux même déconnecté d'Internet mais échangeant des informations entre ses divers services. Ainsi, une socket ne transporte pas nécessairement des données d'un ordinateur à un autre au travers d'un réseau informatique mais peut échanger des informations entre processus exécutés sur un même ordinateur : il s'agit alors d'*InterProcess Communications* ou IPC. Le contrôle à distance de processus est un cas particulier d'IPC nommé RPC pour Remote Procedure Call. La question se portera donc sur la représentation des données échangées et leur encapsulation afin que les interlocuteurs s'accordent sur leur représentation.

En effet, le serveur – rappelons que le **serveur** est en attente constante pour fournir un service à des **clients** (Fig. 3) qui se connectent ponctuellement pour accéder à ce service – TCP de base s'écrit en C

```
#include <sys/socket.h>
#include <resolv.h>
#include <unistd.h>
#include <strings.h>
#include <arpa/inet.h>
```

```c
#define MY_PORT          9999
#define MAXBUF           1024

int main()
{int sockfd;
 struct sockaddr_in self;
 char buffer[MAXBUF];

 sockfd = socket(AF_INET, SOCK_STREAM, 0);  // type de socket
 bzero(&self, sizeof(self));
 self.sin_family = AF_INET;
 self.sin_port = htons(MY_PORT);
 self.sin_addr.s_addr = INADDR_ANY;
 bind(sockfd, (struct sockaddr*)&self, sizeof(self));
 listen(sockfd, 20);
 while (1)
 {struct sockaddr_in client_addr;
  int taille,clientfd;
  unsigned int addrlen=sizeof(client_addr);
  clientfd = accept(sockfd, (struct sockaddr*)&client_addr, &addrlen);
  taille=recv(clientfd, buffer, MAXBUF, 0);
  send(clientfd, buffer, taille, 0);
  close(clientfd);
 }
 close(sockfd);return(0);  // Clean up (should never get here)
}
```

ou en Python

```python
import socket
import string
while True:
    sock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('127.0.0.1', 4242))
    sock.listen(1)
    conn, addr = sock.accept()
    with conn:
        print('connected from ',addr)
        while True:
            data=conn.recv(1)
            if data:
                data=data.decode()
                print(data)
                if 'q' in data:
                    sock.shutdown(socket.SHUT_RDWR)
                    sock.close()
                    break
```

pour connecter un serveur à une socket (`bind`), attendre une connexion d'un client (`listen`), et échanger des informations (`recv`, `send`). Ces séries d'octets n'ont aucune structure et n'ont de sens que parceque les deux interlocuteurs se sont mis d'accord à l'avance sur leur organisation. Ces exemples restent cependant utiles car par exemple dans GNU Radio, le serveur en Python proposé ci-dessus est lancé dans un thread indépendant par

```python
import threading
import mon_serveur
threading.Thread(target=mon_serveur,args=(self,)).start()
```

dans un `Python Snippet` exécuté lors de l'initialisation de l'ordonnanceur. Le passage de l'argument `self` donne accès à toutes les fonctions définies par GNU Radio et en particulier les *setter* et *getter* associés à chaque variable déclarée dans la chaîne de traitement : ainsi, le thread peut faire appel à `self.get_var()` et `self.set_var()` si la variable `var` a été définie afin d'en modifier le contenu. Nous utilisons intensivement ce mécanisme lorsqu'un client doit balayer un paramètre d'une liaison radiofréquence, par exemple la fréquence porteuse du signal.

Ainsi dans GNU Octave, un client de la forme

```
sck=socket(AF_INET, SOCK_STREAM, 0);
server_info=struct("addr","127.0.0.1","port",4242);
connect(sck,server_info);
send(sck,'s');  % start
```

se connectera au port 4242, le même port auquel le serveur s'est lié auparavant, sur le même ordinateur exécutant le serveur (127.0.0.1) pour envoyer un ordre, par exemple la lettre "s" qui pourrait informer du début d'une séquence de traitements. Ici une liaison connectée TCP indiquée par `SOCK_STREAM` indique que les transactions sont garanties par un acquittement de chaque échange, au contraire d'une transaction non connectée ou *datagram* selon UDP selon laquelle une information est transmise sans garantir sa réception. Ces deux modes seront utilisés selon que l'information doive être organisée et acquittée

(TCP) ou simplement émise vers des clients qui sont ou non à l'écoute et dont la perte n'a que peu de conséquences (par exemple un flux de données venant d'un récepteur radiofréquence).

Python et GNU Octave sont deux langages interprétés que nous faisons souvent cohabiter, Python pour sa souplesse d'accès aux ressources matérielles et son utilisation pour connecter ensemble les blocs de traitement de GNU Radio produits par des chaînes d'analyse de signaux conçues dans GNU Radio Companion, et Octave pour la facilité de son implémentation matricielle d'algorithmes d'algèbre linéaire selon le langage issu de Matlab. Le programmeur plus souple que l'auteur en Python n'aura aucune difficulté à traduire les algorithmes proposés dans GNU Octave vers NumPy sans devoir passer par `https://numpy.org/doc/stable/user/numpy-for-matlab-users.html`. Ainsi dans cette présentation, nous nous efforcerons d'échanger non-seulement des scalaires mais surtout des vecteurs de données entre Python et Octave.

De gauche à droite ou de droite à gauche : ordre des octets

Mis à part certaines documentations d'Airbus Defence pour l'Agence Spatiale Européenne [5], il semble assez évident en occident de placer les bits de poids fort à gauche et les bits de poids faible à droite, en écrivant donc mille deux cent trente quatre comme 0x4d2 ou 0b10011010010 tel que l'indique `dec2hex` et `dec2bin` de GNU Octave. La situation est moins limpide pour l'agencement des octets pour une grandeur codée sur 8 bits : historiquement, la DARPA américaine et les pays occidentaux ayant dominé le développement d'Internet, il semble naturel de placer les octets de poids fort à gauche et de poids faible à droite, et donc d'écrire 0x4d2 comme 0x04 en octet d'adresse la plus faible ("à gauche") et 0xd2 en octet d'adresse la plus élevée ("à droite") pour que l'affichage du contenu de la mémoire de son adresse la plus faible vers l'adresse la plus élevée affiche 0x04 0xd2. Comme historiquement Internet fut développé [1] par BBN sur des architectures Honeywell et IBM puis Sun Microsystems sur SPARC et Motorola, il fut naturel d'adopter cet ordre sélectionné par ces architectures de processeurs, nommé *big endian* (octet de poids fort à l'adresse faible) pour faire transiter les informations codées sur plusieurs octets sur le réseau. Cependant, Intel eut l'idée de placer l'octet de poids faible à l'adresse la plus faible, un choix qui devient logique quand des opérations arithmétiques sont faites sur architecture CISC donc d'instructions de longueur variable : l'unité arithmétique et logique lit l'instruction (l'opcode), commence à charger les arguments, et si le premier argument lu est les unités, alors l'opération arithmétique peut commencer pendant la lecture de l'octet suivant contenant les dizaines puis éventuellement les centaines et les milliers, propageant donc la retenue lors de chaque sous opération. Cette organisation de l'octet de poids faible à l'adresse la plus basse est nommée *little endian.* Au contraire une organisation big endian nécessite de lire le nombre entier en commençant par les milliers avant de finir par les unités pour commencer l'opération arithmétique.

Même si aujourd'hui les processeurs Intel/AMD little endian dominent le marché de l'ordinateur personnel grand public, Internet tout comme Java et donc Android restent big endian. Échanger des informations entre les deux architectures nécessite de s'accorder sur l'ordre des octets : c'est le sens des instructions `htons` (ou `htonl` pour 4 octets) pour organiser deux octets (un short en C) dans le bon ordre, de l'hôte (*host*) vers le réseau (*network*) et réciproquement à l'autre bout. Ces macros sont définies dans `/usr/include/netinet/in.h` sous GNU/Linux comme une identité ou un échange des octets selon l'architecture utilisée (`#if __BYTE_ORDER==__BIG_ENDIAN`). Alors que cette opération doit être effectuée explicitement en C, elle sera implicite dans les infrastructures que nous verrons ci-dessous (0MQ ou MQTT [6] parlent de *network byte order* pour l'organisation de leurs champs codés sur plus d'un octet, mais le contenu lui même n'est qu'un paquet d'octets que le développeur doit organiser convenablement) voir inexistante pour les transactions en ASCII (XMLRPC) dans laquelle l'ordre des arguments est celle des chaînes ASCII échangées.

Noter que dans les exemples que nous traiterons ci-dessous, les échanges se font au sein du même ordinateur (127.0.0.1) et un processeur étant cohérent avec lui même, toute erreur sur l'ordre des octets à l'émission est corrigée à la réception (deux erreurs qui se compensent). En production, il est prudent de communiquer avec une machine d'endianness opposée pour identifier les sources potentielles de dysfonctionnement – Java est pour cela redoutable et nous nous garderons bien de l'inclure dans nos cas de tests tant ce langage nous est étranger.

Au contraire de TCP qui garantit les transactions, UDP émet juste des paquets vers qui veut les entendre. Ainsi, Fig. 4 propose une chaîne de traitement GNU Radio Companion qui se contente d'émettre des nombres à virgule flottante en simple précision (*float* pour un symbole orange dans GNU Radio Companion) tandis qu'à l'autre bout GNU Octave (gauche) ou Python (droite) exécute

```
pkg load instrument-control
while (1)
  s=udpport("LocalHost","127.0.0.1","→
      ↪LocalPort",2000);
  val=read(s,4000);
  vector=typecast(val,"single");
  plot(vector); pause(0.1)
  clear("s")
end
```

```
import socket
import array
from matplotlib import pyplot as plt
s=socket.socket(socket.AF_INET,socket.→
    ↪SOCK_DGRAM)
s.bind(("127.0.0.1", 2000))
while True:  # 4000 bytes=1000 float
  val,addr=s.recvfrom(4000)
  vector=array.array('f',val)
  plt.plot(vector)
  plt.show()
```

afin d'ouvrir la socket en mode UDP sur le port 2000 de l'ordinateur local (sur lequel GNU Radio écrit), affiche les données acquises après avoir converti le paquet d'octets en un vecteur de réels en virgule flottante, et fermer la socket. Cette ouverture/fermeture perpétuelle de la socket peut sembler discutable mais c'est la meilleure façon que nous ayons trouvée de garantir que les données traitées sont les dernières transmises et pas de vieilles données qui traînent dans une mémoire tampon. Dans le cas d'UDP, même si certaines données sont ainsi perdues, ce n'est pas grave puisque nous garantissons d'obtenir un vecteur du bon nombre de données fraîches. La meilleure façon de synchroniser les données acquises avec un évènement physique tel que la rotation d'une antenne est d'envoyer un ordre vers GNU Radio pour effectuer la commande, attendre le temps nécessaire à ce que la commande soit achevée ou idéalement un acquittement par une communication TCP en réponse de la requête, puis ouvrir la socket UDP et capturer le nombre de données voulues dans cette configuration, puis répéter pour toutes les configurations envisagées – pour un RADAR à ouverture synthétique par exemple, avec toutes les positions successives des antennes.

