OSI PI Server Reading and Writing script

Combination of C++ and Pyhton.

Python sends request to the C++ Code, which in turn accesses the PiServer.

# Reading: HOW TO RUN:

1)Open cmdline in read folder

2) run

OSI\_read\_tag.exe <network name of PI Server computer>

3) run

read\_simple.py

# Writing: HOW TO RUN:

1)Open cmdline in write folder

2) run

OSI\_update\_tag.exe <network name of PI Server computer> plc\_source\_ip plc\_dest\_ip plc\_trans\_identify plc\_protocol\_identify plc\_length\_modbus plc\_unit\_identifier plc\_function\_code plc\_reference\_number plc\_bit\_count plc\_byte\_count plc\_payload plc\_query\_or\_response plc\_timestamp plc\_packetnumber

3) run

PI\_save\_modbus2.py

# Installing Requirements:

C++ is Windows ONLY, python can be run by any machine that can run python and pcapy

C++ 2011 compiler (or newer) (for editing the sourcecode)

Visual Studio 2013 requires SDL to be disabled: (go to properties settings ->C++ -> General ->SDL checks NO)

Python 2.7.x

Pcapy for python 2.7 (pcapy-0.10.5.win32-py2.7-winpcap4.1.2.exe)

Winpcap 4.1.2 (comes with wireshark)

PISKD (included with this documentation)

PITrusts created for the machine running c++

# What is happening under the hood (fast version), detailed below:

## Reading Python:

1. Connects to c++ at the given IP:Port
2. Send read Tag cmd to c++
3. Receive answer
4. Ack answer (in case of multiple request goto 3.)
5. Receive ‘req done’
6. Read next Tag/Timestamp see 2.

### Reading C++

1. Open Socket for python
2. Connect to PI-Server
3. Wait for requests from Python
4. Query the PiServer for the specified Tag/Timestamp python send.
5. Check if Server returned the right amount of Tags
6. Check if timestamp is ok (Problem: query function does not care about sub second timestamps) =>Check if requested time is smaller or equal than the actual time in the tag.
   1. Convert times to seconds after epoch (Computer Time, seconds since 1.1.1970) and compare full seconds
   2. Check fractional seconds
   3. In case we have to throw away values because they are too early, goto 4. And request more items accordingly.
7. Send good values and timestamps to python.
   1. Each item is acked to make sure that they arrived successfully. It also makes sure that send requests are not joined together, but split nicely (Otherwise the OS can combine send requests together, mixing up the order).
8. Send “req done” to python.
9. Goto 4. For next query

### Writing Python:

1. Connect to CPP
2. Tap the Network adapter or recorded file
3. Unpack Packet into its parts, only correctly formed Ethernet/IP Packets are accepted
4. Filter for Modbus packets (Port filter on 502)
5. Classify for query or response (by checking if source or destination port is 502)
6. Write plc\_source\_ip, plc\_dest\_ip,plc\_trans\_identify ,plc\_protocol\_identify ,plc\_length\_modbus , plc\_unit\_identifier, plc\_function\_code
7. Depending on Function code and query or response decode the remaining datastream: plc\_reference\_number, plc\_bit\_count, plc\_byte\_count ,plc\_payload, plc\_query\_or\_response
   1. Non existing items are written as a “-1” to the C++
   2. Decoding done right now only for function code 4&15, all other function codes store all remaining information in plc\_payload and leave all other fields as “-1”
8. Goto 3. for next packet

### Writing C++: (Note Requires C++ 2011 compiler)

1. Create Socket for python
2. Connect to Pi-Server
3. Wait for data from python
   1. Each tag received is acknowledged.
   2. Note the information which tag to write is given by the order of the arguments (the script just loops through)
   3. Each Tag of is given a microsecond accurate timestamp, the same for each item of a packet.
4. If packet is done goto 3.

# What is happening under the hood (detailed):

## Reading Python:

1. Connects to c++ at the given IP:Port
   1. Variable for socket “s”
2. Send read Tag cmd to c++
   1. Example s.send('tagname number\_of\_items\_requested 22-Sep-2014 11:51:41:882795')
      1. The space between each item is used by c++ to spate the cmds
3. Receive answer
   1. for x in range(0,number\_of\_items\_requested):
   2. plc\_source\_ip = s.recv(BUFFER\_SIZE)
   3. print plc\_source\_ip
   4. s.send('ack')
   5. timestamp = s.recv(BUFFER\_SIZE)
   6. print timestamp
   7. s.send('ack')
4. Receive ‘req done’
   1. ack = s.recv(BUFFER\_SIZE)
5. Read next Tag/Timestamp goto 2.

