Interfacing with hardware

Ronald Phlypo

v.20220314

Abstract

This document describes a high-level, object-oriented approach to socket programming and messaging in python. The goal is to read in parameters from a graphical interface to send to an Arduino and to receive data from an interfaced Arduino that will be used for live plotting.

What is messaging?

Messaging is initiated with the following elements in the basic SMCR approach (Claude E. Shannon, *'A mathematical Theory of Communication,'* The Bell System Technical Journal, Vol. 27, pp. 379–423, 623–656, 1948):

information source produces a (sequence of) message(s) to be sent to a receiving terminal

transmitter operates on the messages to produce a signal that is suitable for transmission

channel a medium used to transmit the signal from transmitter to receiver

receiver perform the inverse operation of the transmitter, i.e., decoding the signal to extract the messages

destination person or thin for whom the message is intended

One can find a schematic view in Figure 1. Our implementation of this model will be based upon the following building blocks that will be detailed next:

incoming data this can either be data generated on the Arduino that has to be transmitted for further processing, or data coming from a graphical interface that needs to be sent to the Arduino for controlling the hardware. This is our information source.

zeromq protocol this is the low-level protocol that will distribute the message on the system, at the transmitter side the published message will be packed into a *dictionary* and will carry a *topic*. It is also possible to transmit numpy data directly.

socket software interface with the system's services through which one can exploit network traffic (this will be our channel). We will use the *backplane* from python_banyan.

zeromq_protocol unpacking the dictionary if the receiver is subscribed to the topic is done using the zeromq protocol.

data exploitation either plotting the data (live updates) or controlling the hardware using the received data or parameters.

Banyan as a high-level messaging protocol

Python banyan (user guide) is an object-oriented approach to message passing, this includes message packing and unpacking as well as sending/receiving. The best to get acquainted with the programming approach is through examples. For our use cases, we will always deal with the same global approach

1. set-up the backplane, initialise the network ports on your machine to exchange data through sockets
2. set-up multiple clients (subscribers) and servers (publishers) that communicate through the single backplane using a publish–subscribe protocol.

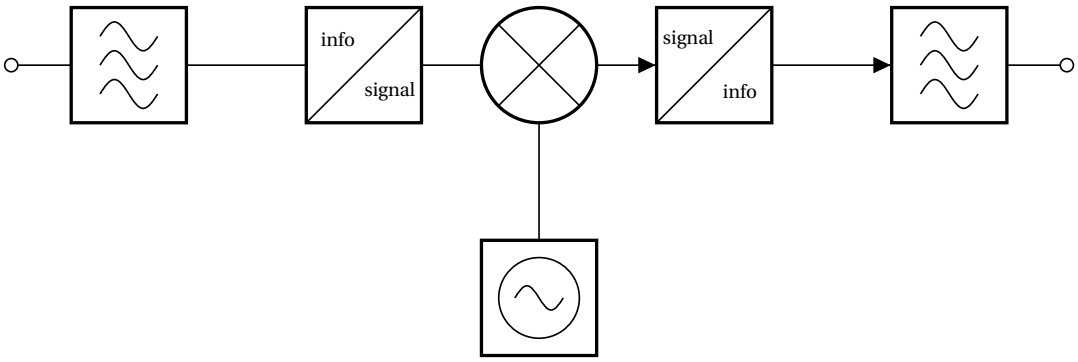

Figure 1: A schematic view of the source, message, channel, receiver model of communication.

The sending/receiving of messages always occurs in an infinite loop (just as with your arduino communication using pymata-express). You will need to make sure that when data is received these are exploited correctly, or when data is available to publish, other components can access the data.

