Sharing data for distributed processing: communicating through a network or between languages

Jean-Michel Friedt, FEMTO-ST time & frequency, Besançon, France August 8, 2023

How to make the most of the various languages at our disposal, between the prototyping speed of Python or GNU Octave and the execution speed of C? We will exchange data between functions written in these languages, either through network sockets or dynamic libraries.

Our objective in this presentation is to see how to make the most of the three languages for acquiring and processing digital signals that we use on a daily basis: C for its speed and compactness as a compiled language, GNU Octave for an interpreted language implementing linear algebra functions inherited from Matlab, and Python. While each language is used independently, we approach the exchange of information between these languages in a concept of centralized or decentralized processing through computer network communication.

Communication between digital systems, embedded or not, seems to have become the norm, forgetting the underlying principles. While historically the purpose of the Internet was to standardize communications between the numerous subnetworks that flourished in the 1960s and 1970s [1] according to a rational and hierarchical architecture with an obsession for decentralization to avoid dependence on a single node centralizing exchanges (think "Cold War" and "nuclear attack" for a network funded by DARPA), today a plethora of additional layers are added on top of Internet Protocol (IP) to bring new "functionalities." We were happy with IP, TCP, and UDP [2], and even with raw sockets on Ethernet when we didn't need packet routing in a point-to-point connection to control an embedded system (`https://sourceforge.net/projects/proexgprcontrol/`), so these application layers seem as unnecessary as they are resource-intensive, or even unstable over time with the constant API changes that we have once again verified while writing this prose.

Nevertheless, here we will explore three application layers propagating signals over IP: XMLRPC, ZeroMQ, and MQTT, which are responsible for organizing data during transactions. Beyond the exchanges through *sockets* that describe communication interfaces compatible with an Internet connection, there are many gateways between programming languages for exchanging data and making the most of each language – the execution speed of the compiled language (C), and the ease of prototyping of the interpreted language (Python and therefore GNU Radio, GNU Octave). Even though this presentation aims to be agnostic of any specific application, the processing of radio frequency signals as proposed by GNU Radio provides the guiding principle of our research and we will rely on the GNU Radio Companion Python code generator to select the exposed technologies. The nature of the data we want to exchange is a continuous stream of radio frequency data acquired by hardware receiver and whose information needs to be processed, locally or remotely, by a computing system implemented in the most appropriate language.

The presentation is not intended to be an exhaustive treatment of all the bridges between languages – as we only have a fairly good knowledge of C, Python, and GNU Octave that we use on a daily basis for the digital processing of discretely sampled signals – but rather a guide to show how each language can contribute to a complex global system with simple and efficient processing parts for speed (C) and prototyping in interpreted language with more lenient typing at the expense of efficiency (Fig. 1). Nevertheless, these bridges come at the expense of a new dependence on communication infrastructures with the inherent risks of APIs breaking and therefore loss of functionality beyond our control.

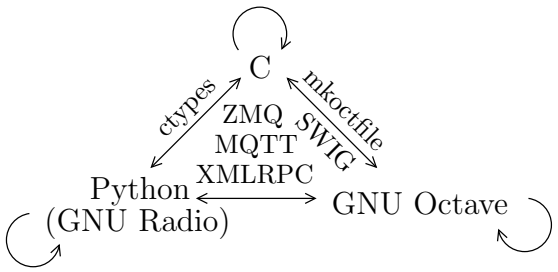


Figure 1: Interactions between C, GNU Octave and native Python or produced by GNU Radio Companion, and associated tools tackled in this article.

1 TCP/IP and UDP/IP

Let's start with the basics: the interoperability of computers connected on a network according to the principles of the Internet.

The hierarchy of abstraction layers is formalized by the OSI (*Open Systems Interconnection*) standard, which may seem arbitrary until one tries to implement it in a practical case [3] to discover that each layer implies different technical expertise and knowledge. Thus, the lowest layer – hardware – will be easily approached by an electronics specialist, while the highest layer – the application layer – involves many abstract computer concepts. In between, information needs to be assembled into packets, routed from one machine to another so that discrete aloow for routing the information from the source to the destination, and the parties involved must agree on the representation of the information and the various services capable of processing the information (socket ports). The principle of the OSI hierarchy is that each higher layer assumes that the lower layers have been implemented and are functional. Therefore, no packet routing by TCP in connected mode, which guarantees the integrity of the exchanges, or UDP in which a server broadcasts information to clients that may or may not receive it, is possible without access control and conversion of physical address into a software address by ARP (Fig. 2).

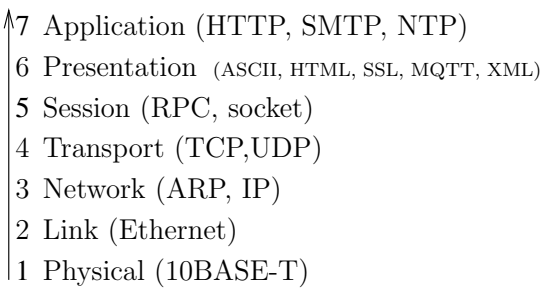


Figure 2: OSI layer hierarchy describing the services needed for a communication through a computer network. We shall be interested here to the upper layers.

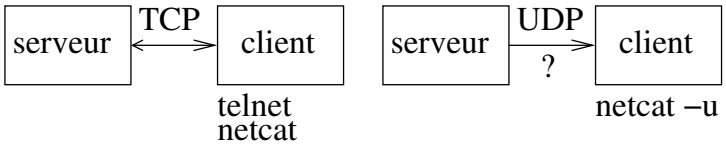


Figure 3: Concept of a server – waiting to provide a service – and a client – requesting a service, exchanging data either in connected mode to guarantee exchanges (TCP) or in datagram mode without validation of transactions (UDP).

Above the IP layer, which translates physical addresses into logical addresses, there are two modes of communication: TCP, which guarantees transactions (connected mode) and in which the server blocks its exchanges until they are acknowledged by the client, and UDP (datagram), in which the server sends data that may or may not be received by a client in an order that is not guaranteed depending on the packet routing along the route between the server and the client. In this second case, the server performs its operations independently of any connection from a client to receive or not receive the acquired data: this mode of communication is best suited when implementing a RADAR, for example, which can freely control a software defined radio receiver and move the antennas while a client receives data when conditions are favorable. Universal clients that make it easy to test servers are `telnet` and `netcat` for TCP, the latter with its `-u` option for receiving information in UDP. One might wonder why not always use TCP, which guarantees data exchanges? A TCP exchange requires storing transiting packets that could be corrupted or whose order has been swapped due to changes in the routing rules on the network during communication: a TCP/IP stack is very heavy to implement and resource-intensive, while a UDP exchange can be implemented in a few lines in the absence of any acknowledgments.

The concept of a socket is at the heart of a Unix system, which cannot function without it, as defined by the POSIX standard [4]. Just execute the `ss` (*socket statistics*) command to see the hundreds of communication pipes open on a GNU Linux system, even when disconnected from the Internet but exchanging information between its various services. Thus, a socket does not necessarily transport data from one computer to another over a computer network, but can exchange information between processes running on the same computer: this is called *Inter-Process Communications* or IPC. Remote control of processes is a special case of IPC called RPC for *Remote Procedure Call*. The question then arises about the representation of the exchanged data and their encapsulation so that the parties involved can agree on their representation.

Indeed, a basic TCP server – remember that the **server** is constantly waiting to provide a service to **clients** (Fig. 3) who connect occasionally to access this service – is written in C

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

#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)
}
```

or in 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
```

to connect a server to a socket (`bind`), wait for a client connection (`listen`), and exchange information (`recv`, `send`). These series of bytes have no structure and have meaning only because the two parties agreed in advance on their organization. These examples are still useful because, for example, in GNU Radio, the Python server proposed above is launched in an independent thread by

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

in a `Python Snippet` executed during the initialization of the scheduler. Passing the argument `self` gives access to all the functions defined by GNU Radio, and in particular, the *setter* and *getter* associated with each variable declared in the processing chain. Therefore, the thread can call `self.get_var()` and `self.set_var()` if the variable `var` has been defined in order to modify its content. We extensively use this mechanism when a client needs to sweep a parameter of a radiofrequency link, such as the carrier frequency of the signal.