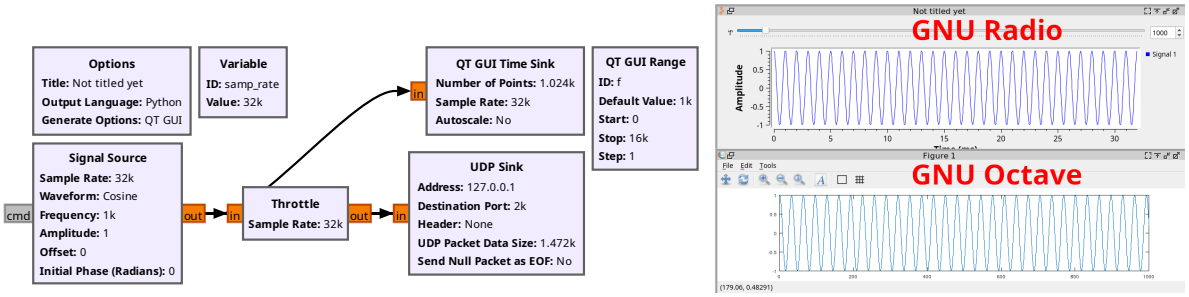


FIGURE 4 – Gauche : chaîne de traitement alimentant une socket UDP sur l'ordinateur local (127.0.0.1) sur le port 2000 en vue de partager avec tout programme susceptible de l'écouter pour traiter ce flux de données. Droite : comparaison de la sortie graphique de l'oscilloscope `Time Sink` de GNU Radio avec la sortie graphique de GNU Octave qui a converti le paquet d'octets reçu en `float` par `typecast()`.

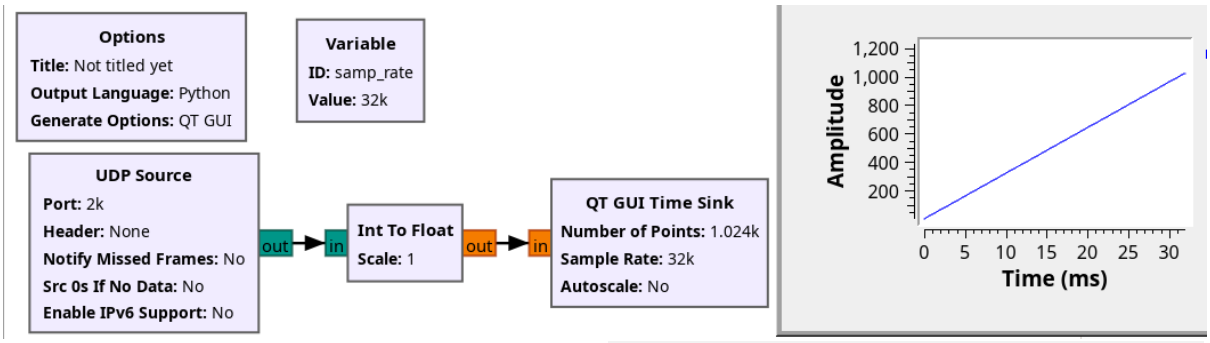
Lors du lancement du programme GNU Radio émettant les données sur le port 2000 de la socket locale (127.0.0.1) en mode UDP, nous pouvons valider l'émission des données par `netcat` au moyen de `nc -l -p 2000 -u 127.0.0.1` avec `-l` pour écouter et `-u` en UDP.

De la même façon, nous pourrons bénéficier de GNU Radio pour proposer une chaîne de traitements pour des signaux acquis par une interface diffusant ses informations par UDP et en exploitant le UDP Sink tel que proposé en Fig. 5, cette fois en échangeant des entiers codés sur 4 octets (32 bits) tel que l'indique l'icône verte dans la chaîne de traitement GNU Radio Companion. Dans cet exemple, nous envoyons une rampe par GNU Octave (`val=int32([k:k+1024]);v=typecast(val,'uint8');` à gauche) ou en Python (`numpy.arange(0+k,1024+k,dtype=numpy.int32)` à droite) mais nous pourrions bien entendu envoyer toute séquence de mesures, par exemples acquises par RS232 depuis un instrument.

Basé sur ces couches basses de la hiérarchie OSI, nous allons désormais explorer quelques mécanismes organisant les transactions et facilitant l'accès aux clients des interfaces exposées par les serveurs.

2 XMLRPC

Un client-serveur TCP/IP ou UDP/IP impose de s'accorder sur le protocole d'échange des informations entre client et serveur. Afin d'ordonner ces transactions en les encapsulant dans un format aisément traité automatiquement, il peut sembler judicieux d'annoncer la nature du service requis et la variable associée : dans l'exemple précédent, seul le concepteur sait que la variable `var` existe avec ses fonctions de lecture et de définition associées, et un client externe ne peut avoir connaissance de la liste des fonctions disponibles. Il semble naturel d'encapsuler les données dans des messages, et donc d'encapsuler dans une syntaxe aisément décodable par un ordinateur que sont par exemple JSON ou XML, au détriment




```
pkg load instrument-control
s=udpport();
k=0;
while (1)
  val=int32([k:k+1024]);
  v=typecast(val,'uint8');
  write(s,v,"127.0.0.1",2000);
  k=k+1;
  if (k==1000) k=0;end
end
```

```
import socket
import numpy
k=0
sock = socket.socket(socket.AF_INET, →
    ↪socket.SOCK_DGRAM)
while True:
  val=numpy.arange(0+k,1024+k,dtype=numpy.→
      ↪int32)
  sock.sendto(val, ("127.0.0.1", 2000))
  k=k+1
  if (k==1000):
    k=0
```

FIGURE 5 – En haut : chaîne de traitement GNU Radio chargée de récupérer un flux de données arrivant sur le port 2000 de l'ordinateur qui exécute le script, et en affiche le contenu. En bas : script GNU Octave (gauche) et Python (droite) pour générer des rampes et les envoyer sur port UDP pour être intrprétées comme des séquences d'entiers codés sur 32 bits.

bien entendu d'une taille de messages accrue et une dépendance aux bibliothèques de décodage de ces messages.

Dans l'implémentation de XMLRPC, les appels aux fonctions distantes (Remote Procedure Call) sont encapsulées dans des messages XML tel que décrit dans [7]. De cette façon, un client se contente d'informer le serveur de quel service (fonction) il désire bénéficier afin d'en modifier le contenu (valeur de la variable). Il suffit de `apt-cache search xmlrpc` sous Debian/GNU Linux pour se convaincre de la multitude de langages implémentant ce protocole, et en particulier Python. Dans ce langage, un client s'écrit en quelques lignes

```
from xmlrpc.client import ServerProxy
s=ServerProxy('http://localhost:8080')
s.set_freq(5000)
```

pour modifier la configuration du serveur que nous avons définie dans GNU Radio au moyen de la chaîne de traitement proposée en Fig. 6. Dans cette chaîne de traitement, la fréquence du signal issu du bloc `Signal Source` est une variable `freq` et nous constatons l'effet de la commande à distance de `freq` par le changement de la période de la sinusoïde sur la sortie graphique temporelle.

Afin de tester le bon fonctionnement du serveur, il n'est même pas utile de programmer une ligne puisque la commande shell `xmlrpc` est fournie dans le paquet `libxmlrpc-core-c3-dev` de Debian/GNU Linux, avec l'incantation `xmlrpc localhost:8080 set_freq i/1664` qui modifie la variable `freq` pour lui assigner la valeur entière (`i`) de valeur 1664. Alternativement une chaîne de caractères peut être transmise en préfixant l'argument de "s" ou un nombre à virgule flottante par "d" (les types sont décrits dans `man xmlrpc`).

D'après les spécifications de XMLRPC [7] nous pouvons forger le message au format XML, ici pour envoyer la valeur 1664 prise comme un entier codé sur 4 octets pour définir la variable `freq` au travers de son setter `set_freq` :

```
curl -X POST -H 'Content-Type: text/xml' -d '<methodCall>\n<methodName>set_freq</methodName>\
\n<params><param><value><i4>1664</i4></value></param></params>\n</methodCall>'\
'http://localhost:8080/RPC2'
```

Ainsi, tout langage qui ne supporte pas XMLRPC mais permet de communiquer par HTTP selon la méthode POST peut communiquer avec un serveur. Malheureusement, nous ne sommes pas parvenus à atteindre ce résultat dans GNU Octave, que ce soit à cause de la difficulté à s'interfacer avec des bibliothèques externes implémentant un protocole (divers auteurs annoncent se lier aux bibliothèques Java depuis GNU Octave pour faire appel à leur implémentation de XMLRPC mais nous n'avons pu reproduire ce résultat) ou d'un dysfonctionnement de l'implémentation de `webwrite` qui ne peut remplir