1. Install python_banyan using `pip install python_banyan` in your conda environment.
2. Go to python_banyan (GitHub) and download the repository containing the examples in your working directory.
3. Get the first example *A Simple Echo Server/Client* running (Tutorial 1).
   (a) Try to change the number of messages sent.
   (b) Identify the two topics and get to know how these are used. Then try to change the *topic* of the client messages in `set_subscriber_topic` or `publish_payload`. What happens if the topic used by the server does no longer match with the one from the client?
   (c) Change the `loop_time` of the server and the client by adding it as an argument to `super(EchoClient, self).__init__` or `super(EchoServer, self).__init__`. What mechanism has been implemented that prevents messages to be dropped even though the loop times could be different (this is closely related to a *handshake* in network protocols)?
   (d) What happens if you change the payload on the server side to a random integer uniformly drawn from the set $[\![0, 10]\!]$?
   (e) Identify the different entities of Figure 1 in this communication process.
   (f) Set up a second client which has the exact same properties as the first (download the original versions again if necessary). How easy is it to perform the exact same task in two different clients? How would you obtain the two clients to have the server print different messages according to their 'ID' (make sure your solution can be generalised easily to n clients).

You may have observed that messages exist of two components: the payload and the topic. This allows for a central backbone (*backplane*) to receive all messages and alert all connected servers and clients about new messages. It is up to the client or server to filter the messages sent by the backbone based on the topic (think about filtering your emails in your inbox, all but the topics to which you subscribed are considered *spam*). You may have noticed that the naming *server* and *client* are somewhat arbitrary, since both have symmetric roles. However, we may consider that a server is always up and running before any client. Clients subsequently connect to the server and need to notify the server(s) that they are up and running.

Can you identify where this notification takes place in *A Simple Echo Server/Client*?

Plotting data: a blocking process

As long as processes are non-blocking, everything goes fine, but when high-rate data throughput is required and a blocking process (i.e., the process does not allow for any other activity than its own) is running, things become more involved. This is the case when one would like to plot data. The plotting itself does not allow to simultaneously collect data samples. One thus needs to manage two processes that run at different temporal scales: the slow plotting process

and the rapid data collection. In order to manage both high-rate data collection and low-rate plotting without (too much) data loss, one needs to buffer data at both the data collection and plotting level. In a first approach we will consider an infinite memory buffer at the data collection side, i.e., all data can be created on demand, whatever the time horizon.

Our implementation scheme becomes:

data collection server Data collection happens either at a fixed sample-rate that is equal to the loop_time (in polling mode), or at irregular time stamps (when the sample collection is event-driven or callback based). In either case, a local memory will be filled with the collected data and will be flushed every time the data has been sent to the plotting server. Publishing of data will be done every time the plotting client acknowledges data reception (or when a time out is reached). One needs to take care to adapt the buffer size to the data request intervals in order to not loose to much data.

plotting client It has a screen refresh rate of 1s, which will be associated to the loop_time. Each time the server sends its currently available data from the data collection it will be added to its local memory (One may use a circular buffer or a limited-size FIFO buffer to avoid memory overflow. Implementations of the circular and FIFO buffer are available from Chamilo in `buffers.py`). The data will be received as tuples (timestamp, data), where data can be any valid python data structure. The parameter `buffer_size` should be adapted to a (hard-coded, your choice) parameter `display_time` which will determine the time that is displayed as well as to the sampling period of the data collection (in case regular sampling is used).

Rightly setting up both a plotting server and a data collection client thus needs a data exchange protocol, which is detailed here:

data server	**plotting client**
awaiting clients to connect (idle)	
protocol 1: initialisation	
	up and running, communicates `refresh_time` parameter
local buffer based on `refresh_time` and `sample_period`	
communicates `sample_period`	
	local buffer based on `display_time` and `sample_period`
protocol 2: data exchange request	
	fire data request REQ
send data, clear buffer	
	add received data to local buffer
server: repeat in `receive_loop` (use `receive_loop_idle_addition`)	
add data sample to buffer	
client: repeat in `receive_loop` (use `receive_loop_idle_addition`)	
	request data [REQ]

After an initialisation, the plotting client will poll the data collection server every time when plotting is finished (or earlier if no plotting is done) to add the data to its local memory. The data collection server will then transfer all its data currently in its local memory and flush its memory.

Implement the above scheme with a data collection server that adds a sample of a sine wave of 0.25Hz every 1ms. The plotting server should be able to display a window of 4s that refreshes every 1s (inpire yourself by the examples given in buffers.py).