Thus, in GNU Octave, a client of the form

```
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
```

will connect to port 4242, the same port to which the server has previously bound on the same computer running the server (127.0.0.1), to send a command, for example the letter "s" which could indicate the start of a processing sequence. Here, a connected TCP connection indicated by `SOCK_STREAM` indicates that the transactions are guaranteed by acknowledgment of each exchange, unlike an unconnected transaction or *datagram* according to UDP in which information is transmitted without guaranteeing its reception. These two modes will be used depending on whether the information needs to be organized and acknowledged (TCP) or simply sent to clients that may or may not be listening and where the loss has little consequence (for example, a data stream coming from a radio frequency receiver).

Python and GNU Octave are two interpreted languages that we often use together, Python for its

flexibility in accessing hardware resources and its use in connecting together the processing blocks of GNU Radio produced by signal analysis chains designed in GNU Radio Companion, and Octave for the ease of its matrix implementation of linear algebra algorithms according to the language derived from Matlab. The programmer, more flexible than the author in Python, will have no difficulty in translating the algorithms proposed in GNU Octave to NumPy without having to go through `https://numpy.org/doc/stable/user/numpy-for-matlab-users.html`. Thus, in this presentation, we will strive to exchange not only scalars but especially data vectors between Python and Octave.

From left to right or from right to left: byte order

Apart from certain Airbus Defence documentation for the European Space Agency [5], it seems quite obvious in the West to place the most significant bits on the left and the least significant bits on the right, thus writing one thousand two hundred thirty-four as 0x4d2 or 0b10011010010 as indicated by `dec2hex` and `dec2bin` in GNU Octave. The situation is less clear for the arrangement of bytes for a magnitude coded on 8 bits: historically, the American DARPA and Western countries that dominated the development of the Internet, it seems natural to place the most significant bytes on the left and the least significant bytes on the right, and thus write 0x4d2 levé ("to the right") so that the display of the memory contents from its lowest address to its highest address displays 0x04 0xd2. As historically the Internet was developed [1] by BBN on Honeywell and IBM architectures and then by Sun Microsystems on SPARC and Motorola architectures, it was natural to adopt this order selected by these processor architectures, called *big endian* (most significant byte at the lowest address) to transmit information coded on multiple bytes over the network. However, Intel had the idea of placing the least significant byte at the lowest address, a choice that becomes logical when arithmetic operations are performed on a CISC architecture with variable length instructions: the arithmetic and logic unit reads the instruction (the opcode), begins loading the arguments, and if the first argument read is the units, then the arithmetic operation can start while reading the next byte containing the tens and possibly the hundreds and thousands, thus propagating the carry during each sub-operation. This organization of the least significant byte at the lowest address is called *little endian*. On the contrary, a big endian organization requires reading the integer by starting with the thousands before ending with the units to start the arithmetic operation.

Even though today Intel/AMD little endian processors dominate the market for consumer personal computers, the Internet, like Java and therefore Android, remains big endian. Exchanging information between the two architectures requires agreeing on the byte order: this is the purpose of the `htons` (or `htonl` for 4 bytes) instructions for organizing two bytes (a short in C) in the correct order, from host to network and vice versa at the other end. These macros are defined in `/usr/include/netinet/in.h` on GNU/Linux as an identity or byte swap depending on the architecture used (`#if __BYTE_ORDER==__BIG_ENDIAN`). While this operation must be performed explicitly in C, it will be implicit in the infrastructures we will see below (0MQ or MQTT [6] speak of *network byte order* for the organization of their fields coded on more than one byte, but the content itself is only a packet of bytes that the developer must organize properly), or nonexistent for ASCII transactions (XML-RPC) in which the order of the arguments is that of the exchanged ASCII strings.

Note that in the examples we will discuss below, the exchanges take place within the same computer (127.0.0.1) and since a processor is consistent with itself, any error in the byte order on transmission is corrected on receipt (two errors that compensate each other). In production, it is prudent to communicate with a machine of opposite endianness to identify potential sources of malfunction – Java is wonderful for this and we will be careful not to include it in our test cases as this language is unfamiliar to us.

As opposed to TCP, which guarantees transactions, UDP just sends packets to whoever wants to hear them. Thus, Fig. 4 presents a GNU Radio Companion processing chain that simply broadcasts single precision floating point numbers (float symbol in orange in GNU Radio Companion), while at the other end GNU Octave (left) or Python (right) executes

```
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()
```

to open the UDP socket on port 2000 of the local computer (on which GNU Radio writes), display the acquired data after converting the byte packet into a vector of floating-point numbers, and close the socket. This perpetual opening/closing of the socket may seem questionable, but it is the best way we

have found to ensure that the processed data is the latest transmitted and not old data held in a buffer. In the case of UDP, even if some data is lost, it is not a problem since we guarantee obtaining a vector of the correct number of fresh data. The best way to synchronize the acquired data with a physical event such as the rotation of an antenna is to send a command to GNU Radio to perform the action, wait for the necessary time for the command to be completed or ideally an acknowledgment by a TCP communication in response to the request, then open the UDP socket and capture the desired number of data in this configuration, and repeat for all the envisaged configurations – for example, for a synthetic aperture radar, with all successive positions of the antennas.

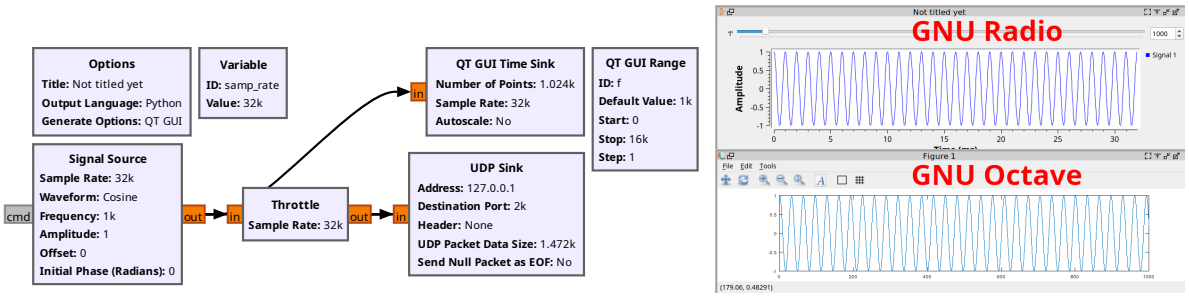
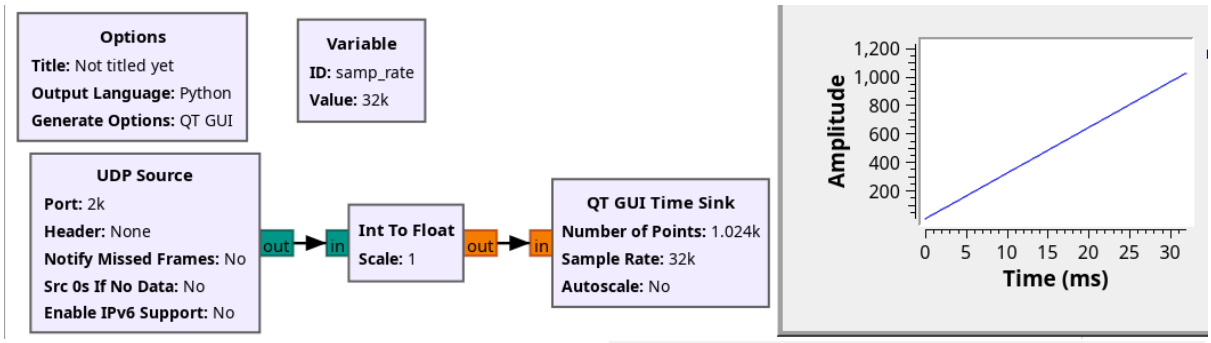


Figure 4: Left: signal processing chain feeding a UDP socket on port 2000 of the local computer (127.0.0.1) in order to share with any program that might be listening to process this datastream. Right: oscilloscope graphical output `Time Sink` of GNU Radio with the GNU Octave graphical output which has converted the received array of bytes to `float` with `typecast()`.

