ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WEPHA151

A VERY LIGHTWEIGHT PROCESS VARIABLE SERVER*

A. Sukhanov†, J. P. Jamilkowski, Brookhaven National Laboratory, Upton, USA

Abstract

The liteServer is very lightweight, low latency, cross-platform network protocol for signal monitoring and control. It provides very basic functionality of popular channel access protocols like CA or pvAccess of EPICS. It supports request-reply patterns: 'info', 'get' and 'set' requests and publish-subscribe pattern: 'monitor' request. The main scope of the liteServer is: 1) control and monitoring of instruments supplied with proprietary software, 2) seamless connectivity to existing control systems, 3) restricted access to process variables (PVs), 4) possibility to implement the server in FPGA without CPU core. The transport protocol is connection-less (UDP) and data serialization format is Universal Binary JSON (UBJSON). The UBJSON provides complete compatibility with the JSON specification, it is very efficient and fast.

INTRODUCTION

Modern instruments are often supplied with rich proprietary software tools, which make it difficult to integrate them to existing control systems. The initial motivation of developing the liteServer was to build a simple server, which controls a device using supplied proprietary software tools (DLLs or shared libraries) and expose the process variables for access from an existing control architecture. The basic features of the communication protocol are suitable for implementation in FPGA fabric which lacks a CPU core, such firmware could be made fault tolerant using TMR [1] technique.

UBJSON

The UBJSON serialization format have been chosen for communication because it provides complete compatibility with the omnipresent JSON – there is a 1:1 mapping between standard JSON specification and UBJSON [2]. In addition, it has following advantages:

- Ease of implementation. It is feasible to implement it in FPGA fabric.
- Ease of use.
- Speed and efficiency – UBJSON uses data representations that are (roughly) 30% smaller than their compacted JSON counterparts and are optimized for fast parsing. Streamed serialization is supported, meaning that the transfer of UBJSON over a network connection can start sending data before the final size of the data is known.

The Universal Binary JSON specification utilizes a single construct with two optional segments (length and data) for all types:

```
[type, 1-byte char]([integer length])([data])
```

Similarly to JSON, UBJSON defines two container types: **array** (analog of a Python list) and **object** (analog of a Python dictionary).

MESSAGING

The message, received by the server is an UBJSON object with obligatory keys **'cmd'**, **'username'** and **'program'.** The 'cmd' is 2-element UBJSON array, the first element is a liteServer command, the second is an array of arguments. The liteServer command may be one of the following: **'info'**, **'get'**, **'set'**, **'monitor'** and **'retransmit'**. The 'retransmit' command is special. It is used to recover data lost during the transaction. The array of arguments comprises up to four other arrays: device names, parameter names, parameter properties and values. All elements of an array can be requested by specifying an empty array. Example of the command to change frequencies of two devices:

```
{{'cmd': ['set', [['dev1', 'dev2'],
['frequency'], ['value'], [1.0, 2.0]]],
'username': 'JohnDoe', 'program':
'liteAccess.py'}
```

The response message is an object with keys representing device:PV and values representing the requested properties as shown below:

```
{'dev1:frequency': {'value': [1.0]},
'dev2:frequency': {'value': [2.0]}}
```

The **'username'** and **'program'** keys are used for optional access restriction.

THE SERVER

The server listens for request and sends responds through an UDP sockets. Messages, which are too big to fit into a single UDP transfer are chopped into smaller chunks. The position of the chunk is transferred as a first word of the socket message. The client acknowledges the server when the whole message has been received. If a chunk has been lost in the transfer, then the client issues a 'retransmit' request.