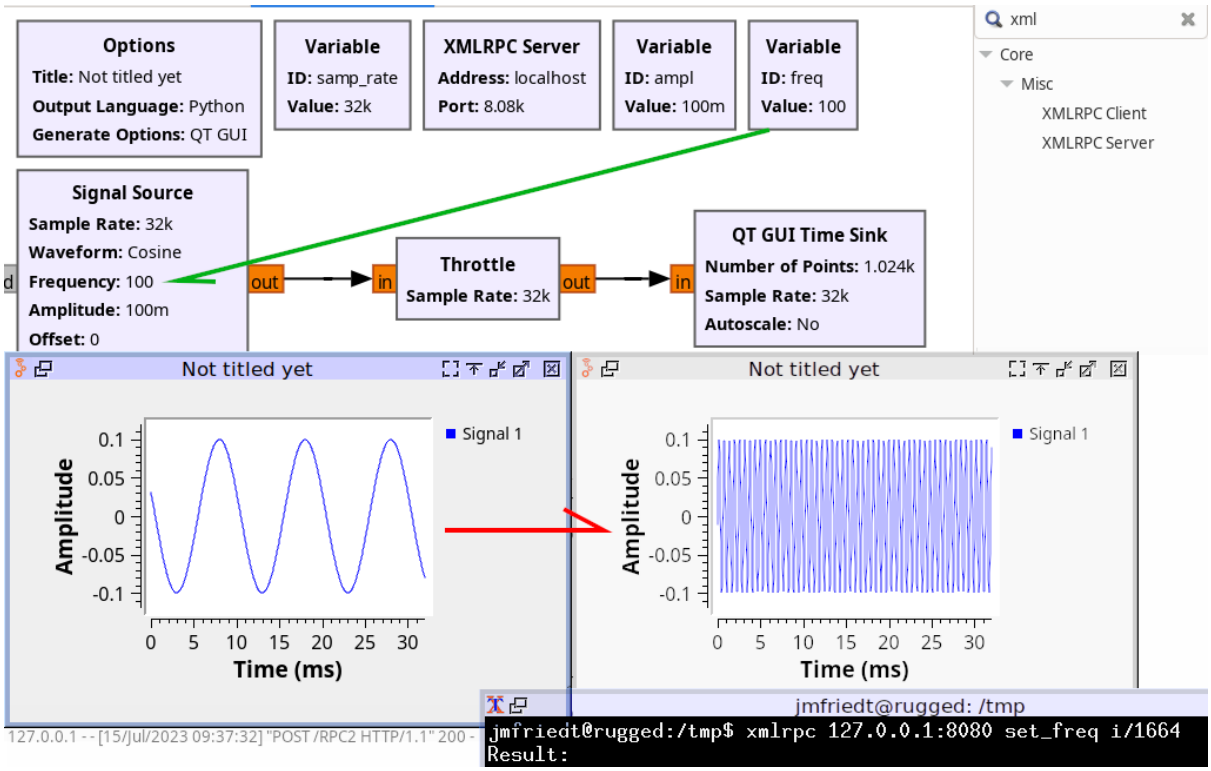


FIGURE 6 – Reconfiguration d'un paramètre d'une chaîne de traitement GNU Radio – donc Python – au moyen d'une commande émise depuis le shell. La variable `freq` définit la fréquence de la source de signaux de forme sinusoïdale, et expose donc la méthode `set_freq` à laquelle nous faisons appel depuis le shell par `xmlrpc` (bas-droite). Le serveur Python acquitte de la réception de la commande (bas-gauche) dans la console de GNU Radio Companion, et la fréquence est effectivement modifiée (flèche rouge) lors de l'émission de la commande.

le champ "data" avec un message XML d'une requête POST tel que décrit à `http://savannah.gnu.org/bugs/?56624`. Nous verrons plus loin (section 7) que nous avons été poussés à apprendre à lier des fonctions C à GNU Octave, offrant donc une solution pour implémenter tout protocole de communication et l'exposer à GNU Octave.

Pour le moment nous ne sommes pas plus avancés pour exposer les variables utilisées dans une chaîne de traitement GNU Radio puisque la requête à la méthode `listMethods` du client selon

```
import xmlrpc.client
proxy = xmlrpc.client.ServerProxy("http://localhost:8080/")

for method_name in proxy.system.listMethods(): # liste des fonctions
    if (method_name.find("set_")>=0):          # expos'ees par le serveur
        print(method_name)

try:
    setampl=proxy.set_ampl(0.2)      # echec (pas de variable ampl)
except xmlrpc.client.Fault as err:
    print("Unsupported function")
try:
    setfreq=proxy.set_freq(200)     # succes, freq redefinie
except xmlrpc.client.Fault as err:
    print("Unsupported function")
```

refuse de nous fournir la liste des variables. Ainsi, seul un essai des diverses fonctions en interceptant les erreurs (`try: ... except:`) permet de tester si une variable existe ou non : dans l'exemple ci-dessus, `set_freq` est un succès mais `set_ampl` échoue puisque l'amplitude n'est pas une variable définie dans la chaîne de traitement.

Afin d'exposer la liste des méthodes, `https://docs.python.org/3/library/xmlrpc.server.html` enseigne que nous devons activer cette fonctionnalité dans le serveur. Pour ce faire, nous ajoutons dans GNU Radio un Python Snippet contenant la commande `self.xmlrpc_server_0.register_introspection_functions()` qui active la capacité du serveur `xmlrpc_server_0` (ID du block associé) à fournir l'ensemble des services et par conséquent de n'appeler que les variables effectivement définies (Fig. 7)

Nous avons donc maintenant deux approches pour définir les paramètres du serveur depuis le client :

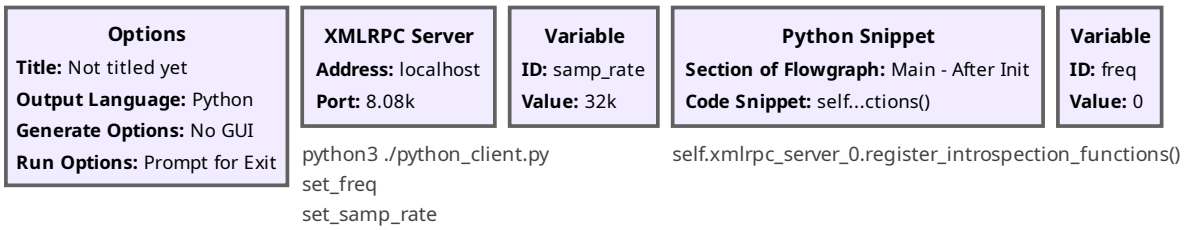


FIGURE 7 – Activation de la méthode `register_introspection_functions()` dans un Python Snippet afin de permettre aux clients XMLRPC d'obtenir la liste des variables connues du serveur.

un serveur TCP/IP ou un serveur XMLRPC. Ces connexions connectées sont appropriées pour garantir que le message émis par le client est bien compris et acquitté par le serveur. Cependant, il est courant que nous désirions que le serveur poursuive ses activités d'acquisition et de traitement de signaux même si aucun client n'est à l'écoute de ses services : cette liaison non-bloquante est prise en charge par UDP, qui une fois de plus n'encapsule pas les informations mais ne fait que regrouper les octets pour les communiquer à d'éventuels clients à l'écoute. Si personne n'écoute, les informations sont simplement perdues, et si le routage change en cours de communication, ni l'ordre ni l'intégrité du flux de données n'est garanti. Afin de se faciliter la tâche d'organiser les informations transmises, nous allons compléter la démonstration précédente des échanges UDP en bénéficiant d'une bibliothèque plus abstraite qu'est Zero-MQ 0MQ).

3 ZeroMQ

Nous avons rencontré ZeroMQ pour la première fois comme bloc de communication de GNU Radio Companion, le générateur de code Python pour le traitement numérique de signaux radiofréquences. Barry Dugan, qui se charge de la documentation des blocs GNU Radio Companion, en a proposé une description détaillée dans [8]. Nous nous limiterons ici à présenter les implémentations Python, GNU Octave et C, sachant que le logiciel propriétaire Matlab supporte aussi ce protocole de communication, comme nombre d'autres langages tel que l'indique

```
$ apt-cache search mq | grep z
python3-zmq - Python3 bindings for 0MQ library
libczmq4 - High-level C binding for ZeroMQ
libgnuradio-zeromq3.10.5 - gnuradio zeromq functions
octave-zeromq - ZeroMQ binding for Octave
```

pour n'en citer que quelques uns.

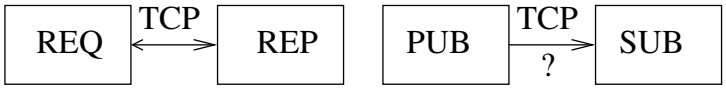


FIGURE 8 – ZeroMQ (ou 0MQ) propose une couche applicative au-dessus de TCP/IP et UDP/IP pour abstraire les informations transmises et les encapsuler dans des paquets munis de métadonnées. 0MQ fournit une fonctionnalité de mode connecté (apparentée en fonctionnalité à TCP) nommée REQ-REP garantissant les transactions bloquantes, et un mode non-connecté (apparenté en fonctionnalité à UDP) nommé PUB-SUB que nous favoriserons pour nos applications de transferts de données acquises par récepteur de radio logicielle pour un traitement déporté.

Divers ouvrages fournissent des codes de base [9, 10] mais soit avec des erreurs, soit avec des incompatibilités liées aux évolutions des versions : éplucher les nombreuses générations de documentations parfois incompatibles avec les bibliothèques actuelles peut s'avérer fastidieux (passage de 0MQ version 3 de 2013 à version 4 de 2021), mettant en évidence le danger de l'évolution de ces APIs complexes pour un projet pérenne.