When launching the GNU Radio program transmitting the data on port 2000 of the local socket (127.0.0.1) in UDP mode, we can validate the emission of data using `netcat` with `nc -l -p 2000 -u 127.0.0.1` with `-l` to listen and `-u` for UDP.

Similarly, we can use GNU Radio to provide a processing chain for signals acquired by an interface broadcasting its information via UDP and utilizing the UDP Sink as proposed in Fig. 5, this time exchanging 4-byte (32-bit) encoded integers as indicated by the green icon in the GNU Radio Companion processing chain. In this example, we send a ramp from GNU Octave (`val=int32([k:k+1024]);v=typecast(val,'uint8`) on the left) or in Python (`numpy.arange(0+k,1024+k,dtype=numpy.int32)` on the right) but we could, of course, send any sequence of measurements, for example acquired by RS232 from an instrument.




```
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: Top: GNU Radio signal processing chain for fetching the data stream coming on port 2000 of the computer executing the script, and displaying its content. Bottom: GNU Octave (left) and Python (right) scripts generating ramps and send the data on a UDP port to be interpreted as 32-bit integer values.

Based on these lower layers of the OSI hierarchy, we will now explore some mechanisms organizing

transactions and facilitating client access to the interfaces exposed by the servers.

2 XMLRPC

A TCP/IP or UDP/IP client-server requires agreeing on the protocol for exchanging information between the client and server. In order to organize these transactions by encapsulating them in a format that can be easily processed automatically, it may seem logical to announce the nature of the required service and the associated variable. In the previous example, only the designer knows that the variable "var" exists with its associated read and define functions, and an external client may not be aware of the available function list. It seems natural to encapsulate the data in messages, and thus to encapsulate them in a syntax easily decodable by a computer, such as JSON or XML, at the expense, of course, of increased message size and dependency on message decoding libraries.

In the implementation of XMLRPC, calls to remote functions (Remote Procedure Call) are encapsulated in XML messages as described in [7]. In this way, a client only needs to inform the server of which service (function) it wants to use in order to modify its content (variable value). Just run "apt-cache search xmlrpc" under Debian/GNU Linux to see the multitude of languages implementing this protocol, especially Python. In this language, a client can be written in just a few lines

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

to modify the server's configuration that we defined in GNU Radio using the processing chain proposed in Fig. 6. In this processing chain, the frequency of the signal from the `Signal Source` block is a variable `freq` and we observe the effect of the remote command to the `freq` by the change in the period of the sine wave on the time domain graphical output.

In order to test the proper functioning of the server, it is not even necessary to write a line of code, since the shell command `xmlrpc` is provided in the package `libxmlrpc-core-c3-dev` of Debian/GNU Linux. This command can be used with the command `xmlrpc localhost:8080 set_freq i/1664` to modify the variable `freq` and assign the integer value (`i`) of 1664. Alternatively, a string can be transmitted by prefixing the argument with "s" or a floating-point number with "d" (the types are described in `man xmlrpc`).

According to the XMLRPC specifications [7], we can construct the message in XML format to send the value 1664 as a 4-byte integer to set the variable `freq` using its 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'
```

Thus, any language that does not support XMLRPC but can communicate via HTTP using the POST method can communicate with the server. Unfortunately, we did not succeed in achieving this result in GNU Octave, either due to the difficulty of interfacing with external libraries implementing a protocol (some authors report linking to Java libraries from GNU Octave to use their XMLRPC implementation, but we were unable to reproduce this result) or due to a poor implementation of `webwrite` that cannot fill the "data" field with an XML message of a POST request as described at `http://savannah.gnu.org/bugs/?56624`. We will see later (section 7) that we were led to learn how to link C functions to GNU Octave, providing a solution to implement any communication protocol and expose it to GNU Octave.

At the moment, we are not able to expose the variables used in a GNU Radio processing chain, as the request to the `listMethods` method of the client with

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

for method_name in proxy.system.listMethods(): # list of fonctions
    if (method_name.find("set_")>=0):          # provided by the server
        print(method_name)

