MKI59: Robotlab practical

Practical 6: Concurrent programming

December 21, 2017

Often in robotics programming you want your software to control or observe multiple things simultaneously. This may require that these parts of your software run in parallel: Either on different machines (like a kinect-gesture server or the wit.ai server) or on the same machine but executed on multiple CPUs or cores. In this lecture, we go through different conceptual scenarios for concurrent software with the Nao. We will use the following terminology:

- **host computer:** the computer that is running the main program controlling the Nao, mostly your own computer.
- **Nao computer:** the computer inside the Nao which is running Naoqi.
- **external computer:** a computer running a service which your software depends on (e.g., the Nao computer is an external computer and so are the wit.ai and kinect servers).

1 Scenario 1: Offloading to the Nao computer

You have been using this setup since Lecture 1 in one of two ways. The first is where you subscribe to an event, and Naoqi running on the Nao will start some code to detect the particular event. Once the event triggers, some information is sent back to the host computer which can then respond accordingly. While waiting for this event, you can still perform computations on the host computer. The second way is by using a non-blocking call to Naoqi. For example, the function setAngles in ALMotion is offloaded to the Nao computer, where it is exectued. While a non-blocking function is being executed on the Nao computer, you can still perform computations on the host computer.

2 Scenario 2: Using an external computer

Similarly to Scenario 1, you can request certain functions to run on other computers. Still, unless the function is non-blocking, the code running on the host computer will wait until the remote function returns a value. This means that it is impossible, for example, to record data from joint movements while simultaneously moving the Nao based on objects detected using the camera. Of course, one way this can be done is to have three different computers running three different programs that, if necessary, communicate with each other. A much nicer solution is to use concurrent programming to run multiple threads on the same machine simultaneously.

3 Scenario 3: Concurrent programming

Unfortunately, Python does not have good threading support. Although you can create and execute multiple threads, the particular implementation of threads (using a global interpreter lock) limits execution to one thread at the same time. This is regardless of how many CPUs you have. The biggest benefit Python multi-threading gives is hence not in computation time (or parallel execution), but in smartly interleaving pieces of code when they are locked IO (e.g., when reading/writing from disk). Compared to other languages, like Java, that do support simultaneous execution of multiple threads by exploiting parallel cores, we have to use a different strategy in Python.

3.1 Multi-processing in Python

The only way to do true parallel execution in Python is to start multiple processes from a main Python process. This approach produces quite some overhead, as each process has its own memory and start-up costs. However, it is surprisingly simple and safe to use, precisely because it is constrained to its own memory.

To start a new process from within Python, you need to import `multiprocessing` and `Queue`. Then, any function that you have defined can be started as a separate proces. See the following example:

Code snippet 1: Initializing starting and stopping a process

```
proc1=multiprocessing.Process(name='process-name',
                              target=function_x,args=(arg1,arg2,))

# start the process
proc1.start()
# wait for a while
time.sleep(20)
# stop the process
proc1.join()
```

Implement your own function_x. What happens if you make this loop forever? If you want to kill the child process, use terminate(). To better understand what join() does, follow this link.

This basic principle will allow you to setup multiple processes and within each one of them control or observe different parts of the Nao. The next thing, of course, will be to send information between the processes. Because we are working with processes that have their own memory, and not with threads that share memory, sharing information is relatively safe. To share information between processes we will use `Queue`. A queue in Python is a first in first out data structure that can be shared between multiple processes. Each process can add and/or remove objects from the queue. Importantly, getting an object from the queue can be blocking or non-blocking. The latter will allow us to continue doing stuff in a process, instead of waiting until the queue has a new object.

The next sample code starts three processes: Two writer processes and one reader. One queue is shared between each writer process and the reader process. The first writer process writes a number to its queue every second. The second writer process writers another number to its queue every three seconds. The reader process tries to read numbers from each queue every 0.2 seconds. Whenever it reads a number, it prints the number and the source. This scheme can be useful when you want to fuse information from different modalities. For example, when you have one process that keeps track of a head position using the camera and another process that continuously analyzes the pitch of the user's voice.