Tout comme TCP et UDP, ZeroMQ propose un mode connecté garantissant les transactions – `request-reply` dans la nomenclature ZeroMQ – mais avec une lourdeur protocolaire de liaison bidirectionnelle avec acquittement, et un mode de diffusion d'informations sans garantie de réception qu'est

le *datagram* d'UDP – nommé chez ZeroMQ `publish-subscribe` (Fig. 8). Un exemple de serveur en C, nécessitant l'installation du paquet `libczmq-dev` sous Debian/GNU Linux, est de la forme

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <zmq.h>

int main()
{int k=0;
 char message[256];
 void *context = zmq_ctx_new();
 void *publisher = zmq_socket(context, ZMQ_PUB);
 if (zmq_bind(publisher, "tcp://127.0.0.1:5556")==0)
    {while (1)
       {sprintf(message,"Hello %03d",k);k++;
        zmq_send(publisher, message, strlen(message), 0);
        sleep(1);
       }
     zmq_close(publisher);
     zmq_ctx_destroy(context);
    } else printf("Socket error\n");
    return 0;
}
```

ou en Python pour une version qui communique un tableau de valeurs (vecteur) produit par NumPy

```
import numpy as np # pkg load signal;
import zmq         # pkg load zeromq;
import time
port = "5556"

context=zmq.Context()
sock=context.socket(zmq.PUB)
sock.bind("tcp://*:"+str(port)) # broadcast
k=0
while True:
    payload=np.arange(0+k,1024+k)
    print(k)
    k=k+1
    sock.send(payload)
    time.sleep(1)
```

tandis que le client en C se présente comme

```
// https://stackoverflow.com/questions/67025151/zeromq-pub-sub-example-in-c-libzmq
#include <stdio.h>
#include <zmq.h>

int main()
{   long long *res;
    void *context = zmq_ctx_new();
    void *subscriber = zmq_socket(context, ZMQ_SUB);
    char message[1024*8];
    int len;
    zmq_connect(subscriber, "tcp://127.0.0.1:5556");
    zmq_setsockopt(subscriber, ZMQ_SUBSCRIBE, "", 0);
    res=(long long*)(message);
    while (1) {
        len=zmq_recv(subscriber, message, 1024*8, 0);
        if (len!=-1) {printf("%d: %lld %lld %lld\n",len,(res[0]),res[1],(res[1023]));}
        else printf("error\n");
    }
    zmq_close(subscriber);
    zmq_ctx_destroy(context);
    return 0;
}
```

ou avec GNU Octave si le paquet `octave-zeromq` a été installé par

```
pkg load zeromq;
Nt=1024
sock1=zmq_socket(ZMQ_SUB);
zmq_connect(sock1,"tcp://127.0.0.1:5556");
zmq_setsockopt(sock1, ZMQ_SUBSCRIBE, "");
recv=zmq_recv(sock1, Nt*8, 0);
% vector=typecast(recv,"single complex");
vector=typecast(recv,"int64")
```

ou en Python avec

```
import numpy as np # pkg load signal;
import zmq         # pkg load zeromq;
import array
```

```
from matplotlib import pyplot as plt

Nt=256
context=zmq.Context()

sock1=context.socket(zmq.SUB) # sock1=zmq_socket(ZMQ_SUB);
sock1.connect("tcp://127.0.0.1:5556");
sock1.setsockopt(zmq.SUBSCRIBE, b"")
vector1=[]
while (len(vector1)<Nt):
  raw_recv=sock1.recv()
  recv=array.array('f',raw_recv) # f->l pour des entiers
  print(recv)      # vector1tmp=recv[0::2]
  plt.plot(recv)   # vector2tmp=recv[1::2] pour interleaved
  plt.show()
```

Nous pouvons nous convaincre que le serveur PUB continue son activité même en l'absence de client à l'écoute en observant le décompte qui s'incrémente chaque seconde :

```
$ ./ex1_server &
$ ./ex1_client
Hello 007
Hello 008
Hello 009
^C
$ ./ex1_client
Hello 013
Hello 014
^C
```

montre bien que même en interrompant le client SUB, le serveur PUB continue d'incrémenter sa variable qui sera passée de 9 à 13 lors de la reconnexion.

```
$ python3 ./server.py &
$ python3 ./client.py
Hello 8
Hello 9
Hello 10
^C

$ python3 ./client.py
Hello 16
Hello 17
Hello 18
^C
```

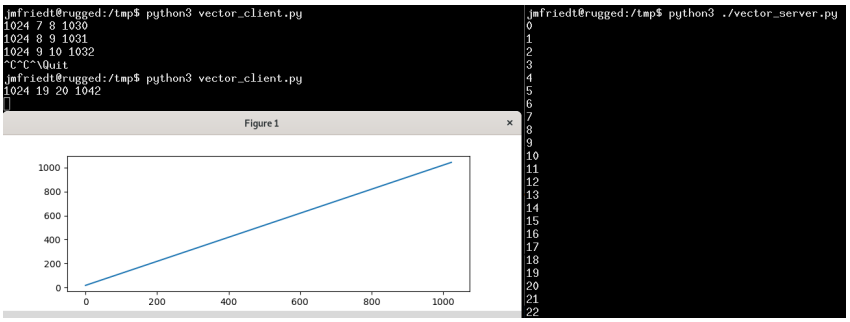


FIGURE 9 – Gauche : illustration d'un serveur PUB qui continue son décompte que le client SUB soit connecté ou non. Lors de la reconnexion avec une interruption par CTRL-C, le client reprend le décompte à la valeur atteinte par le serveur, les valeurs intermédiaires étant perdues. Cependant (droite) si une attente qui maintient la socket connectée – par exemple en attendant de fermer le premier affichage de `matplotlib` – sépare deux lectures, les valeurs transmises pendant l'attente sont conservées en tampon et affichées, avec un incrément de 7 à 8 puis 9 alors que le serveur avait atteint un décompte bien plus important au moment du dernier affichage, tel que nous le démontrons en tuant le client et le relançant avec un décompte qui reprend à 19.

Cependant, si le client ne déconnecte pas sa socket SUB, la séquence est bien contigüe et les paquets sont mémorisés tant qu'ils n'ont pas été traités. La profondeur de la mémoire tampon est dépendante de la version de 0MQ : le concept de High Watermark, décrit dans les options de l'API à `http://api.zeromq.org/2-1:zmq-setsockopt`, est implémenté pour éviter le dépassement de mémoire si un *subscriber* consomme trop lentement les données produites par le *publisher*, mais la version 2 de 0MQ proposait une taille nulle donc un tampon infini uniquement limité par la mémoire physique, tandis que la version 3 de 0MQ propose une profondeur de 1000 messages [11, p.76] sans garantie que nous ayons assez de mémoire pour respecter cet objectif. La seule façon que nous ayons trouvée pour garantir que le flux de données est le dernier en date et non pas des restes d'un message antérieur – par exemple dans le cas du traitement de signaux radiofréquences acquis par une antenne mobile pour être certain que les

données sont bien acquises sur la nouvelle position de l'antenne – est de fermer et rouvrir la socket SUB afin d'en éliminer la queue de données en attente (Fig. 9).

Au contraire dans un scénario de REP-REQ, un jeu de ping-pong impose au client REQuest de demander une nouvelle donnée au serveur REPly (Fig. 10) et de ne pas re-émettre de nouveau paquet sans y avoir été convié, au risque sinon de recevoir une erreur de type "*Operation cannot be accomplished in current state*".

Client REQ

```
import socket
import zmq
import array
context=zmq.Context()
sock1=context.socket(zmq.REQ)
sock1.connect("tcp://127.0.0.1:5556");
while True:
 noerror=1;
 while noerror:
  sock1.send(b"Hello")
  rcv=sock1.recv()
  # print(rcv.decode('ascii')) si str
  r=array.array('i',rcv)
  print(f"{len(r)} {r[0]} {r[1]} {r[-1]}"→
      ↪)
```

Serveur REP

```
import numpy as np # pkg load signal;
import zmq         # pkg load zeromq;
import time
port = "5556"
payload="Hello"
context=zmq.Context()
k=0
while True:
  sock=context.socket(zmq.REP)
  sock.bind("tcp://*:"+str(port)) # →
      ↪broadcast
  while True:
    message = sock.recv()
    print(message)
    payload=np.arange(0+k,1024+k,dtype=np→
        ↪.int32)
    sock.send(payload)
    print(k)
    k=k+1
    time.sleep(1)
```

```
jmfriedt@rugged:/tmp$ python3 ./client.py
1024 0 1 1023
1024 1 2 1024
1024 2 3 1025
1024 3 4 1026
1024 4 5 1027
1024 5 6 1028
```