try:
    setampl=proxy.set_ampl(0.2)      # fails (no ampl variable)
except xmlrpc.client.Fault as err:
    print("Unsupported function")
try:
    setfreq=proxy.set_freq(200)     # success, freq redefined
except xmlrpc.client.Fault as err:
    print("Unsupported function")
```

refuses to provide the list of variables. Therefore, only trying the different functions and intercepting errors (`try: ... except:`) allows to test if a variable exists or not. In the example above, `set_freq` is successful but `set_ampl` fails because the amplitude is not a defined variable in the processing chain.

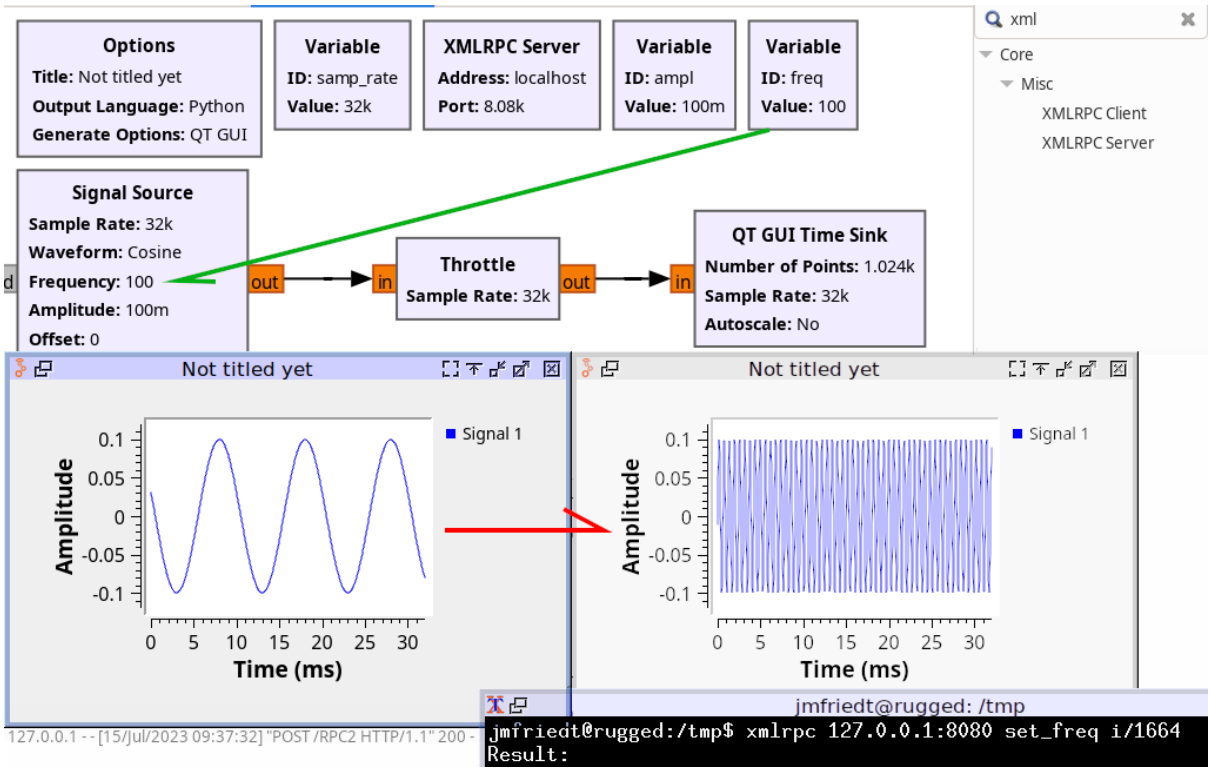


Figure 6: Reconfiguration of a parameter of a GNU Radio processing chain – therefore Python – through a command sent from the shell. The variable `freq` defines the frequency of the sine wave signal source, and therefore exposes the `set_freq` method which we call from the shell through `xmlrpc` (bottom right). The Python server acknowledges the reception of the command (bottom left) in the GNU Radio Companion console, and the frequency is effectively modified (red arrow) upon sending the command.

In order to expose the list of methods, `https://docs.python.org/3/library/xmlrpc.server.html` explains that we need to enable this functionality in the server. To do this, we add a Python snippet in GNU Radio containing the command `self.xmlrpc_server_0.register_introspection_functions()` to activate the ability of the server `xmlrpc_server_0` (associated block ID) to provide all services and consequently only call the variables that are actually defined (Fig. 7).

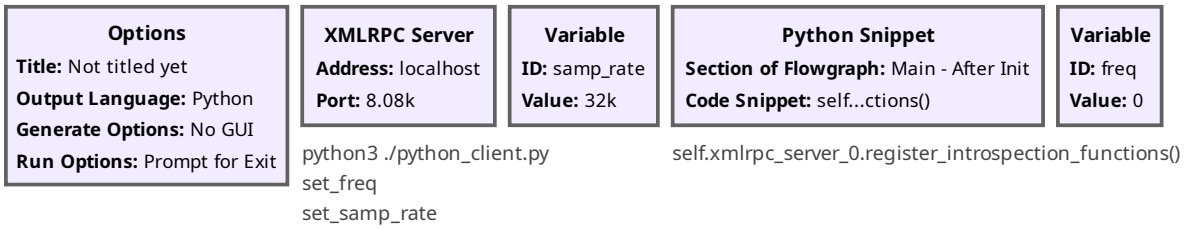


Figure 7: Activation of the `register_introspection_functions()` method in a Python Snippet in order to allow XMLRPC clients to fetch the list of variables known by the server.

We now have two approaches to define server parameters from the client side: a TCP/IP server or an XMLRPC server. These connected connections are appropriate to ensure that the message sent by the client is well understood and acknowledged by the server. However, it is common for the server to continue its acquisition and signal processing activities even if no client is listening to its services: this non-blocking connection is supported by UDP, which once again does not encapsulate information but only groups bytes to communicate to potential clients listening. If no one is listening, the information is simply lost, and if the routing changes during communication, neither the order nor the integrity of the data stream is guaranteed. In order to facilitate the organization of transmitted information, we will complete the previous demonstration of UDP exchanges by using a more abstract library called ZeroMQ (0MQ).

3 ZeroMQ

We first encountered ZeroMQ as a communication block in GNU Radio Companion, a Python code generator for digital processing of radio frequency signals. Barry Dugan, who is responsible for the documentation of GNU Radio Companion blocks, provided a detailed description of ZeroMQ in [8]. Here, we will limit ourselves to presenting implementations in Python, GNU Octave, and C, knowing that the proprietary software Matlab also supports this communication protocol, as do many other languages as indicated below.

```
$ 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
```

to name just a few.

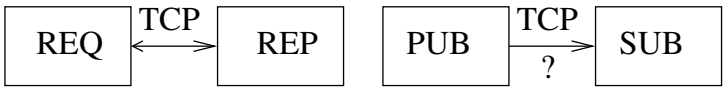


Figure 8: ZeroMQ (or 0MQ) provides an application layer above TCP/IP and UDP/IP to abstract the transmitted information and encapsulate it in packets with metadata. Like TCP or UDP, 0MQ provides a connected mode (REQ-REP) that guarantees blocking transactions, and a non-connected mode (PUB-SUB) that we will favor for our applications of transferring data acquired by software-defined radio receiver for remote processing.

Various works provide basic codes [9, 10], but either with errors or incompatibilities due to version updates: examining the numerous generations of documentation that are sometimes incompatible with current libraries can be tedious (transition from 0MQ version 3 from 2013 to version 4 from 2021), highlighting the danger of the evolution of these complex APIs for a long-lasting project.

Like TCP and UDP, ZeroMQ offers a connected mode that guarantees transactions – request-reply in ZeroMQ's terminology – but with a heavy bidirectional protocol for acknowledgement, and a mode for broadcasting information without guarantee of reception, which is the UDP datagram – named publish-subscribe at ZeroMQ (Fig. 8). An example of a server in C, requiring the installation of the package libczmq-dev on Debian/GNU Linux, is as follows:

```c
#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;
}
```

or in Python for a version sharing arrays of values (vectors) generated by NumPy

```python
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)
```

while a C client is written as

```
// 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;
}
```

or using GNU Octave if the `octave-zeromq` package was installed with

```
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")
```

or in Python as

```
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 for integers
  print(recv)      # vector1tmp=recv[0::2]
  plt.plot(recv)   # vector2tmp=recv[1::2] for interleaved
  plt.show()
```

We can convince ourselves that the PUB server continues its activity even in the absence of a client listening by observing the incrementing countdown every second.

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

clearly shows that even by interrupting the SUB client, the PUB server continues to increment its variable, which will go from 9 to 13 upon reconnection.

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

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

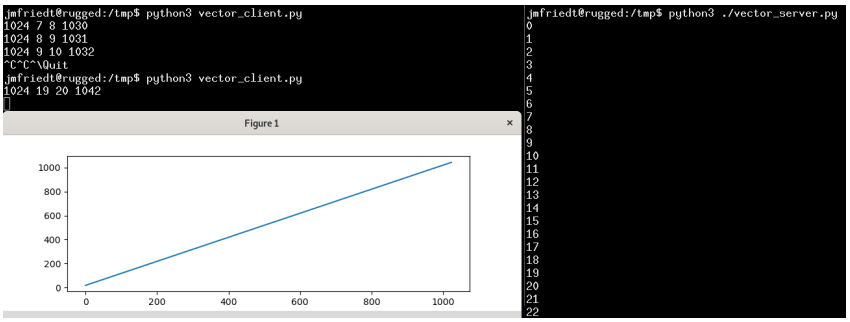


Figure 9: Left: illustration of a PUB server that continues its countdown whether the SUB client is connected or not. When reconnecting after an interruption by CTRL-C, the client resumes the countdown to the value reached by the server, with intermediate values being lost. However (right), if a wait that keeps the socket connected – for example, waiting to close the first display of `matplotlib` – separates two reads, the values transmitted during the wait are stored in a buffer and displayed, with an increment of 7 to 8 and then 9, while the server had reached a much higher countdown at the time of the last display, as we demonstrate by killing the client and relaunching it with a countdown that resumes at 19.

However, if the client does not disconnect its SUB socket, the sequence remains contiguous and the packets are stored until they are processed. The only way we have found to ensure that the data stream is the most recent and not remnants of a previous message – for example, when processing radio frequency signals acquired by a mobile antenna to ensure that the data is acquired at the new antenna position – is to close and reopen the SUB socket to eliminate the queue of pending data (Fig. 9).

On the other hand, in a REP-REQ scenario, a ping-pong game requires the REQuest client to ask the REPly server for new data (Fig. 10) and not to resend a packet without being invited to do so, otherwise risking receiving an error of the type "*Operation cannot be accomplished in current state*".

REQ Client

```
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]}"→
      ↪)
```

REP server

```
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)
```

Producing the data flow from GNU Radio makes prototyping more fun by continuously generating data, potentially from a physical radio frequency receiver but here from synthetic signals which therefore require a `Throttle` block to limit GNU Radio scheduler's data production to `samp_rate` samples per second. In Figure 11, we observe that the data produced in Python by the code generated by GNU Radio Companion is properly read in C (bottom-left) or Python (bottom-right) which allows displaying the waveform using `matplotlib` and verifying its adequacy with the sawtooth pattern produced, but with a variable number of transmitted data as imposed by the GNU Radio scheduler. Indeed in this context, we cannot make assumptions about the length of the transmitted vectors and will either have to wait to accumulate enough points to process the data (for example convolution or Fourier transform which require a known number of values in a vector to calculate the integral) or only process the useful subset and store the other values.

Similarly, the communication between GNU Radio (using Python) and GNU Octave is demonstrated in Fig. 12, which highlights the necessity of explicitly converting the byte packet produced by the publisher in order to interpret it correctly, either as floating point numbers (`single`) or as complex numbers for an IQ data stream, or as integers (`int32` for 4 bytes/integer) – the list of arguments

```
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: Exchanges between client and server in which each vector is required by the client REQ to be provided by the server REP, ensuring transaction sequencing and absence of data loss.

