Messaging/interfaces/connectivity

There are several ways messages or data can be exchanged/transmitted/received in and around CoppeliaSim, but also between CoppeliaSim and an external application, other computer, machine, etc.:

One can send/receive data via:

- signals
- custom data
- calling plugin functions
- calling script functions
- Remote API
- broadcasting a message
- events
- ROS
- ZeroMQ
- WebSocket
- serial port
- sockets
- other

Calling plugin functions

Scripts can call specific plugin functions, so-called callback functions: in order to be able to do this, the plugin must first register its callback functions via simRegisterScriptFunction. This is a convenient mechanism to extend CoppeliaSim's functionality, but can also be used for complex data exchange between scripts and plugins. Following illustrates a very simple plugin function and its registration:

```
void myCallbackFunc(SScriptCallBack* p)
{
    int stack = p -> stackID;
    CStackArray inArguments;
    inArguments.buildFromStack(stack);

    if ( (inArguments.getSize() > 0) && inArguments.isString(0) )
    {
        std::string tmp("we received a string: ");
        tmp += inArguments.getString(0);
        simAddLog("ABC", sim_verbosity_msgs,tmp.c_str());

        CStackArray outArguments;
        outArguments.pushString("Hello to you too!");
        outArguments.buildOntoStack(stack);
    }
    else
        simSetLastError(nullptr, "Not enough arguments or wrong arguments.");
}

// Somewhere in the plugin's initialization code:
simRegisterScriptCallbackFunction("func", nullptr, myCallbackFunc);
```

Calling script functions

A script function can obviously be called from within the same script, but also:

- across scripts (via sim.callScriptFunction or sim.getScriptFunctions)
- from a plugin (via simCallScriptFunctionEx)
- from a ROS client (via a callback mechanism)
- or from a remote API client

The called script function can perform various tasks, then send back data to the caller. This is also a simple way to extend the functionality of an external application in a quick manner. It is however important that the called script doesn't perform lengthly tasks, otherwise everything will come to a halt (lengthly tasks should rather be triggered externally, and processed at an appropriate moment by the script itself when called from the regular system callbacks).

Broadcasting messages

A script or a remote API client can broadcast a message to all scripts at once, via the sim.broadcastMsg function. For instance, following will constantly broadcast a message to all scripts:

Python Lua

```
#python

def sysCall_init():
    sim = require('sim')
    self.objectHandle = sim.getObject('.')

def sysCall_sensing():
    message = {'id': 'greetingMessage', 'data': {'msg': 'Hello!'}}
    sim.broadcastMsg(message)

def sysCall_msg(msg, origin):
    if origin != self.objectHandle and msg.id == 'greetingMessage':
        print("Received following message from script {}:".format(origin))
        print(msg['data']['msg'])
```

Events

A script or a plugin can receive notifications for everything that happens in CoppeliaSim itself, e.g. when an object is created, modified, removed, or when an internal state changed. A script can use the

sysCall_event callback function to that end, coupled with some filtering for efficiency's sake. For instance, following script tracks a pose change with any scene object, and a change in scene object selection:

Python Lua

```python
#python
#luaExec eventCallback = true      -- enable event callback

import cbor2 as cbor

def sysCall_init():
    sim = require('sim')
    filters = {}
    filters[sim.handle_sceneobject] = ['pose']
    filters[sim.handle_scene] = ['selectionHandles']
    sim.setEventFilters(filters)

def sysCall_event(events):
    ev = cbor.loads(events)
    print(ev)
```

Most properties generate an event when changed (i.e. modified explicitly or internally by CoppeliaSim). See also the event viewer add-on located at [Modules > Developer tools > Event viewer].

ZMQ

The ZeroMQ library, wrapped inside the ZMQ plugin, offers several API functions related to ZeroMQ messaging. When using Python, one is of course also free to using the pyzmq package for the ZeroMQ functionality. Following illustrates a simple requester:

Python Lua

```python
#python

def sysCall_thread():
    sim = require('sim')
    simZMQ = require('simZMQ') # suppose we do not use the pyzmq package
    print('Connecting to hello world server...')
    self.context = simZMQ.ctx_new()
    self.requester = simZMQ.socket(self.context, simZMQ.REQ)
    simZMQ.connect(self.requester, 'tcp://localhost:5555')

    for request_nbr in range(11):
        print('----------------------------------------')
        data = 'Hello'
        print(f'[requester] Sending "{data}"...')
        simZMQ.send(self.requester, data, 0)
        rc, data = simZMQ.recv(self.requester, 0)
        print(f'[requester] Received "{data}"')

def sysCall_cleanup():
    simZMQ.close(self.requester)
    simZMQ.ctx_term(self.context)
```

And following would be the corresponding responder:

Python Lua

```python
#python

def sysCall_thread():
    sim = require('sim')
    simZMQ = require('simZMQ') # suppose we do not use the pyzmq package
    self.context = simZMQ.ctx_new()
    self.responder = simZMQ.socket(self.context, simZMQ.REP)
    rc = simZMQ.bind(self.responder, 'tcp://*:5555')
    if rc != 0:
        raise Exception('failed bind')

    while True:
        rc, data = simZMQ.recv(self.responder, 0)
        print(f'[responder] Received "{data}"')
        data = 'World'
        print(f'[responder] Sending "{data}"...')
        simZMQ.send(self.responder, data, 0)

def sysCall_cleanup():
    simZMQ.close(self.responder)
    simZMQ.ctx_term(self.context)
```

WebSocket

The WebSocket plugin, offers several API functions allowing to interact with a web browser. When using Python, one is of course also free to using a Python specific WS package. Following is a simple echo server:

Python Lua

```python
#python

def onMessage(server, connection, data):
    simWS.send(server, connection, data)

def sysCall_init():
    simWS = require('simWS') # suppose one is not using any specific Python package related to WS
    server = simWS.start(9000)
    simWS.setMessageHandler(server, 'onMessage')
```

And following is a simple broadcaster:

Python Lua

```python
#python

def onOpen(server, connection):
    if server not in clients:
        clients[server] = {}
    clients[server][connection] = 1

def onClose(server, connection):
    del clients[server][connection]

def broadcast(server, data):
```

```
    for connection in clients.get(server, {}):
        simWS.send(server, connection, data)

def sysCall_init():
    simWS = require('simWS') # suppose one is not using any specific Python package related to WS
    clients = {}
    server = simWS.start(9000)
    simWS.setOpenHandler(server, onOpen)
    simWS.setCloseHandler(server, onClose)
```

Serial port

CoppeliaSim implements several serial port API functions for Lua. With Python, use the Python serial port package.

Sockets

CoppeliaSim ships with the LuaSocket extension library for Lua, while several packages are available for Python. Following illustrates how to fetch a webpage:

Python | Lua

```
#python

import requests
response = requests.get('http://www.google.com')
page = response.text
```

Other

Many other means of communication can be directly supported from within a script, via a Lua extension library or via a Python package. Indirectly, by passing via a plugin, there are even more possibilities, since a plugin can virtually link to any type of c/c++ communication library.