```
jmfriedt@rugged:/tmp$ python3 ./server.py
b'Hello'
0
b'Hello'
1
b'Hello'
2
b'Hello'
3
b'Hello'
4
b'Hello'
5
b'Hello'
6
```

FIGURE 10 – Échanges entre client et serveur dans lequel chaque vecteur est requis par le client REQ pour être fourni par le serveur REP, garantissant le séquencement des transactions et l'absence de pertes de données.

Produire le flux de données depuis GNU Radio rend le prototypage plus ludique par une production continue de données, potentiellement depuis un récepteur de signaux radiofréquence physique mais ici depuis des signaux synthétiques qui imposent donc un bloc `Throttle` pour imposer à l'ordonnanceur GNU Radio de limiter sa production de données à `samp_rate` échantillons par seconde. Dans la Fig. 11, nous constatons que les donnés produites en Python par le code produit par GNU Radio Companion est convenablement lu en C (bas-gauche) ou Python (bas-droite) qui permet d'afficher la forme d'onde au moyen de `matplotlib` et constater son adéquation avec le motif en dents de scie produit, mais avec un nombre de données transmis variable tel que l'impose l'ordonnanceur GNU Radio. En effet dans ce contexte, nous ne pouvons faire d'hypothèse sur la longueur des vecteurs transmis et devrons soit attendre d'accumuler assez de points pour effectuer un traitement (par exemple convolution ou transformée de Fourier qui nécessitent un nombre connu de valeurs dans un vecteur pour permettre de calculer l'intégrale) ou de ne traiter que le sous-ensemble utile et mémoriser les autres valeurs.

De la même façon, la communication de GNU Radio (donc Python) avec GNU Octave est démontrée en Fig. 12 qui met en évidence la nécessité de convertir explicitement le paquet d'octets produit par le *publisher* pour l'interpréter de façon adéquate, soit comme des nombres à virgule flottante (`single`) voir complexe pour un flux de données IQ, soit ici comme entier (ici `int32` pour 4 octets/entier) – la liste des arguments supportés par `typecast` de GNU Octave est fournie par `help typecast`. Cette conversion du paquet d'octets vers le type adéquat est aussi valable pour Python avec l'argument de `array` tel que documenté à `https://docs.python.org/3/library/array.html`. Il est donc du ressort du développeur

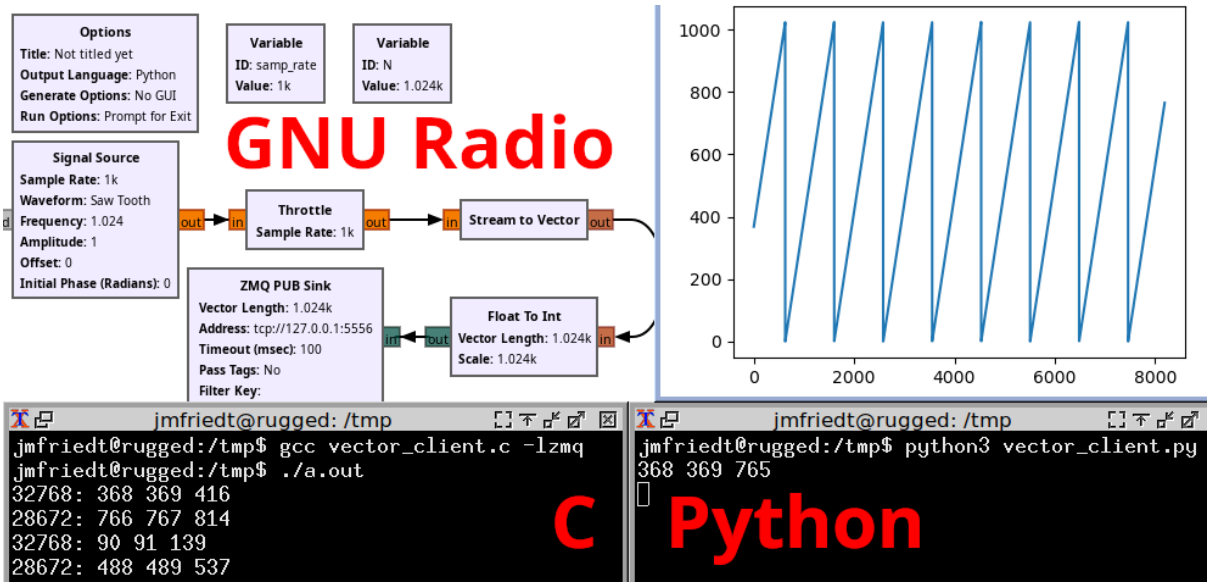


FIGURE 11 – Génération d'un signal en dent de scie par GNU Radio (*Signal Source* de forme *Saw Tooth*, d'amplitude 1 et de fréquence 1,024 produit au rythme de 1000 points/s et d'amplitude unitaire multipliée par 1024 lors du passage de nombre à virgule flottante – symbole orange – en entier codé sur 32 bits – symbole vert – émis en 0MQ PUB et reçu en Python en bas à droite par un 0MQ SUB pour un affichage par `matplotlib`, validant la cohérence de la transaction, et en C en bas à gauche, confirmant qu'une multitude de clients peuvent recevoir simultanément les vecteurs de données publiés.

de bien s'accorder sur les types de données échangées, soit d'implémenter un protocole garantissant la cohérence des transactions si la nature des données échangées risque de varier.

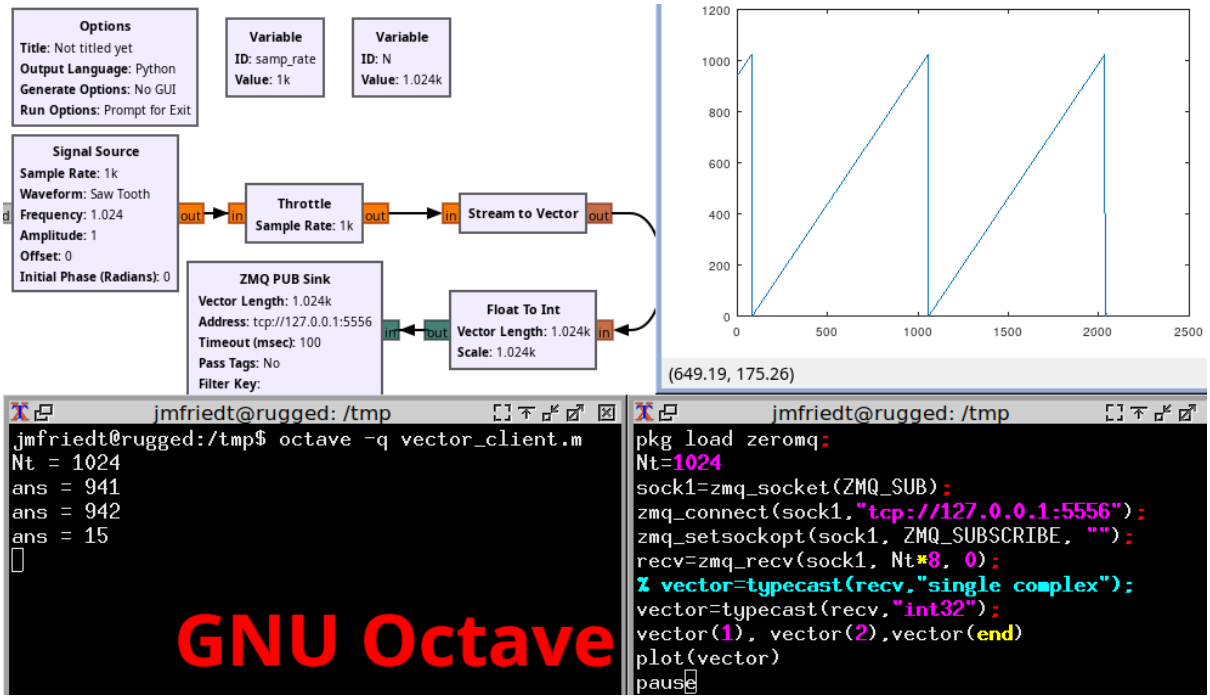


FIGURE 12 – Génération d'un signal en dent de scie par GNU Radio (*Signal Source* de forme *Saw Tooth*, d'amplitude 1 et de fréquence 1,024 produit au rythme de 1000 points/s et d'amplitude unitaire multipliée par 1024 lors du passage de nombre à virgule flottante – symbole orange – en entier codé sur 32 bits – symbole vert – émis en 0MQ PUB et reçu par GNU Radio par un 0MQ SUB pour un affichage graphique, validant la cohérence de la transaction.

0MQ permet donc de facilement échanger des informations de façon connectée ou en diffusant les informations sans acquittement, avec le concept de thèmes auxquels les clients peuvent s'abonner pour

ne conserver qu'un sous ensemble des informations transmises. On notera cependant que les informations sont diffusées en clair, gage d'efficacité mais avec les risques encourus tant il est devenu simple aujourd'hui de manipuler les trames IP (Fig. 13). Nous reconnaissons facilement dans la sortie de `tcpdump` les 32 octets de l'entête d'IPv4 qui commence par 0x40 [2], avec une transaction en TCP (le "06" de 0x4006, le premier quartet étant la durée de vie maximale du paquet TTL initialisé à 64), l'IP source et l'IP destination que sont 0x7f000001 ou 127.0.0.1 en décimal, le port de communication 0x15b4 qui vaut 5556 en décimal etc ... En effet dans le cadre de la diffusion de données issues d'un récepteur radiofréquence, une écoute des données transmises n'a probablement que peu d'importance, mais l'injection de données erronées pourrait être dramatique : ce n'est pas tant l'obfuscation des données que leur intégrité qui pourrait valoir une couche d'authentification qui n'est clairement pas présente quand nous affichons par `tcpdump` les données transmises à la socket sur laquelle s'est connectée le client SUB en C.

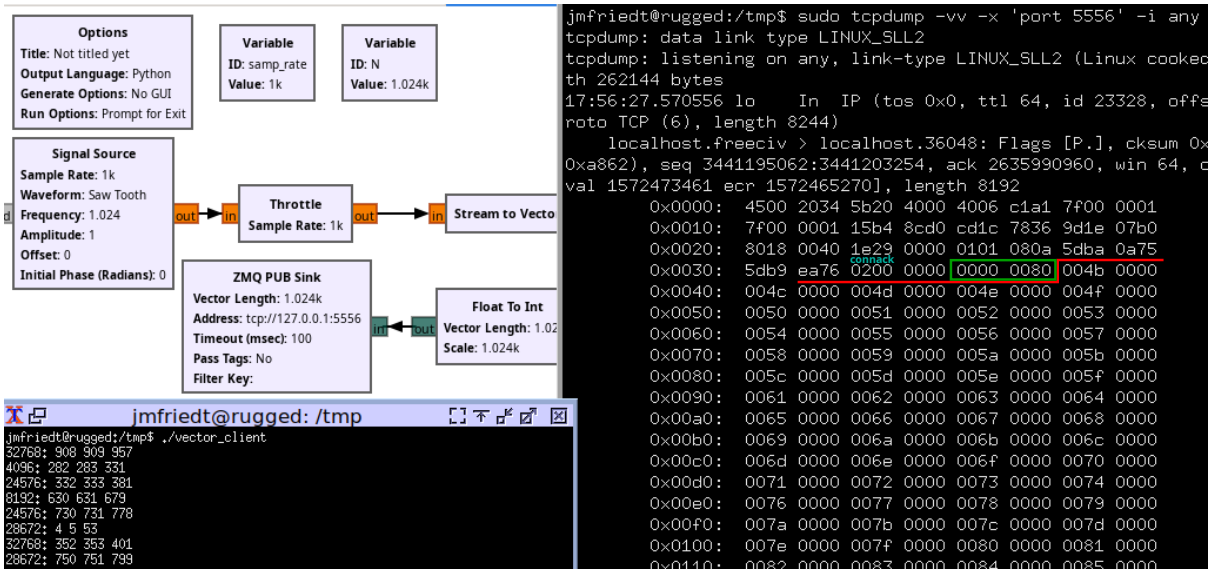


FIGURE 13 – Observation par `tcpdump -vv -x 'port 5556' -i lo` (plus sélectif que `any`) des données transmises par 0MQ : après les 32 octets de l'entête IP, nous observons que la surcharge du protocole 0MQ est minime avec quelques informations de la nature de la transaction (CONNACK pour une liaison du serveur vers un client), la taille de la charge utile (vert) 0x8000=32768 (pour rappel sur le réseau l'ordre est little endian) suivi (délimitation rouge) des données en clair, en accord avec les informations communiquées par le serveur en C (première transaction).

4 MQTT

MQTT (*Message Queuing Telemetry Transport*) s'annonce comme un protocole pour l'"Internet des Objets" (IoT) dans lequel des systèmes embarqués sont conçus pour communiquer. La nécessité d'une pile TCP/IP pour implémenter MQTT est dans ce contexte surprenante : IP, ICMP et UDP tenaient dans quelques kilo-octets de RAM et flash, bien moins que la mémoire nécessaire à mémoriser les paquets TCP qui doivent s'accumuler en cas de perte ou de re-routage d'un paquet qui en changerait l'ordre. Il semblerait que le principal bénéfice annoncé de MQTT, au-delà de sa centralisation sur un unique serveur (*broker* dans la nomenclature MQTT) est l'encryption SSL des paquets échangés (encore une fois au détriment de la puissance de calcul du client qui cherche à transmettre ses informations).

Tout comme 0MQ, MQTT se place dans un contexte de publish-subscribe, mais cette fois non pas dans une liaison point à point mais au travers d'un unique *broker* qui centralise les transactions et apparaît donc comme un point faible du réseau (Fig. 14) : ainsi, tout service dans MQTT est implémenté comme un client, qu'il soit *publisher* ou *subscriber*, avec un unique serveur qu'est le *broker*. Chaque *publisher* peut proposer des services au travers de filtres, et chaque *subscriber* peut filtrer les informations qu'il désire traiter. Ce mécanisme apparaît peu efficace dans l'échange de flux de données radiofréquences qui vise l'efficacité, mais `https://opensource.com/article/18/6/mqtt` fournit un exemple concret d'utilisation de ces fonctionnalités dans l'analyse de la production d'énergie dans l'état de New York où un très riche jeu de données est lentement mis à disposition par les producteurs d'électricité et accessible par MQTT avec les filtres appropriés selon une hiérarchie rappelant l'arborescence d'un système de fichiers tel que l'illustre MQTT Explorer à `http://mqtt-explorer.com/`.

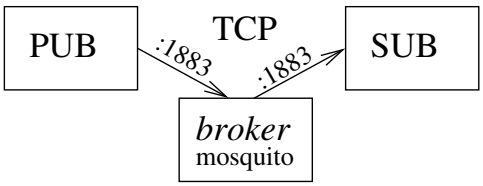


FIGURE 14 – MQTT s'appuie sur un concentrateur de données – le *broker* – qui centralise les échange et même si nous restons dans un modèle de Publish-Subscribe avec des thèmes publiés auquel un client peut ou non souscrire, cette fois tous les interlocuteurs s'appellent des clients qui se connectent au serveur qu'est le *broker* qui communique sur le port 1883.

4.1 MQTT pour Python, bash et C

Une implémentation de *broker* MQTT se nomme `mosquito` et c'est cette bibliothèque que nous exploiterons après l'avoir installée par `sudo apt install mosquitto mosquitto-clients` sous Debian/-GNU Linux. Du côté du client, une implémentation de MQTT, toujours promue par la fondation Eclipse [12], se nomme Paho et fournit la compatibilité vers une multitude de langages incluant C et Python. Sous Debian/GNU Linux, nous installons donc `sudo apt install libpaho-mqtt-dev` pour la C, identifié en cherchant quel paquet de la distribution fournit l'entête décrivant le contenu des bibliothèques `MQTTClient.h`, et `python3-paho-mqtt` pour Python.

Nous commençons par nous assurer qu'un *broker* est en cours d'exécution sur le système d'exploitation Linux permettant ces développements, soit par `ps aux | grep mosq` qui doit indiquer
`/usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf`
soit en consultant `/var/log/mosquitto/mosquitto.log` en tant qu'administrateur. La communication entre ce serveur et les clients MQTT se fera au travers du port 1883 qui doit donc être ouvert par tout mécanisme de protection des transactions par réseau (*firewall*). Nous validons le bon fonctionnement du *broker* depuis la ligne de commande par quelques échanges simples de type publish-subscribe :