In the Python implementation, the liteServer module provides base classes Device and PV for devices and its Process Variables objects to be handled by the server. The

* Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy.
† sukhanov@bnl.gov

PV constructor accepts the following basic keyword arguments:

```
features = 'RWA' # feature,
      # Readable, Writable, Archivable
desc = ''    # description
opLimits = None # operational limits
value = [] # value, should be iterable
```

The type of the value is derived from the type of the provided argument, the size of the value is dynamic. More specific properties could be added using an add_prop() member function. An example below shows a fully-functional server which hosts five devices. Each device consist of ten individually-incremented counters and a modifiable color image.

```
import time,threading
import numpy as np
from liteServer import Device, PV, EventExit

class Scaler(Device):
  def __init__(self,name):
    h,w,p = 1200,1600,3
    img = np.zeros(h*w*p).astype('uint8')\
        .reshape(h,w,p)
    pvs = {
      'counters':   PV('R','10 counters',10*[0.])
      'increments': PV('RW','Increments',range(10))
      'frequency':  PV('RW','Update frequency',
                    ,opLimits=(0,10)),
      'pause': PV('RW','Pause all',[False]),
      'image':    PV('R','Image',[img])
    }
    threading.Thread(target=self._state_machine)\
        .start()

  def _state_machine(self):
    self._cycle = 0
    while not EventExit.is_set():
      EventExit.wait(1./self.frequency.value[0])
      if self.pause.value[0]:
        continue
      # increment counters individually
      increments =  self.increments
      for i,inc in enumerate(increments.value):
        self.counters.value[i] += incr
        self.counters.timestamp = [time.time()]
      # increment all pixels in the image
      self.image.value[0] = (self.image.value[0]\
        + 1).astype('uint8')

# Instantiate five devices
devices = [Scaler('dev'+str(i+1)) for i in range(5)]

# Instantiate server and enter an event loop
server = liteServer.Server(devices)
server.loop()
```

THE CLIENT

A Python module liteAccess provides access to process variables, hosted by a liteServer. It defines a PV class for accessing multiple process variables from the server. For example, an access object for two process variables ‘par1’ and ‘par2’ from two devices ‘dev1’ and ‘dev2’ would be:

```
manyPVs = PV(('host;port',
('dev1','dev2'),('par1','par2')))
```

where the host and port are the host name and port number of the server.

The access object for all parameters of ‘dev1’ would be:

```
  aPVs = PV(('host;port',('dev1')))
```

The value of a process variable is Python decorator, the following statement:

```
list_of_values = manyPVs.value
```

executes a ‘get’ transaction with the server and decodes response to a list of values. Similarly, the statement:

```
manyPVs.value = list_of_values
```

encodes a list of values to UBJSON format and executes a ‘set’ transaction with the server.

A PV.info() method retrieves information of requested process variables. Example of PV.info() and getting value of a process variable:

```
from liteAccess import PV
aPV = PV(('localhost;9700',('dev1'),('frequency')))
print(aPV.info())
{'dev1:frequency': {'value': '?',
  'count': [1],
  'features': 'RW',
  'desc': 'Update frequency',
  'opLimits': [0, 10]}}
print(aPV.value)
{'dev1:frequency': {'value': [1.0]}}
```

BRIDGE

A bridge program is used to integrate any liteServer-based device into existing control system. In RHIC Control Architecture the bridge is a simple ADO (Accelerator Device Object) manager, which automatically creates ADO parameters and translates ADO requests get(), set() and getAsync() into corresponding liteServer requests: **get**(), **set**() and **monitor**().

GUI CLIENT

A spreadsheet-based GUI client is provided to allow for simple control and monitoring of process variables. It automatically represents the PVs as corresponding GUI elements like TextEdits, SpinBoxes, and CheckBoxes.

CONCLUSION

Several devices at RHIC are served by the Python-based liteServers, hosted on Windows and Raspberry Pi platforms:

- Magnetometers
- Laser interferometers
- Infrared cameras.

Servers are connected to the RHIC Control Architecture using a bridge ADO manager program. Transfer of several-megabyte data samples at 50 MB/s has been demonstrated. The typical time of a short get/set transaction is ~2 milliseconds.

Size of the liteServer and liteAccess modules is ~350 lines of code each, size of the bridge program for RHIC Controls: 200 lines.

Planned Improvement

- Implement liteServer in FPGA.
- Improve support for the 'monitor' request.
- Add optional TCP protocol (for large transfers).
- Improve multi-client performance.

- Provide a plotting application (the image processing is already supported by an imageViewer).
- Develop a bridge for EPICS system.

REFERENCES

[1] TMR, https://en.wikipedia.org/wiki/Triple_modular_redundancy

[2] Universal Binary JSON, http://ubjson.org/