1. Do so with a regularly sampled signal at the data collection client using circular buffers on both sides.

2. What changes are needed if one would implement this using event-based callbacks? Are circular buffers still optimal?
   To test, at the data collection server side, add the realisation of a normally distributed random variable every loop_time (mean 0, variance $\sigma^2 \times \Delta t$, where σ^2 is a user-defined value and Δt is the loop_time) to the current value and add an event to the buffer only if the absolute difference of the value recorded at the last event and the current value exceeds a user-defined positive threshold ε (this is known as a simple diffusion process).

You may inspire yourself by the examples given in the `buffers.py` file (this example uses buffers, but not the server-client communication).

You might notice that there is some temporal overhead for sending data between different processes. If one works on the same machine and has multiple processors, then the `shared_memory` from the `multiprocessing` library may come to the rescue (optional).

Integrating Banyan into an existing loop

Clients and servers can be set-up at any time and are kept alive as long as they have a job that is running. If you would like to plot data, you may want to start up a specific server that is only plotting live streamed data (plotting server). A client could interface with the Arduino, and so on...

Communicating with the Arduino (the pyMata4 loop)

Make sure you have an object-oriented approach to the interface with the numpad connected to the Arduino. Have a look at the Banyan integration with GPIO as an inspiration to set up a communication server to interact with your Arduino through the Banyan backplane (no need to use GPIO, you may just stay with your pyMata4).

We'll set up two communication entities, the Arduino server (running pyMata4) and a login client that will check the four-digit code and send back a validation or refusal to login on the Arduino. At first the Arduino server will be a simple server (no connection to Arduino) just sending over some four-digit codes (chosen randomly) for testing, only later will you integrate your hardware keypad.

1. Set-up the four-digit login client and a mock Arduino server with the following properties (increasing feature set, do so step by step)
   (a) the login client should be able to check for an incoming four-digit code (topic `credentials`) that is received through incoming messages and compare this with a hard-coded, local four-digit code;
      i. when the two four-digit codes match, a reply message is to be sent that confirms a successful login on the topic `status`.
      ii. if they do not match, a reply is sent that login was unsuccessful;
   (b) before sending over the code, the four-digit code will be hashed (use python's built-in hash tools in hashlib) and compared to the locally hashed four-digit code. What is the advantage of this approach?
   (c) [to go further] the login client will block for 1 minute if three consecutive failed logins are registered within the last minute.
2. connect your Arduino server to the Arduino and use the keypad to login. A status bit `unlocked` should be set to `True` upon successful login.

The graphics client

We will use plotly dash for the plotting (use `pip install dash` in your conda environment). The main reason is that it lives in a web-browser, that there are native methods to update the graphics regularly (live updates), and that there's a data acquisition toolbox (dash DAQ use `pip install dash_daq` in your conda environment) that facilitates visual interaction through buttons, sliders, gauges, and many more.

Run the livestream example, to do so you must alter the last line to read `app.run_server(debug=True, host="127.0.0.1")` and install pyorbital using `pip install pyorbital`

Take the example on the live updates and transform it into a class-based example with a `run` method that launches the *app*. Inspire yourself by the Tkinter example that is given as a Banyan example, especially its `get_message` method. The update interval should be set to once per second so as to allow the client to update the plot without blocking the data exchanges.

1. Set up a plotly dash graphic as a client (call this the plotting client)
2. Set up a server (call this the data server) with a random generator sending out blocks of random numbers (1000 points every second) via a topic to which the plotting client subscribes
3. Update the graphic every second through a live streaming update, showing only the last 4s of data.
4. Now replace the random generator in the data server by a dash DAQ component (such as a rotary button) and plot its data in a live stream (use events).

Setting up the Arduino interaction

You're up and running to interact with your Arduino hardware.

- Unlock the Arduino with a four-digit code on the keypad.
- Plot the angular data of the dc motor in a live stream. Make sure to add a button to pause and start acquisition (live streaming). Make sure to also stream the data of the command rotary button.
- Make a dash_daq interface that allows you to send parameter values to your Arduino board (sending to the Arduino data server which handles interaction with the board). A rotary button in the dash_daq should have the same effect as the hardware button (and should have no effect if the login was unsuccessful).
- Feel free to interact with the Arduino board using different visualisations and virtual buttons.