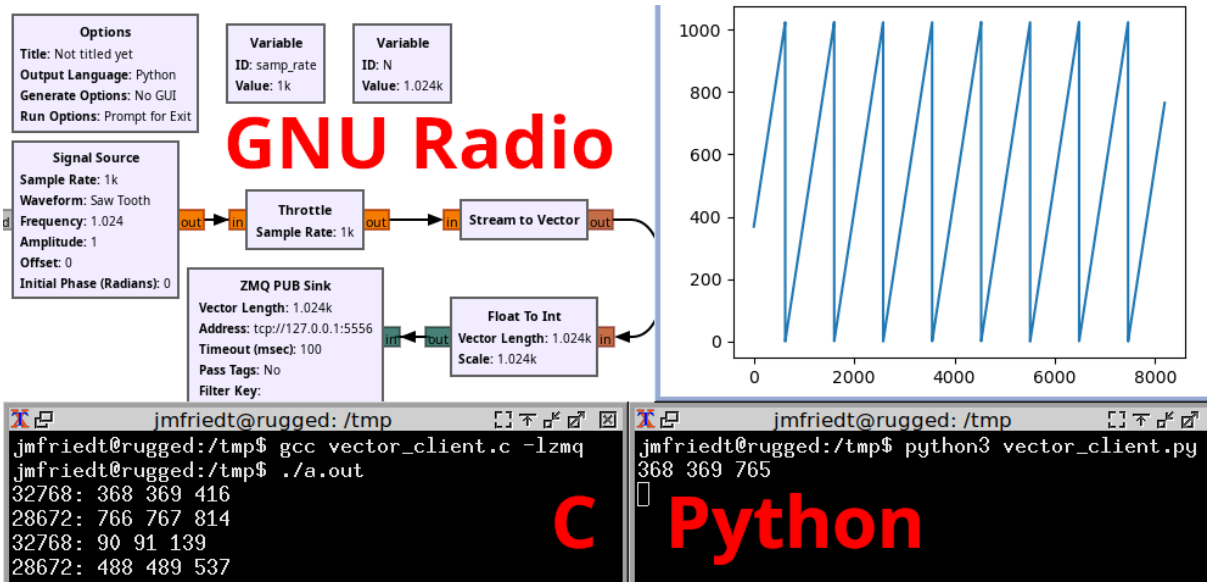


Figure 11: Generation of a sawtooth signal by GNU Radio (*Signal Source* of shape *Saw Tooth*, amplitude 1 and frequency 1.024 produced at a rate of 1000 samples/s and unit amplitude multiplied by 1024 when converting from floating-point number – orange symbol – to 32-bit integer – green symbol – transmitted in 0MQ PUB and received in Python at the bottom right by a 0MQ SUB for display using `matplotlib`, validating the consistency of the transaction, and in C at the bottom left, confirming that multiple clients can simultaneously receive the published data vectors.

supported by GNU Octave's `typecast` is provided by `help typecast`. This conversion of the byte packet to the appropriate type is also valid for Python with the `array` argument as documented at `https://docs.python.org/3/library/array.html`. Therefore, it is the responsibility of the developer to ensure that the exchanged data types are consistent, or to implement a protocol that guarantees transaction coherence if the nature of the exchanged data is likely to vary.

0MQ allows for easy exchanging information either through connected or broadcasted protocols, with the concept of topics that clients can subscribe to in order to only keep a subset of the transmitted information. However, it should be noted that the information is broadcast in clear text, which guarantees efficiency but also comes with the risk of easy manipulation of IP packets nowadays (Fig. 13). In the output of `tcpdump`, we can easily recognize the 32 bytes of the IPv4 header starting with 0x40 [2], along with a TCP transaction (the "06" in 0x4006, with the first nibble being the maximum packet time-to-live TTL set to 64), the source IP and the destination IP (0x7f000001 or 127.0.0.1 in decimal), the communication port 0x15b4 which equals 5556 in decimal, and so on.

Indeed, in the context of broadcasting data from a radiofrequency receiver, eavesdropping on the transmitted data is probably of little importance, but injecting erroneous data could be catastrophic. It is not so much the obfuscation of the data that is important, but rather their integrity, which could

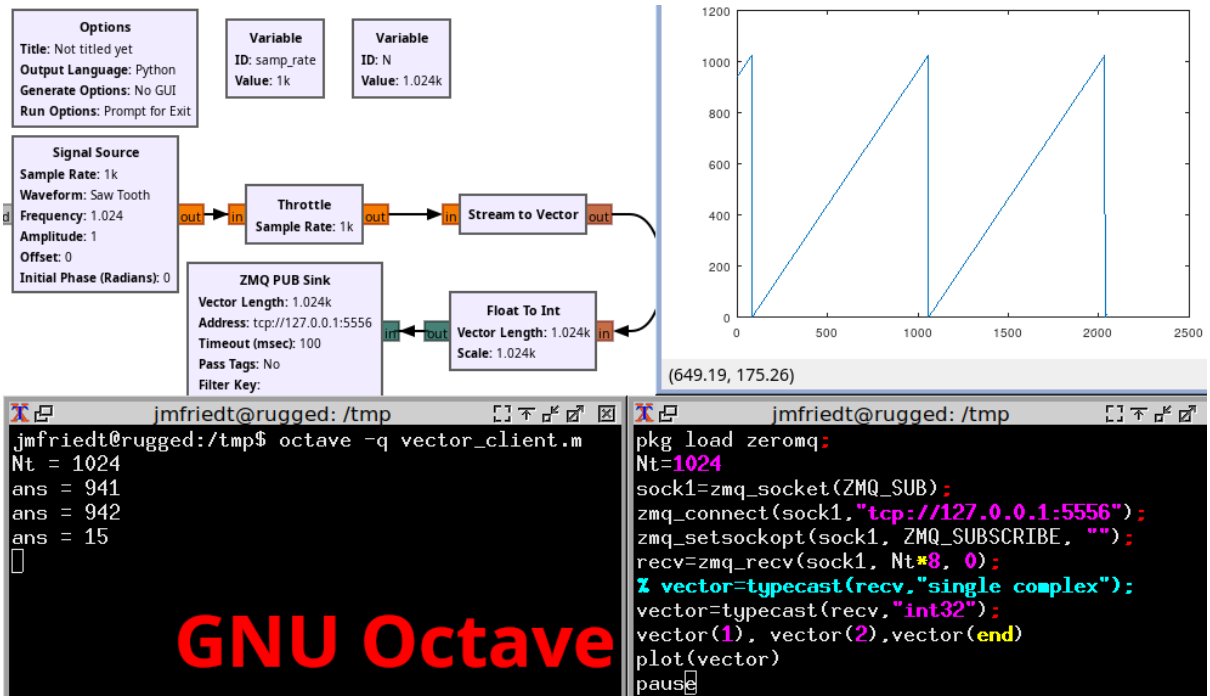


Figure 12: Generation of a sawtooth signal by GNU Radio (*Signal Source* of shape *Saw Tooth*, amplitude 1, and frequency 1.024, produced at a rate of 1000 points/s and unit amplitude multiplied by 1024 when passing from floating-point number – orange symbol – to 32-bit integer – green symbol – transmitted through 0MQ PUB and received by GNU Radio through 0MQ SUB for graphical display, validating the coherence of the transaction.

justify an additional layer of authentication that is clearly missing when we display the transmitted data to the SUB client in C using `tcpdump`.

