```
#-----------------------------------------------------------------------
# ## Mestrado em Engenharia Eletrônica e Computação - UCPel
# ## Disciplina de Instrumentação Ubíqua
#
# Descrição:    Servidor XML-RPC em Python
# Aluno:        Douglas Adalberto Scheunemann
# Data:         Setembro de 2015
#
# Observações:  A classe SensorThread derivada de threading.Thread
# foi criada para simular a aquisição de dados de sensores.
# A classe Sensors_API encapsula as funções de acesso ao array de
# sensores instanciados no exemplo que estarão disponíveis através.
# da API XML-RPC.
#-----------------------------------------------------------------------

from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler
from array import array
import random
import math
import threading
import time

str_erro_ok = 'ok'
str_erro_name = 'Sensor não encontrado'


# Caminho de acesso
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Criação do servidor
server = SimpleXMLRPCServer(("localhost", 8000),
                            requestHandler=RequestHandler)

#-----------------------------------------------------------------------
# Classe base para gerenciamento de sensores
#-----------------------------------------------------------------------
class SensorThread(threading.Thread):
    __amostras = None; # array de amostras
    __tam_buffer = None # tamanho do buffer de amostras
    __lock_data = None # semáforo para acesso aos dados
    __periode_s = None # período de amostragem
    __ev_amostrar = None # evento para sincronização de start
#   e stop da amostragem

    # Construtor da classe
    def __init__(self, name, tam_buffer):
        threading.Thread.__init__(self) # construtor da classe base
        self.name = name
        # Inicialização da instância
        self.__amostras = []
        self.__lock_data = threading.Lock()
        self.__tam_buffer = tam_buffer
        self.__ev_amostrar = threading.Event()
        # Libera a execução da thread
        threading.Thread.start(self)
```

```
    # função executada após chamada do método start
    def run(self):
        while(1):
            self.__ev_amostrar.wait() # aguarda evento de amostragem
            self.__lock_data.acquire()
            if(len(self.__amostras) >= self.__tam_buffer):
                self.__amostras.pop(0)
                # remove o dado mais antigo do buffer
            self.__amostras.append(random.random())
            # número randômico de 0 a 1
            self.__lock_data.release()
            time.sleep(self.__periode_s)

    # Inicia execução da amostragem do sensor
    def start(self, periode_s):
        self.__periode_s = periode_s
        self.__amostras = []
        self.__ev_amostrar.set()

    # Interrompe a amostragem do sensor
    def stop(self): # overload
        self.__ev_amostrar.clear() # termina laço da thread

    # Retorna o vetor de amostras do sensor
    def values(self):
        self.__lock_data.acquire()
        amostras_ret = self.__amostras
        self.__lock_data.release()
        return amostras_ret

    # Retorna o valor médio do buffer de amostras do sensor
    def media(self):
        self.__lock_data.acquire()
        acc = 0
        for v in self.__amostras:
            acc += v
        media_ret = acc/len(self.__amostras)
        self.__lock_data.release()
        return media_ret

    # Retorna número de amostras contidos no buffer do sensor
    def count_amostras(self):
        return len(self.__amostras)

#-----------------------------------------------------------------------
# Classe para declaração da API do servidor
#-----------------------------------------------------------------------
class Sensors_API():
    sensors = [] # Array de sensores acessíveis através da API

    # Procura um sensor
    def find_sensor(self, name):
        for s in self.sensors:
            if (s.name == name):
                return s
        return None
```

```
    def start(self, name, periode_s):
        s = self.find_sensor(name)
        if s == None:
            return str_erro_name
        else:
            s.start(periode_s)
            return str_erro_ok

    def stop(self, name):
        s = self.find_sensor(name)
        if s == None:
            return str_erro_name
        else:
            s.stop()
            return str_erro_ok

    def values(self, name):
        s = self.find_sensor(name)
        if s == None:
            return str_erro_name
        else:
           return s.values()

    def media(self, name):
        s = self.find_sensor(name)
        if s == None:
            return str_erro_name
        else:
            return s.media()

    def count_amostras(self, name):
        s = self.find_sensor(name)
        if s == None:
            return str_erro_name
        else:
            return s.count_amostras()

#---------------------------------------------------------------------
# Alocação de objetos para teste
#---------------------------------------------------------------------
s_api = Sensors_API()
s_api.sensors.append(SensorThread("T1", 100))
s_api.sensors.append(SensorThread("T2", 50))
s_api.sensors.append(SensorThread("U1", 30))

#---------------------------------------------------------------------
# Registro das funções acessíveis na API
#---------------------------------------------------------------------
server.register_instance(s_api,  allow_dotted_names=True)
# allow_dotted_names possui restrições e segurança
server.register_introspection_functions()
# system.listMethods, system.methodHelp and system.methodSignature.

# Roda o servidor em laço eterno
server.serve_forever()
```