### Reading C++

1. Open Socket for python
2. Connect to PI-Server
3. Wait for requests from Python
   1. iResult = recv(ClientSocket, recvbuf, recvbuflen, 0) //and store them in recvbuf
4. Query the PiServer for the specified Tag/Timestamp python send.
   1. piValues = piData->RecordedValuesByCount(requested\_time.c\_str(), requested\_number\_of\_items, dForward, btInside, "", fvRemoveFiltered, NULL);
   2. Going forward in time, do not interpolate values, do not filter values
5. Check if Server returned the right amount of Tags
6. Check if timestamp is ok (Problem: query function does not care about sub second timestamps) =>Check if requested time is smaller or equal than the actual time in the tag.
   1. Convert times to seconds after epoch (Computer Time, seconds since 1.1.1970) and compare full seconds
      1. Variables: timevalue >= requested\_time\_utc?
   2. Check fractional seconds
      1. fractional\_seconds>= requested\_fractional\_seconds?
   3. In case we have to throw away values because they are too early, goto 4. And request more items accordingly.
      1. Variables how\_many\_items\_to\_skip, check if at second run of algorithm how\_many\_items\_to\_skip did not change by comparing it with old\_ how\_many\_items\_to\_skip. If it did not change, break the while loop.
7. Send good values and timestamps to python.
   1. Get each Item
      1. piValue1 = piValues->Item[k]; vValue1 = piValue1->Value; MyPIValue mPV4(piValue1);
   2. Send to Values to python
      1. char \*arg2 = mPV4.bstrValue;
      2. iResult = send(ClientSocket, arg2, (int)strlen(arg2), 0);
   3. Send Timestamp to python
      1. Extract fractional part from: piValue1->TimeStamp->get\_UTCSeconds(timevalue);
      2. Get date with seconds from: char \*arg1 = mPV4.bstrTimeStamp;
      3. Send the combination to python
   4. Each item is acked to make sure that they arrived successfully. It also makes sure that send requests are not joined together, but split nicely (Otherwise the OS can combine send requests together, mixing up the order).
8. Send “req done” to python.
9. Goto 4. For next query

### Writing Python:

1. Connect to CPP
   * 1. Variable “s”
2. Tap the Network adapter or recorded file
   * 1. Variable “cap”
3. Unpack Packet into its parts, only correctly formed Ethernet/IP Packets are accepted
   * 1. Check for Ethernet function code 8 and IP function code 6. All other packets are ignored.
4. Filter for Modbus packets (Port filter on 502)
5. Classify for query or response (by checking if source or destination port is 502)
6. Write plc\_source\_ip, plc\_dest\_ip,plc\_trans\_identify ,plc\_protocol\_identify ,plc\_length\_modbus , plc\_unit\_identifier, plc\_function\_code
7. Depending on Function code and query or response decode the remaining datastream: plc\_reference\_number, plc\_bit\_count, plc\_byte\_count ,plc\_payload, plc\_query\_or\_response
   1. Non existing items are written as a “-1” to the C++
   2. Decoding done right now only for function code 4&15, all other function codes store all remaining information in plc\_payload and leave all other fields as “-1”
8. Goto 3. for next packet

### Writing C++: (Note Requires C++ 2011 compiler)

1. Create Socket for python
   * 1. ClientSocket = accept(ListenSocket, NULL, NULL);
2. Connect to Pi-Server
   * 1. spPISDK.CreateInstance(\_\_uuidof(PISDK));
3. Wait for data from python
   * 1. iResult = recv(ClientSocket, recvbuf, recvbuflen, 0); recvbuf[iResult] = NULL; //marks the end of the received string so that it is recognized as such by cpp
   1. Each tag received is acknowledged.
   2. Note the information which tag to write is given by the order of the arguments (the script just loops through)
      1. Variable “I”
   3. Each Tag of is given a microsecond accurate timestamp , the same for each item of a packet.
      1. std::chrono::system\_clock::now();//Using the c++ library chrono , requireing c++ 2011
      2. spPIValues->Add(timestamp.c\_str(), recvbuf, spNVValAttr);
4. If packet is done goto 3.