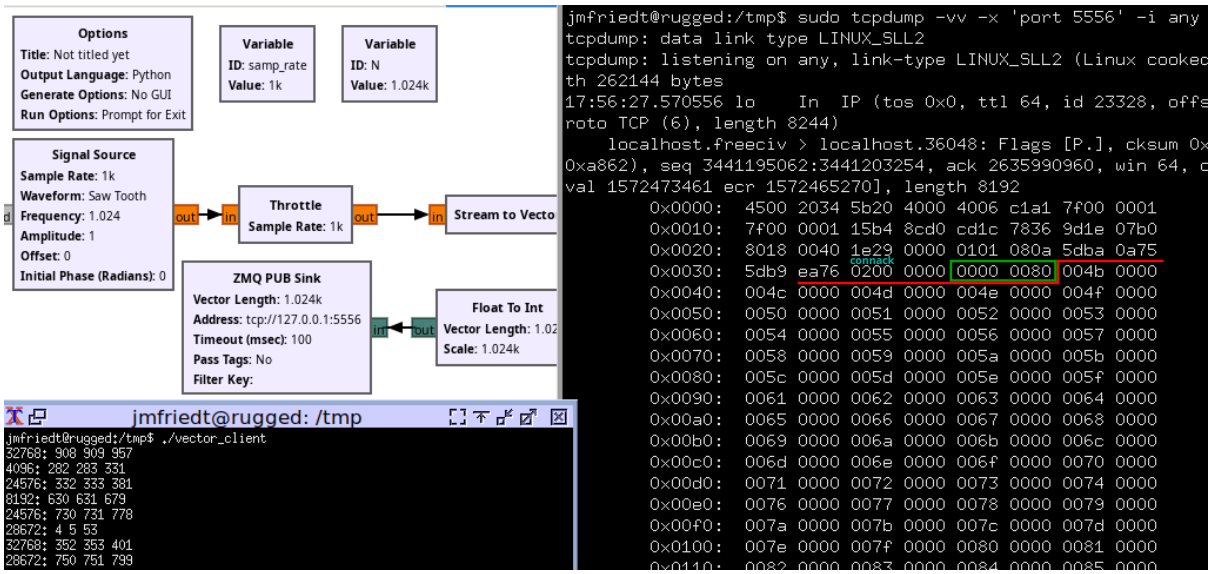


Figure 13: Observation using `tcpdump -vv -x 'port 5556' -i lo` (more selective than `any`) of the data transmitted by 0MQ: after the 32 bytes of the IP header, we can see that the overhead of the 0MQ protocol is minimal, with some information about the nature of the transaction (CONNACK for a server-to-client connection), the payload size (green) 0x8000=32768 (remember that the order is little endian on the network), followed by the clear text data (red delimitation), in accordance with the information provided by the server in C (first transaction).

4 MQTT

MQTT (*Message Queuing Telemetry Transport*) is emerging as a protocol for the "Internet of Things" (IoT), where embedded systems are designed to communicate. The need for a TCP/IP stack to implement MQTT is surprising in this context: IP, ICMP, and UDP took up only a few kilobytes of RAM and flash, much less memory than is needed to store TCP packets that must accumulate in case of packet loss or rerouting that would change their order. It seems that the main benefit of MQTT, beyond its centralization on a single server (referred to as a *broker* in MQTT terminology), is the SSL encryption of the exchanged packets (once again at the expense of the client's computing power seeking to transmit its information).

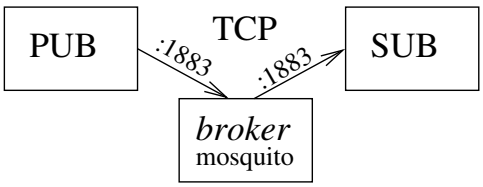


Figure 14: MQTT relies on a data hub – the *broker* – which centralizes the exchanges, and even though we remain in a Publish-Subscribe model with published topics that a client may or may not subscribe to, this time all participants are called clients that connect to the server which is the *broker* that communicates on port 1883.

Similar to 0MQ, MQTT operates in a publish-subscribe context, but this time not in a point-to-point connection but through a single *broker* that centralizes the transactions and therefore appears as a weak point in the network (Fig. 14). Thus, every service in MQTT is implemented as a client, whether a *publisher* or *subscriber*, with a single server that is the *broker*. Each publisher can offer services through filters, and each subscriber can filter the information it wants to process. This mechanism appears inefficient in context of exchanging radiofrequency data streams that aim for efficiency, but `https://opensource.com/article/18/6/mqtt` provides a concrete example of the use of these functionalities in the analysis of energy production in the state of New York, where a very rich dataset is slowly made available by electricity producers and accessed via MQTT with appropriate filters according to a hierarchy resembling a file system structure.

4.1 MQTT for Python, bash and C

An implementation of the MQTT *broker* is called `mosquito` and this library is the one we will use once installed with `sudo apt install mosquitto mosquitto-clients` on Debian/GNU Linux. On the client side, an implementation of MQTT, also promoted by the Eclipse Foundation [11], is called Paho and provides compatibility for a multitude of languages including C and Python. On Debian/GNU Linux, we therefore install `sudo apt install libpaho-mqtt-dev` for C, identified by searching for which package in the distribution provides the header describing the content of libraries `MQTTClient.h`, and `python3-paho-mqtt` for Python.

First, we make sure that a *broker* is running on the Linux operating system to allow these developments, either by `ps aux | grep mosq` which should indicate
`/usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf`, or by reading the content of
`/var/log/mosquitto/mosquitto.log` as an administrator. The communication between this server and MQTT clients will take place through port 1883, which must be open through any network transaction protection mechanism (firewall). We validate the proper functioning of the *broker* from the command line through some simple publish-subscribe exchanges:

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

Convinced of the broker's good operating conditions, we start by implementing an exchange of data vectors, always ensuring the coherence of the exchanges since MQTT simply transmits byte arrays without encoding the data organization: in C, the publish service (which we will not call server considering the broker) is coded as

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

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

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);
    printf("Waiting for up to %d seconds for publication\n",(int)(TIMEOUT/1000));
    rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
    printf("Message with delivery token %d delivered\n", token);
    MQTTClient_disconnect(client, 10000);
    MQTTClient_destroy(&client);
    return rc;
}
```

which is compiled using `gcc source.c -lpaho-mqtt3c`. The same result is achieved with Python with