```
$ mosquitto_pub -t "mycomputer" -m "Hello"
$ mosquitto_pub -t "mycomputer" -m "World"
$ mosquitto_sub -t "mycomputer"
Hello
World
```

Convaincus du bon fonctionnement du broker, nous nous efforçons d'implémenter l'échange de vecteurs de données, toujours en nous assurant de la cohérence des échanges puisque MQTT se contente de faire transiter des tableaux d'octets sans en coder l'organisation : en C, le service publish (que nous n'appellerons pas serveur compte tenu du broker) se code tout d'abord par un entête commun `entete.h` pour garantir la cohérence entre les deux clients

```c
#define ADDRESS     "tcp://127.0.0.1:1883"
#define CLIENTID    ""
#define TOPIC       "float_vect"
#define QOS         1
#define TIMEOUT     10000L
#define N 1024
```

qui est appelé par

```c
#include "stdio.h"
#include "stdlib.h"
#include "stdint.h"
#include "MQTTClient.h"
#include "entete.h"

int main(int argc, char* argv[])
{MQTTClient client;
 MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
 MQTTClient_message pubmsg = MQTTClient_message_initializer;
 MQTTClient_deliveryToken token;
 int rc;
 int32_t payload[N];
 int k;
 for (k=0;k<N;k++) payload[k]=k;

 MQTTClient_create(&client, ADDRESS, CLIENTID, MQTTCLIENT_PERSISTENCE_NONE, NULL);
 conn_opts.keepAliveInterval = 20;
 conn_opts.cleansession = 1;
```

```
 MQTTClient_connect(client, &conn_opts);
 pubmsg.payload = payload;
 pubmsg.payloadlen = sizeof(payload); // 4096
 pubmsg.qos = QOS;
 pubmsg.retained = 0;
 MQTTClient_publishMessage(client, TOPIC, &pubmsg, &token);
 rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
 printf("Message with delivery token %d delivered\n", token);
 MQTTClient_disconnect(client, 10000);
 MQTTClient_destroy(&client);
 return rc;
}
```

pour se compiler par `gcc source.c -lpaho-mqtt3c`. Le même résultat s'obtient en Python avec

```
import paho.mqtt.client as mqtt
import numpy
client=mqtt.Client()
client.connect("127.0.0.1")
data=numpy.arange(0,1024,dtype=numpy.int32)
client.publish("float_vect", data.tobytes() ,0)
```

et dans les deux cas nous vérifions qu'une transaction s'effectue par la publication de données reçues par la commande shell `mosquitto_sub -t "float_vect"` avec un service subscribe qui souscrit au flux `float_vect` tel que nous l'avons imposé en créant le *publisher*. Cependant, `mosquitto_sub` ne connaissant pas la nature des données transitant, il n'affiche que des octets sous forme de caractère ASCII qui n'ont aucun sens. Il nous faut donc proposer des clients – *subscribe* – capables de décoder les informations transmises, par exemple en s'inspirant de `https://eclipse.dev/paho/files/mqttdoc/MQTTClient/html/subasync.html` en C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include "MQTTClient.h"
#include "entete.h"

volatile MQTTClient_deliveryToken deliveredtoken;
void delivered(void *context, MQTTClient_deliveryToken dt)
{ printf("Message with token value %d delivery confirmed\n", dt);
  deliveredtoken = dt;
}

int callback_func(void *ctxt, char *tpcNam, int tpcLen, MQTTClient_message *msg)
{ int i;
  int32_t* ptr;
  ptr = (int32_t*)msg->payload;
  for(i=0; i<msg->payloadlen/sizeof(int32_t); i++)
     printf("%d ",ptr[i]);
  putchar('\n');
  MQTTClient_freeMessage(&msg);
  MQTTClient_free(tpcNam);
  return 1;
}

void connlost(void *context, char *cause)
{ printf("\nConnection lost: %s\n", cause);
}

int main(int argc, char* argv[])
{ MQTTClient client;
  MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
  int rc;
  int ch;
  MQTTClient_create(&client, ADDRESS, CLIENTID,
      MQTTCLIENT_PERSISTENCE_NONE, NULL);
  conn_opts.keepAliveInterval = 20;
  conn_opts.cleansession = 1;
  MQTTClient_setCallbacks(client, NULL, connlost, callback_func, delivered);
  MQTTClient_subscribe(client, TOPIC, QOS);
  do {ch = getchar(); }
  while(ch!='Q' && ch != 'q');
  MQTTClient_disconnect(client, 10000);
  MQTTClient_destroy(&client);
  return rc;
}
```

ou en Python

```
import paho.mqtt.client as mqtt
import numpy
import time
def callback_func(client, userdata, message):
    print("rcv ", numpy.frombuffer(message.payload, dtype=numpy.int32))
client=mqtt.Client()
client.connect("127.0.0.1")
client.subscribe("float_vect")
client.on_message=callback_func
client.loop_start()
time.sleep(40)
```

qui tous deux mettent en évidence un mécanisme intéressant de fonction appelée automatiquement (*callback*) lors de la réception d'un paquet de données sans qu'il soit explicitement nécessaire de faire appel à un thread séparé, la convention du C imposant que toute lecture soit bloquante sinon.

4.2 MQTT pour GNU Octave

MQTT n'est pas fourni comme paquet de Debian/GNU Linux pour GNU Octave, mais son installation depuis les sources disponibles à `https://sourceforge.net/p/octave-mqtt` se fait sans problème. En effet depuis le répertoire de téléchargement des sources, `make dist` va fabriquer une archive `.tar.gz` dans le répertoire `target`, et dans GNU Octave nous lançons la commande
`pkg install target/octave-mqtt-0.0.3.tar.gz`
pour installer `octave-mqtt` dans `$HOME/.local/share/octave/api-v57/packages/mqtt-0.0.3/`. Ce nouveau paquet donne désormais accès aux fonctions nécessaires à se connecter au serveur (*broker*) et donc toutes les transactions sont vues du point de vue d'un client : pour une connexion subscribe qui reçoit les messages

```
pkg load mqtt
client = mqttclient("tcp://127.0.0.1");
subs = subscribe(client, "float_vect");
vector=[]
do
  recv = read(client, "float_vect");
  if (isempty(recv)==0)
   % vector=typecast(recv,"single complex");
     vector=typecast(recv.Data,"int32")
  end
until (isempty(vector)==0)
```

et pour une connexion publish qui les expédie :

```
pkg load mqtt
client = mqttclient("tcp://127.0.0.1")
vector=[1:1024]
data=typecast(vector,"char")
write(client, "float_vect", data, "QualityOfService", 1);
```

Cependant, GNU Octave ne permet pas un typage fort des variables échangées et nous constatons que le tableau émis est sous forme de nombre flottants exprimés en double précision (donc 8 octets par donnée) et que du point de vue de Python ce tableau se lit en modifiant la fonction de callback au moyen de

```
def callback_func(client, userdata, message):
    print(numpy.frombuffer(message.payload, dtype=numpy.float64))
```

donc une conversion de la série d'octets en flottant double précision par `dtype=numpy.float64`.

Finalement, nous concluons ce survol de MQTT en mettant en évidence la simplicité d'intégrer une bibliothèque en Python dans GNU Radio : en effet, `https://github.com/crasu/gr-mqtt` propose une interface entre GNU Radio et MQTT en encapsulant simplement les fonctions que nous venons d'expliciter dans la méthode `work` d'un bloc dédié Python compatible avec les appels GNU Radio.