Code snippet 2: Starting three processes: Two writers and one reader that gobbles up the output from the writers. Note: The code stops after 21 seconds and can be stopped by pressing Ctrl-C at which point it terminates all processes. If this doesn't happen correctly you may need to killall Python processes from the terminal.

```
import multiprocessing
import Queue
import time
import signal
import sys

def writer1(queue):
    name=multiprocessing.current_process().name
    print name,'Starting'
    i=100
    while True:
        print name,'sending',i
        queue.put(i)
        i+=1
        time.sleep(1)
    print name, 'Exiting'

def writer2(queue):
    name=multiprocessing.current_process().name
    print name,'Starting'
    i=0
    while True:
        print name,'sending',i
        queue.put(i)
        i+=1
        time.sleep(3)
    print name, 'Exiting'



def reader(queue1,queue2):
    name=multiprocessing.current_process().name
    time_out=0.1
    print name, 'Starting'
    msg1=None
    msg2=None
    while True:
        try:
            # get msg from queue. if no msg is available block for time_out
            # seconds. if msg is added, handle it. otherwise if time_out
            # is over, throw Queue.Empty exception.
            msg1=queue1.get(True,time_out)
        except Queue.Empty:
            # if no msg has been added to the queue within time_out seconds
            # do this
            print name, 'no message in queue1'
            try:
                msg2=queue2.get(True,time_out)
            except Queue.Empty:
```

```
                print name, 'no message in queue2'
            else:
                print 'from queue2 received',msg2
        else:
            # if a msg has been added, handle it and sleep (non-blocking)
            print 'from queue1 received',msg1
    print name, 'Exiting'

if __name__ == '__main__':
    print 'starting'
    try:
        q1=multiprocessing.Queue()
        q2=multiprocessing.Queue()
        writer1 = multiprocessing.Process(name='writer1-proc',
            target=writer1,args=(q1,))
        writer2 = multiprocessing.Process(name='writer2-proc',
            target=writer2,args=(q2,))
        reader = multiprocessing.Process(name='reader-proc',
            target=reader,args=(q1,q2,))

        writer1.start()
        writer2.start()
        reader.start()

        t=1
        while t<21:
            time.sleep(1)
            t+=1

        writer1.join()
        writer2.join()
        reader.join()

    except KeyboardInterrupt:
        print "Caught KeyboardInterrupt, terminating processes"
        writer1.join()
        writer2.join()
        reader.join()
```

4 Assignment

In this assignment you will take your first steps into parallel programming for the Nao using the host computer. First, run the example code from Snippet 2 and study it well. Play around with the setup, for example, try to add a third writer thread.

Once you are comfortable with multi-processing in Python you can start combining the scheme with the Naoqi framework. Make one process that continuously tracks for blue balls and reports the bal location (via a queue) to a behaviour process. Make another process that continuously listens to the sound volume in the room. This process too reports its information, i.e., the volume, to the behaviour process. The behaviour process reads from both queues and responds differently based on the information at hand. For example, change the colors of the LEDs using ALLeds:

Code snippet 3: Changing the LED colors.

```
try:
    proxy = ALProxy("ALLeds", IP, PORT)
except Exception,e:
    print "Could not create proxy to ALLeds"
    print "Error was: ",e
    sys.exit(1)

# Example showing how to fade the ears group to rgb color
name = 'EarLeds'
red = 1.0       # intensity of red channel
green = 0.0     # intensity of green channel
blue = 0.0      # intensity of blue channel
duration = 0.5
proxy.fadeRGB(name, red, green, blue, duration)
```

- blue ball in sight, low volume: low intensity blue light
- blue ball in sight, high volume: high intensity blue light
- blue ball not in sight, low volume: low intensity red light
- blue ball not in sight, high volume: high intensity red light

Good luck! More info on "multiprocessing" in Python can be found here.