```python
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)
```

and in both cases we verify that a transaction is executed by publishing data received by the shell command `mosquitto_sub -t "float_vect"` with a subscribe service that subscribes to the `float_vect` stream as we have selected when creating the publisher. However, since `mosquitto_sub` does not know the nature of the data structure being transmitted, it only displays bytes as ASCII characters that have no meaning. Therefore, we need to write clients – *subscribe* – able to decode the transmitted information, for example by taking inspiration from `https://eclipse.dev/paho/files/mqttdoc/MQTTClient/html/subasync.html` in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include "MQTTClient.h"
#define ADDRESS     "tcp://localhost:1883"
#define CLIENTID    "ExampleClientSub"
#define TOPIC       "float_vect"
#define QOS         1
#define TIMEOUT     10000L

volatile MQTTClient_deliveryToken deliveredtoken;
void delivered(void *context, MQTTClient_deliveryToken dt)
{
    printf("Message with token value %d delivery confirmed\n", dt);
    deliveredtoken = dt;
}
int msgarrvd(void *context, char *topicName, int topicLen, MQTTClient_message *message)
```

```
{
    int i;
    int32_t* payloadptr;
    payloadptr = (int32_t*)message->payload;
    for(i=0; i<message->payloadlen/sizeof(int32_t); i++)
    {
        printf("%d ",payloadptr[i]);
    }
    putchar('\n');
    MQTTClient_freeMessage(&message);
    MQTTClient_free(topicName);
    return 1;
}

void connlost(void *context, char *cause)
{
    printf("\nConnection lost\n");
    printf("     cause: %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, msgarrvd, delivered);
    if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
    {
        printf("Failed to connect, return code %d\n", rc);
        exit(EXIT_FAILURE);
    }
    printf("Subscribing to topic %s\nfor client %s using QoS%d\n\n"
           "Press Q<Enter> to quit\n\n", TOPIC, CLIENTID, QOS);
    MQTTClient_subscribe(client, TOPIC, QOS);
    do
    {
        ch = getchar();
    } while(ch!='Q' && ch != 'q');
    MQTTClient_disconnect(client, 10000);
    MQTTClient_destroy(&client);
    return rc;
}
```

or in 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)
```

which both highlight an interesting mechanism of automatically invoking a callback function when receiving a packet without the need to explicitly use a separate thread, as the C convention wouls require all reads to be blocking.

4.2 MQTT for GNU Octave

MQTT is not provided as a Debian/GNU Linux package for GNU Octave, but it can be installed without issues from the available source archives at `https://sourceforge.net/p/octave-mqtt`. Indeed, from the source download directory, running `make dist` will create a `.tar.gz` archive in the `target` directory. Then, in GNU Octave, we execute the command `pkg install target/octave-mqtt-0.0.3.tar.gz` to install `octave-mqtt` in `$HOME/.local/share/octave/api-v57/packages/mqtt-0.0.3/`. This new

package now provides access to the functions necessary to connect to the server (*broker*), and thus all transactions are seen from the perspective of a client: for a subscribe connection that receives the 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)
```

and for a publish connection that sends:

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

However, GNU Octave does not allow strong typing of the exchanged variables and we observe that the emitted array is in the form of floating-point numbers expressed in double precision (thus 8 bytes per data) and that from Python's point of view, this array is read by modifying the callback function using

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

hence a conversion of the byte array to double precision floating point thanks to `dtype=numpy.float64`.

Finally, we conclude this overview of MQTT by highlighting the simplicity of integrating a Python library into GNU Radio: indeed, `https://github.com/crasu/gr-mqtt` offers an interface between GNU Radio and MQTT by simply encapsulating the functions that we have just explained in the `work` method of a dedicated Python block compatible with GNU Radio calls.

5 Unix filesystem

By addressing mechanisms involving sockets – the source of divergence between the implementation of Unix that exposes network interfaces according to a specific API [12], and its original philosophy of "everything is a file" – we have omitted the probably simplest approach to communicate between two processes, the *pipe*. In fact, if we create a pseudo-file that communicates its input with its output through `mkfifo /tmp/myfifo`, then any data entering the pipe will be accessible to any process connected to its output as if it were a regular file, but without data storage on physical media. A special case would be the link between the `stdout` of one process and the `stdin` of another process using the symbol `|`, but here we will focus on the case of files as accessible by `open()`, `read()`, `write()`, and `close()`.

To ensure proper functionality, launch the GNU Radio program shown in Fig. 15, and observe that nothing happens (no display on the `Time Sink` output oscilloscope) until the other end of the pipeline is connected. However, by running `cat < /tmp/myfifo` in a terminal, we can see that the scheduler starts generating data, the binary representation of which is displayed in the terminal. Furthermore, using GNU Octave, we can perform the same operation with the following code:

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

which code opens the file once, and then continuously reads the last 1000 floating-point numbers (implicitly encoded as 4 bytes, i.e., 4000 bytes) in order to display their contents. By manipulating the slider bar that varies the frequency of the signal, we can observe increasing latency, as all the data cannot be consumed in real time. We attempted to close and reopen the file within the loop, but this did not change the fact that the pipeline does its job by storing all injected data until it is consumed.

Note that `/tmp/myfifo` must be created *before* launching GNU Radio; otherwise, a true file (without the `p` attribute in the first field of `ls -la /tmp/myfifo`) will be created in its absence and gradually filled without providing the expected result.

This approach is elegantly presented for cascading data in a software-defined radio processing chain during the 2023 session of the Software Defined Radio Academy, available at [13].

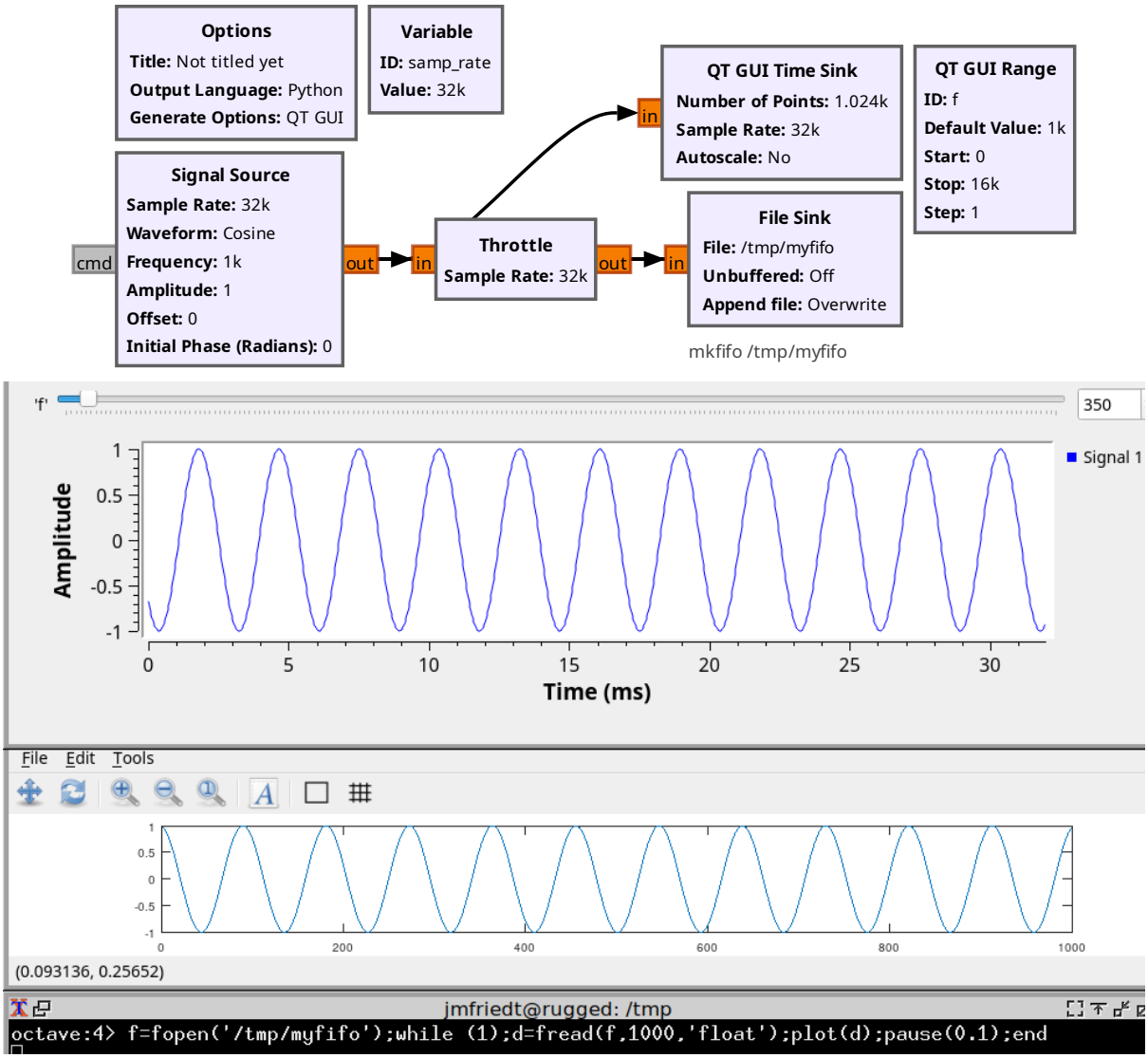


Figure 15: A simple GNU Radio Companion processing chain (top) produces a data stream at a rate of approximately 32000 samples per second (block `Throttle`) and feeds it into a file connected to a FIFO. At the other end, GNU Octave opens this file, reads its contents, and displays it on a graph as fast as possible (bottom). The oscilloscope (middle) allows validation of when the data is produced by GNU Radio during the execution of the p rocessing chain.