5 Système de fichiers Unix

En abordant des mécanismes faisant appel aux sockets – le point de divergence entre l'implémentation d'Unix qui expose les interfaces réseau selon une API spécifique [13], et sa philosophie originale de "tout est fichier" – nous avons omis l'approche probablement la plus simple pour faire communiquer deux processus, le *pipe* ou tuyau. En effet, si nous créons un pseudo-fichier qui fait communiquer son entrée avec sa sortie par `mkfifo /tmp/myfifo`, alors toute donnée qui entre dans le tuyau sera accessible par tout processus qui s'est connecté à sa sortie comme s'il s'agissait d'un vulgaire fichier, mais sans stockage des données sur un support physique. Un cas particulier serait le lien entre `stdout` d'un processus et `stdin` d'un autre processus par le symbole `|`, mais ici nous nous intéresserons au cas de fichiers tels qu'accessibles par `open()`, `read()`, `write()` et `close()`.

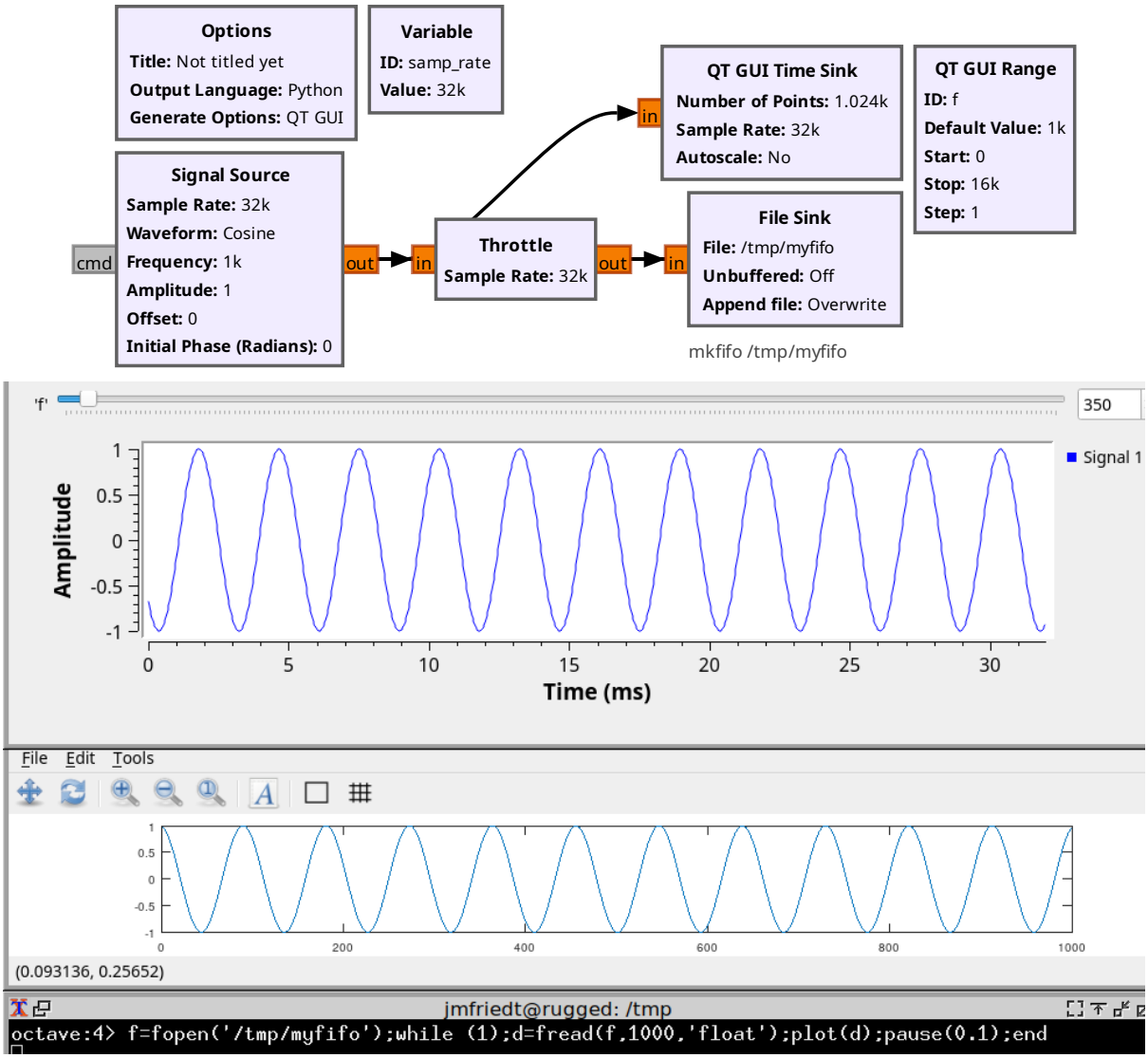


FIGURE 15 – Une chaîne de traitement simple GNU Radio Companion (haut) produit un flux de données au rythme de 32000 points par seconde environ (bloc `Throttle`) et alimente ainsi un fichier connecté à une FIFO. À l'autre bout, GNU Octave a ouvert ce fichier, en lit le contenu et l'affiche sur un graphique raffraîchi aussi vite que possible (bas). L'oscilloscope (milieu) permet de valider quand les données sont produites par GNU Radio lors de l'exécution de la chaîne de traitements.

On pourra donc s'assurer du bon fonctionnement en lançant le programme GNU Radio proposé en Fig. 15, pour constater que rien ne se passe (pas d'affichage sur la sortie oscilloscope `Time Sink`) tant que l'autre bout du tuyau n'est pas connecté. Cependant, en exécutant dans un terminal `cat < /tmp/myfifo` nous constaterons que l'ordonnanceur lance la génération des données dont la représentation binaire s'affiche dans le terminal. Plus expressif, sous GNU Octave nous effectuons la même opération par

```
f=fopen('/tmp/myfifo');while (1);d=fread(f,1000,'float');plot(d);refresh();end
```

qui ouvre le fichier une seule fois puis lit continuellement les 1000 derniers nombres à virgule flottante (implicitement codés sur 4 octets donc 4000 octets) pour en afficher le contenu. En manipulant l'ascenceur qui fait varier la fréquence du signal nous constaterons une latence croissante puisque toutes les données ne peuvent être consommées en temps réel. Nous avons tenté de fermer et ouvrir le fichier dans la boucle mais rien n'y fait, le tuyau fait bien son travail en mémorisant toutes les données injectées tant qu'elles ne sont pas consommées.

On prendra garde à bien créer `/tmp/myfifo` *avant* de lancer GNU Radio, faute de quoi un vrai fichier (sans attribut `p` dans le premier champ de `ls -la /tmp/myfifo`) sera créé en son absence, et sera rempli petit à petit sans fournir le résultat escompté.

Cette approche est élégamment présentée pour cascader les données dans une chaîne de traitement de radio logicielle au cours de la session 2023 de la conférence Software Defined Radio Academy disponible à [14].

6 C dans Python : `ctypes` et pybind11

Nous avons exploré jusqu'ici le partage de données au travers de sockets, rendant transparente la communication au sein d'un même ordinateur ou au travers d'ordinateurs connectés sur un réseau, pour partager l'acquisition et le traitement de données entre diverses tâches potentiellement écrites dans des langages différents. Cependant, une alternative pour bénéficier des atouts de divers langages sans passer par une socket tient en la production d'exécutables capables de faire appel à une bibliothèque dynamique ou une version binaire du code compilé depuis le langage interprété. Notre collègue Benoît Dubois (FEMTO-Engineering) nous a ainsi fait découvrir `ctypes` pour appeler des fonctions écrites en C depuis Python, amenant la vitesse du langage compilé à la souplesse du langage interprété. Ce faisant, nous nous sommes interrogés sur la cohérence des zones mémoires adressées par chaque langage et si la structure de donnée est bien partagée ou dupliquée. Pour ce faire, diverses fonctions C manipulant divers types de pointeurs sont définies dans un fichier `t.c`

```c
#include <stdio.h>
void fonctionc(const char *y) {printf("C: %p\n",y);}
long fonctiond(double *y) {printf("C: %p\n",y); return((long)y); }
long fonctionv(void* y) {printf("C: %p\n",y); return((long)y); }
void fonctionp(void)  {printf("Hello\n");}
```

et compilées en objet par `gcc -c t.c -fPIC` dont le comportement est indépendant de l'emplacement en mémoire du binaire (PIC – *Position Independent Code* – n'utiise que des sauts relatifs et pas de sauts à une adresse absolue). La bibliothèque dynamique – *shared object* d'extension `.so` est alors produite par `gcc -shared t.o -o t.so` dont nous vérifions le contenu par `nm -D t.so` pour valider qu'elle contient les fonctions que nous avons définies. Nous constatons que nous désirons passer un pointeur de caractères (aussi nommé chaîne de caractères, donc des arguments entre "..." dans tous les langages), un pointeur sur un tableau de flottants (par exemple un vecteur ou une matrice en C), un pointeur de type non-défini tel que souvent utilisé lorsque la nature de l'argument n'est pas précisé à la compilation (par exemple une structure de données) et finalement une procédure sans argument.

Cette bibliothèque en C est appelée depuis Python au moyen de `ctypes` par

```python
import ctypes as ct
import numpy as np
clib = ct.CDLL("./t.so")  # charge la bibliotheque
clib.fonctiond.restype=ct.c_int  # type de retour
clib.fonctiond.argtypes=[np.ctypeslib.ndpointer(dtype=np.float64,ndim=1,flags="→
    ↪C_CONTIGUOUS")]
                         # ^^^ type de l'argument
a=np.arange(10,dtype='float64')
print(f"Python: {a.ctypes.data:x}") # emplacement de a
clib.fonctiond(a)                   # affiche le pointeur sur a
clib.fonctionp()
```

pour afficher lors de son exécution

```
$ python3 ./t.py
Python: 1650310
C: 0x1650310
Hello
```

démontrant que le pointeur est bien le même sur la structure créée par NumPy et celle reçue par la fonction `fonctiond()` en C.

