Aula 8 - PARI

Miguel Riem Oliveira <mriem@ua.pt> 2020-2021

Sumário

```
Comunicações TCP-IP
Introdução ao Robot Operating System (ROS)
```

Os sistemas robóticos são muito frequentemente sistemas complexos. Isto leva a que o desenvolvimento de um programa que cubra todas as funcionalidades necessárias seja uma tarefa longa e complicada.

Mais ainda, a utilização de um programa único cria dependências desnecessárias e indesejáveis entre funcionalidades, mesmo em programas organizados em multíplos ficheiros. A solução mais elegante para contruir programas complexos é ... não o fazer.

De fato, o melhor é manter os programas desenvolvidos simples e focados no menor número de funcionalidades possível. No entanto, isto implica que estes programas sejam capazes de comunicar entre si.

Neste contexto, a comunicação entre programas torna-se um elemento fundamental no desenvolvimento de sistemas robóticos avançados.

O Robot Operating System (ROS) é um conjunto de bibliotecas e ferramentas *open source* concebidas para facilitar o desenvolvimento de programas para robôs. Uma das grandes vantagens do ROS é a forma como facilita a comunicação entre diferentes programas (nós).

Recomenda-se a realização de todos os tutoriais ROS na secção 1.1 begginer level antes da realização dos exercícios desta aula. Pode eventualmente não fazer os tutoriais de *c++*.

Exercício 1 - Standalone TCP-IP Communication

Implemente este exemplo de comunicação TCP-IP pura (não baseado em ROS).

Neste exemplo, a comunicação ocorre entre um *programa servidor* e um *programa cliente*. O cliente envia de dois em dois segundos informação definida na lista *messages*. Note que a informação enviada é de vários tipos (numérica, texto).

server_test.py

```
#!/usr/bin/env python
# ------------------------------------------------
# Miguel Riem Oliveira.
# PARI, September 2020.
# Adapted from https://stackabuse.com/basic-socket-programming-in-python/
# ------------------------------------------------
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # create TCP/IP socket
local_hostname = socket.gethostname()  # retrieve local hostname
local_fqdn = socket.getfqdn()  # get fully qualified hostname
ip_address = socket.gethostbyname(local_hostname)  # get the according IP address

# output hostname, domain name and IP address
print ("working on %s (%s) with %s" % (local_hostname, local_fqdn, ip_address))
server_address = (ip_address, 23456)  # bind the socket to the port 23456

print ('starting up on %s port %s' % server_address)
sock.bind(server_address)

# listen for incoming connections (server mode) with one connection at a time
sock.listen(1)

while True:
    print ('waiting for a connection')
    connection, client_address = sock.accept()  # wait for a connection

    try:  # show who connected to us
        print ('connection from', client_address)

        while True:  # receive the data in small chunks (64 bytes) and print it
            data = connection.recv(64)
            if data:
                print ("Data: %s" % data) # output received data
            else:
                print ("no more data.") # no more data -- quit the loop
                break
    finally:
        # Clean up the connection
        connection.close()
```

client_test.py

```
#!/usr/bin/env python
# ------------------------------------------------
# Miguel Riem Oliveira.
# PARI, September 2020.
# Adapted from https://stackabuse.com/basic-socket-programming-in-python/
# ------------------------------------------------
import socket
import time

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # create TCP/IP socket
local_hostname = socket.gethostname()  # retrieve local hostname
local_fqdn = socket.getfqdn()  # get fully qualified hostname
ip_address = socket.gethostbyname(local_hostname)  # get the according IP address

server_address = (ip_address, 23456)  # bind the socket to the port 23456, and connect
sock.connect(server_address)
print ("connecting to %s (%s) with %s" % (local_hostname, local_fqdn, ip_address))

# define example data to be sent to the server
messages = [30, 'Robotics', 31, 14, 'Automation', 18]
for message in messages:
    print ('Sending message: ' + str(message))
    message_formated = str(message).encode("utf-8")
    sock.sendall(message_formated)
    time.sleep(2)  # wait for two seconds

sock.close()  # close connection
```

Do ponto de vista de um programador de sistemas robóticos, este exemplo acima tem várias desvantagens:

1. **Complexidade**: A primeira é que o código tem muitos pormenores e detalhes (large code footprint), exigindo uma boa dose de atenção do programador, que devia estar preocupado com a aplicação robótica e não com a comunicação entre módulos.
2. **Paradigma de comunicação**: Por outro lado, este programa está limitado à comunicação peer to peer entre um programa servidor e outro programa cliente. Este paradigma de comunicação, em que o programa cliente envia uma mensagem que tem obrigatóriamente de ser recebida, por vezes não ser o ideal.
3. **Topologia de comunicação**: Este programa funciona numa comunicação do tipo peer to peer. Quer isto dizer que apenas existem dois programas a comunicar um com o outro. A inclusão de programas adicionais no ecosistema de comunicação não é imediata.
4. **Serialização da informação transmitida** A informação que é enviada do cliente para o servidor tem de ser serializada. Quer isto dizer que a informação tem de ser colocada no formato de uma série de bytes consecutivos. Quando se usam estruturas de dados complexas (classes, dicionários, etc), isto implica um processo de conversão da informação (da estrutura de dados para um array de bytes, no cliente, e no sentido inverso do lado do servidor). Estes

processos têm o nome de serialize / deserialize ou marshalling / unmarshalling, e podem ser complexos de implementar.