6 C in Python: `ctypes` and pybind11

So far, we have explored data sharing through sockets, making communication transparent within a single computer or across computers connected through a network, in order to share data acquisition and processing between various tasks potentially written in different languages. However, an alternative to leveraging the advantages of different languages without using a socket is to create executables capable of invoking a dynamic library or a binary version of the compiled code from the interpreted language. Our colleague Benoît Dubois (FEMTO-Engineering) uses `ctypes`, a library for i calling C functions from Python, bringing together the speed of a compiled language with the flexibility of an interpreted language. In doing so, we wondered about the coherence of the memory areas addressed by each language and whether the data structure is shared or duplicated. To do this, various C functions manipulating various types of pointers are defined in a file `t.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");}
```

compiled to an object using the command `gcc -c t.c -fPIC`, and then creating a dynamic library

named `t.so` from the object using the command `gcc -shared t.o -o t.so`. We check the contents of the library by using the command `nm -D t.so` to ensure that it contains the functions we have defined. We observe that we want to pass a character pointer (also called a string, hence called with arguments between "..." in all languages), a pointer to an array of floats (such as a vector or matrix in C), a pointer of an undefined type often used when the nature of the argument is not specified at compile time (such as a data structure), and finally a procedure with no arguments.

This C library is called from Python using `ctypes` with

```
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()
```

to display when executed

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

demonstrating that the pointer is the same in the structure created by NumPy and the one received by the function `fonctiond()` in C.

Recently (since its version 3.9), GNU Radio has decided to expose its processing blocks in C++ to Python using pybind11, a technique that leverages C++ features to link this language with Python at compile time. The example above becomes almost compatible with Python by declaring the functions in a file that we will name `tpybind.cpp` containing

```
#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");
}
```

and despite our disgust at `#include` a C source code, this program is compiled with

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

to produce a file with extension `.cpython-311-x86_64-linux-gnu.so` that we copy into `/usr/lib/python3.11/lib-dynload` to make it accessible. It should be noted that the consistency of names between the library that will be loaded in Python by `import tpybind` and the first argument of `PYBIND11_MODULE` is important: it must be the same name [14]. By doing this, we will be able to execute in Python

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

and indeed the passage of an array (pointer) of floating point numbers does not seem to be supported when we consult the list of arguments of pybind11 in [15]. A method called `memoryview` seems to be designed to expose the content of memory of a structure in Python to C/C++ functions, but we were not able to tell it how to feed the pointer of floating point numbers from `functiond()`.

7 C in GNU Octave: SWIG and mkoctfile

SWIG (*Simplified Wrapper and Interface Generator*), an old method used by GNU Radio to export its C++ libraries to Python before switching to pybind, was introduced by William Daniau in his presentation on interfacing C++ functions with various interpreted languages [16], but Octave is not

among the considered tools. We were able to easily interface functions accepting character strings (`char*` pointer) or functions written in C without arguments with an Octave using

```
%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*);
%}
```

which is compiled using `swig -octave t.i` to produce `t_wrap.cxx`, which is then compiled into a library recognized by GNU Octave using `g++ -c -I/usr/include/octave-7.3.0 -fpic -std=c++20 t_wrap.cxx`, and linked into an executable using
`gcc -shared t.o t_wrap.o -L/usr/lib/x86_64-linux-gnu/octave/7.3.0 -rdynamic -loctinterp -loctave -lpthread -lm -o t_wrap.oct` (where the object `t.o` is the same as before when illustrating how to link with Python using `ctypes`), which allows calling it from 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)
```

to display the address of the argument of the function `fonctionc()`. However, once again, as was the case with pybind, we were unable to pass a pointer to an array of numbers (integers, floats) since the GNU Octave matrix is a complex C++ class representing the properties of the array in addition to its content.

However, GNU Octave offers native interfacing with C/C++ functions through `mkoctfile`. Thus, a trivial program inspired by `https://docs.octave.org/latest/External-Code-Interface.html` in the form of

```
#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));
}
```

is compiled using `mkoctfile pointeraddr.cc` (the extension is important because `mkoctfile` selects `gcc` or `g++` according to the extension `.c` or `.cc`) to produce a file with the extension `.oct` whose name must be the same as that of the function. When running under GNU Octave

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

we see that the pointer address is correctly displayed and returned to the interpreter. The documentation warns of some possible subtleties when including C in C++ `https://docs.octave.org/latest/Calling-External-Code-from-Oct_002dFiles.html` that are typically encountered when these two languages coexist.

8 Conclusion

We have endeavored to demonstrate how to communicate between different languages in order to distribute processing either by making the most of each language, or by sharing resources across separate computers. To do this, we explored XMLRPC, 0MQ and MQTT for socket-based communication, or `ctypes`, pybind, and SWIG for integrating C functions into GNU Octave. Many more mechanisms have been proposed to the point of getting lost, with more or less advanced debugging capabilities: for example, since version 3.9, GNU Radio has decided to abandon SWIG in favor of native integration of C++ with Python through pybind (`https://pybind11.readthedocs.io/en/stable/basics.html`), a transition that did not happen without pain, breaking compatibility with all existing processing blocks. Thus, the choice of the right infrastructure will guarantee the sustainability of developments and the continuity of a project... until the next incompatible API update!

One point we did not address in this presentation is the use of `websockets` as an alternative to the native POSIX sockets, but at the highest application layer of the OSI model. André Buhart (F1ATB) mentions this approach in *RemoteSDR* at `https://f1atb.fr/index.php/2020/07/19/gnu-radio-to-web-client/`. The reader is encouraged to explore this path if cross-platform compatibility is required.

All the programs proposed in this article are available at `http://github.com/jmfriedt/gnuradio_communication`

References

[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] POSIX standards, section *Sockets* at `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) in page 10/85.

[6] *MQTT V3.1 Protocol Specification* (2010) at `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) at `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] Eclipse Paho Downloads, à `https://eclipse.dev/paho/index.php?page=downloads.php`

[12] J. Train, J.D. Touch, L. Eggert & Y. Wang, *NetFS: networking through the file system*, ISI Technical Report ISI-TR-2003-579 (2003) at `https://www.strayalpha.com/pubs/isi-tr-579.pdf` and of course the Plan9 description in R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey & P. Winterbottom, *Plan 9 from bell labs*, Proc. Summer 1990 UKUUG Conference, which states "*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.*"

[13] J. Ketterl DD5JFK, *OpenWebRX*, Software Defined Radio Academy (2023) at 4h55 of `https://www.youtube.com/watch?v=yFFFAMRQHP4` and in particular communication through *pipes* at 5h10

[14] Note that adding the name of the processing architecture on which compilation is performed to the library name is source of much sorrow when cross-compiling, for example in Buildroot, as discussed at `https://github.com/gnuradio/gnuradio/issues/5455` and associated links.

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

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