Récemment (depuis sa version 3.9), GNU Radio a décidé d'exposer ses blocs de traitement en C++ à Python au moyen de pybind11, une technique qui bénéficie des fonctionnalités de C++ de faire le lien entre ce langage et Python au moment de la compilation. L'exemple ci-dessus devient presque compatible avec Python en déclarant les fonctions dans un fichier que nous nommerons `tpybind.cpp` contenant

```c
#include <pybind11/pybind11.h>

#include "t.c"

PYBIND11_MODULE(tpybind, m) { // must be the same name than the lib
    m.doc() = "pybind11 example plugin"; // optional module docstring
    m.def("fonctiond", &fonctiond, "double pointer");
    m.def("fonctionv", &fonctionv, "void pointer");
    m.def("fonctionc", &fonctionc, "byte/char pointer");
    m.def("fonctionp", &fonctionp, "no argument");
```

```
}
```
et malgré notre dégoût à `#include` un code source en C, ce programme se compile par

```
g++ -O3 -Wall -shared -std=c++11 -fPIC $(python3 -m pybind11 --includes) tpybind.cpp \
-o tpybind$(python3-config --extension-suffix)
```

pour produire un fichier d'extension `.cpython-311-x86_64-linux-gnu.so` que nous copions dans `/usr/lib/python3.11/lib-dynload`
pour le rendre accessible. On notera que la cohérence des noms entre la bibliothèque qui sera chargée en Python par `import tpybind` et le premier argument de `PYBIND11_MODULE` est important : il doit s'agir du même nom [15]. Ce faisant, nous pourrons exécuter en Python

```
import tpybind
tpybind.fonctionp()         # Hello
tpybind.fonctionc("Hello") # C: 0x7fff42394320
tpybind.fonctiond([1.,2.]) # incompatible function arguments.
```

et en effet le passage d'un tableau (pointeur) de nombre flottants ne semble pas supporté lorsque nous consultons la liste des argument de pybind11 à [16]. Une méthode `memoryview` semble conçue pour exposer le contenu de la mémoire d'une structure en Python vers les fonctions C/C++ mais nous n'avons pu lui expliquer comment alimenter le pointeur de flottants de `fonctiond()`.

7 C dans GNU Octave : SWIG et `mkoctfile`

SWIG (*Simplified Wrapper and Interface Generator*), ancienne méthode utilisée par GNU Radio pour exporter ses bibliotèques C++ vers Python avant le passage à pybind, a été introduit par William Daniau dans sa présentation d'interfaçage de fonctions C++ avec divers langages interprétés [17] mais Octave ne fait pas partie des outils considérés. Nous avons pu facilement interfacer les fonctions acceptant des chaînes de caractères (pointeur de `char*`) ou sans argument écrites en C avec une fonction Octave par

```
%module t_wrap /* MUST be the name of the .oct file */
%feature("autodoc", 1);
%inline %{
extern int fonctiond (double*);
extern void fonctionc (const char*);
extern void fonctionp (void);
extern void fonctionv (void*);
%}
```

qui s'analyse syntaxiquement au moyen de `swig -octave t.i` pour produire le code C++ `t_wrap.cxx` qui est compilé en une bibliothèque reconnue par GNU Octave par `g++ -c -I/usr/include/octave-7.3.0 -fpic -std=c++20 t_wrap.cxx` lié en un exécutable par `gcc -shared t.o t_wrap.o -L/usr/lib/x86_64-linux-gnu/o -rdynamic -loctinterp -loctave -lpthread -lm -o t_wrap.oct`. On notera que dans cette utilisation de SWIG, l'objet `t.o` est le même qu'auparavant quand nous nous étions lié à Python par `ctypes`, et la bibliothèque dynamique résultante permet bien d'appeler depuis GNU Octave

```
> t_wrap
> t_wrap.fonctionc("hello")
> C: 0x7fdea44d8470
> t_wrap.fonctiond([1])
> error: in method 'fonctiond', argument 1 of type 'double *' (SWIG_TypeError)
```

pour afficher l'adresse de l'argument de la fonction `fonctionc()`. Cependant, ici encore et comme avec `pybind`, nous avons été incapables de passer un pointeur vers un tableau de nombres (entiers, flottants) puisque la matrice de GNU Octave est une classe complexe de C++ représentant les propriétés du tableau en plus de son contenu. SWIG semble avoir été abandonné par GNU Radio par la difficulté à déverminer les erreurs à l'exécution et surtout pour éliminer une dépendance avec encore une bibliothèque externe plus ou moins mal maintenue, au bénéfice de `pybind` qui tire profit des dernières évolutions de C++.

Cependant, GNU Octave propose de nativement interfacer des fonctions C/C++ grâce à `mkoctfile`. Ainsi, un programme trivial inspiré de `https://docs.octave.org/latest/External-Code-Interface.html` de la forme

```
#include <octave/oct.h>

DEFUN_DLD (pointeraddr, args, , "Pointer address")
{if (args.length () != 1) print_usage ();
 printf("%p\n",&args(0));
 return octave_value ((unsigned long)&args(0));
}
```

se compile par `mkoctfile pointeraddr.cc` (l'extension est importante car `mkoctfile` sélectionne `gcc` ou `g++` pour compiler selon l'extension `.c` ou `.cc`) pour produire un fichier d'extension `.oct` dont le nom doit être le même que celui de la fonction. Lors de l'exécution sous GNU Octave

```
> dec2hex(pointeraddr(a))
0x7f05004d92c0
ans = 7F05004D92C0
```

nous constatons que l'adresse du pointeur est correctement affichée et renvoyée à l'interpréteur. La documentation met en garde contre quelques subtilités possibles lors de l'inclusion de C dans le C++ `https://docs.octave.org/latest/Calling-External-Code-from-Oct_002dFiles.html` classiquement rencontrées lorsque ces deux langages cohabitent.

8 Conclusion

Nous nous sommes efforcés de démontrer comment faire communiquer divers langages afin de distribuer les traitements soit en tirant le meilleur parti de chaque langage, soit en partageant les ressources au travers d'ordinateurs distincts. Pour ce faire, nous avons exploré XMLRPC, 0MQ et MQTT pour une communication par sockets, ou `ctypes`, pybind et SWIG pour l'intégration de fonctions en C dans GNU Octave. Nombre d'autres mécanismes ont été proposés au point de s'y perdre, avec des capacités de déverminage plus ou moins avancées : ainsi, depuis sa version 3.9, GNU Radio a décidé d'abandonner SWIG au profit de l'intégration native de C++ avec Python au moyen de pybind (`https://pybind11.readthedocs.io/en/stable/basics.html`), transition qui ne s'est pas faite sans douleur en rompant la compatibilité avec tous les blocs de traitement existant. Ainsi, le choix de la bonne infrastructure garantira la pérennité des développements et la continuité d'un projet ... jusqu'à la prochaine évolution incompatible de l'API !

Un point que nous n'avons pas abordé au cours de cet exposé est l'utilisation des `websockets` comme alternative aux `sockets` natives de POSIX mais sur la couche applicative la plus élevée de la description OSI. André Buhart (F1ATB) rappelle cette approche dans *RemoteSDR* à `https://f1atb.fr/index.php/2020/07/19/gnu-radio-to-web-client/`. Le lecteur est encouragé à approfondir cette voie si la portabilité entre systèmes d'exploitation est requise.

L'ensemble des programmes proposés dans cet article est disponibles à `http://github.com/jmfriedt/gnuradio_communication`

Références

[1] K. Hafner & M. Lyon, *Where wizards stay up late : The origins of the Internet*, Simon and Schuster (1998)

[2] W.R. Stevens, *TCP/IP Illustrated (Vol I & II)*, Addison-Wesley (1994)

[3] J.-M Friedt, *Décodage d'images numériques issues de satellites météorologiques en orbite basse : le protocole LRPT de Meteor-M2* (3 parties), GNU/Linux Magazine France 226–228 (Mai-Aout 2019)

[4] Norme POSIX, section *Sockets* à `https://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_10`

[5] Airbus Defence & Space, *Sentinel-1 SAR Space Packet Protocol Data Unit* (2015) en page 10/85.

[6] *MQTT V3.1 Protocol Specification* (2010) à `https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html`

[7] *XML-RPC Specification*, `http://xmlrpc.com/spec.md` (2003)

[8] B. Dugan, *KV4FV : Understanding ZMQ-Blocks*, Software Defiend Radio Academy (2021) à `https://www.youtube.com/watch?v=LPjZaOmNfxc`

[9] P. Hintjens, *ZeroMQ : messaging for many applications*, O'Reilly Media (2013)

[10] F. Akgul, *ZeroMQ*, Packt Publishing (2013)

[11] P. Hintjens, *Code Connected Volume 1 – Learning ZeroMQ*, à `https://archive.org/details/cc1pe` et les sections `ZMQ_RCVHWM` et `ZMQ_SNDHWM` de la page de manuel `man 3 zmq_setsockopt`

[12] Eclipse Paho Downloads, à `https://eclipse.dev/paho/index.php?page=downloads.php`

[13] J. Train, J.D. Touch, L. Eggert & Y. Wang, *NetFS : networking through the file system*, ISI Technical Report ISI-TR-2003-579 (2003) à `https://www.strayalpha.com/pubs/isi-tr-579.pdf` et bien entendu la description de Plan9 dans R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey & P. Winterbottom, *Plan 9 from bell labs*, Proc. Summer 1990 UKUUG Conference, qui explique "*Graphics and networking were added to UNIX well into its lifetime and remain poorly integrated and difficult to administer. More important, the early focus on having private machines made it difficult for networks of machines to serve as seamlessly as the old monolithic timesharing systems.*"

[14] J. Ketterl DD5JFK, *OpenWebRX*, Software Defined Radio Academy (2023) à 4h55 de `https://www.youtube.com/watch?v=yFFFAMRQHP4` et en particulier la communication par *pipes* à 5h10

[15] On notera que cet ajout du nom de l'architecture du processeur qui compile dans le nom de la biblithèque est source de bien des chagrins lors de la cross-compilation, par exemple dans Buildroot, tel que discuté à `https://github.com/gnuradio/gnuradio/issues/5455` et liens associés.

[16] pybind11, *List of all builtin conversions* à `https://pybind11.readthedocs.io/en/stable/advanced/cast/overview.html`

[17] W. Daniau, , GNU/Linux Magazine France **226** (Mai 2019) à `https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-226/interfacage-de-code-c-pour-ruby-et-python-avec-swig`