Obviamente que existem várias formas de resolver os problemas listados acima. Por exemplo, para 2 e 3, existem bibliotecas que implementam vários paradigmas e topologias de comunicação (e.g. zeromq), e no caso 4 da serialização / deserialização, também há soluções dedicadas a este problema (e.g. google protocol buffers).

No entanto, o ponto aqui é que isto implicaria um grande esforço de implementação e debugging focado nos problemas da comunicação. Recorde que o objetivo principal era dividir um progama complexo em pequenos programas que teriam de comunicar entre si. Esta solução só será válida enquanto a comunicação entre módulos não impuser um grande aumento da complexidade do sistema, caso contrário o propósito inicial de simplificação é derrotado.

Exercício 2 - Exemplo de serialização

Este exercício tem o objetivo de detalhar um processo de serialização / deserialização. Partindo do Exercício 1, assuma que tem uma estrutura de dados complexa que é uma instanciação de uma classe *Dog*, declarada num ficheiro denominado *dog_lib.py*:

dog_lib.py

```
from colorama import Fore, Style

class Dog:
    def __init__(self, name, color, age):
        self.name, self.color, self.age = name, color, age
        self.brothers = []  # no brothers for now

    def addBrother(self, name):
        self.brothers.append(name)

    def __str__(self):
        return 'name: ' + Fore.RED + str(self.name) + Fore.RESET + \
               ', age: ' + Fore.RED + str(self.age) + Fore.RESET +\
               ', color: ' + Fore.RED + str(self.color) + Fore.RESET +\
               ', brothers: ' + Fore.BLUE + str(self.brothers) + Style.RESET_ALL
```

Do lado do cliente, crie uma instância da class *Dog*, adicionando alguns irmãos. e.g.:

```
import dog_lib
dog = dog_lib.Dog(name='Toby', age=7, color='brown')  # instantiate a new dog
dog.addBrother('Lassie')
dog.addBrother('Boby')
print('CLIENT: my dog has ' + str(dog))
```

Depois envie o conteúdo desta classe numa mensagem para o servidor. Terá de arranjar uma forma de colocar toda a informação contida na classe na mensagem a enviar.

Depois, **do lado do servidor**, a mensagem deverá ser descodificada e deve ser criada uma instância da classe *Dog* que espelhe a existente do lado do cliente.

Imprima as instâncias nos dois programas para confirmar que são cópias exatas.

Exercício 3 - Publicação e subscrição em ROS

Crie um novo pacote ROS com o nome pari_aula8_ex3. oo

O package deve depender do *rospy* (todos os que conteham programas python devem) e também do *std_msgs* e.g.:

```
catkin_create_pkg pari_aula8_ex3 std_msgs rospy
```

Depois, adapte o exemplo de publicação / subscrição de modo a que os dois programas possibilitem, com a inserção de argumentos pela linha de comandos, definir o nome to tópico em que irão escrever / ler. No caso do programa *publisher.py*, este deve ainda permitir pelo mesmo mecanismo alterar o conteúdo da mensagem que envia periódicamente bem como a frequência de envio.

Usando as novas funcionalidades implementadas. experimente lançar uma constelação de nós para testar a flaxibilidade do sistema de comunicações do ROS. Por exemplo, lance um publicador do tópico "conversations" e dois subscritores a este tópico. Depois lance um outro publicador do tópico "chat" e apenas um subscritor.

Se ainda não o fez é altamente recomendável que instale e configure o terminator (ou similar, e.g., tmux) de modo a gerir mais facilmente a grande quantidade de programas a lançar.

Utilize o rqt_graph para visualizar em tempo real o grafo de computação do sistema criado.

Veja vídeo com um exemplo.

Exercício 4 - Serialização e deserialização em ROS

Apoiando-se no tutorial para criaçao de mensagens em ROS, faça a extensão do exercício 3 de modo a que a informação enviada seja a mesma que no exercício 2 (class dog).

Uma vez que a class dog não é standard, deverá criar uma mensagem custom *Dog.msg* que contenha os mesmos campos da classe do exercício 2.

Crie um novo pacote ROS com o nome *pari_aula8_ex4*