Data Distribution Service for Python Applications

Nanbor Wang and Svetlana Shasharina
Tech-X Corporation
www.txcorp.com
Project funded by DOE Grant: DE-SC0000842 and Tech-X Corporation
Introduction and Goal

Why Python:
- Python is a popular language due to its robust dynamic scripting language features and runtime supports
  - Rapid prototyping
  - Web scripting, XML processing,
  - GUI/database applications
  - Steering scientific applications
- Many complex applications are based on Python with high-performance cores implemented using other technologies
- Currently, no standardized DDS Python mapping available and for developers to use DDS in their Python applications
- Hence the project goal is to
  - Implement a Python bridge for DDS to allow Python applications to participate in DDS data exchanges
  - Allow Python developers to interact with DDS data spaces directly
  - Eliminate the need to generate Python wrappers for topic-specific C/C++ mapping codes
Typical Approach Puts the Bridge Above the Topic/Application level

Python Applications

Data Publishing Logic
- TopicWriter
- Writer
- Publisher

Data Subscribing Logic
- TopicReader
- Reader
- Subscriber

Domain Participant

Network
Typical Approach Implies Regeneration of Glue for Each New Topic

- *Use vendor provided tools such as “idlpp”, to generate the C/C++ mappings from type definitions*
- Use some wrapper tools such as Boost.Python or SWIG, to wrap C/C++ code into bridges
  - Type/application-specific interfaces
  - General DDS API
- Applications use these generated python classes to pub/sub
- Issues:
  - Wrappings are type/application-specific
  - Requires extra steps outside of Python to generate the bridge objects
  - When topic definition changes, need to re-generate all the bridging objects and the application codes
  - *Disruptive to Python’s dynamic/interpretive language features*
- Solutions: A generic Python DDS bridge implementation
Tech-X Implemented an Architecture for a Generic Python DDS Bridge

Python Applications

TopicWriter

Type Metadata

generates

TopicReader

pyDDS Services

Writer

Publisher

Domain Participant

Network

Reader

Subscriber
Our Approach Allows Dynamic Generation of Python Topic Code

- Import PyDDS as a python module into Python application codes
  - We are aware of another PyDDS implementation from Github: https://github.com/forrestv/pyDDS
- Dynamically generate topic-specific DDS classes using services provided by PyDDS
- Interact with DDS subsystem directly via PyDDS and the generated topic-specific objects
- Benefits:
  - No need to use tools outside of Python
  - No need to re-generate Python bridges when IDL changes
  - More natural Python application development flow
PyDDS Relies on Boost.Python, Python and OpenSplice

Key components fall into 3 main categories:

• General DDS API
  • Managing various entities and built-in data types (DomainParticipant, Publisher/Subscriber, QoS, WaitSet, etc.)

• Type management facility
  • Internalizing type definitions
  • Generating typed objects

• Type specific API
  • Data Readers/Data Writers
  • Listeners*
Current Status

- Implementation based on OpenSplice Community Edition version 5.4.1/5.5.1
  - Linux and Windows platforms
  - With stock Python installation 2.6 or later
  - Boost library is optional for end-users
- Supports for simple topic types using IDL
  - All basic types and strings
- Supports for programmatic construct of topic types
- Simple read/write
- WaitSets/Conditions
- Listener callbacks
- Some support for SimD-like abstractions
# A one-stop interface into the pydds global factory methods
import PyDDS;
pydds=PyDDS.PyDDS()
dp=pydds.create_participant("")  # DomainParticipant Factory
publisher=dp.create_publisher(publisherQos)

# For higher abstraction, PyDDS defines a default dataspace object.
# It can be explicitly instantiated or generated by PyDDS with
# default (nameless, partitionless) dataspace automatically.
# A Dataspace object can instantiate default subscriber/publisher
# objects with default QoS policies automatically on demand.

myDataspace
    = pydds.connect_dataspace
        ("Domain name", "Partition name")
dp=myDataspace.get_participant()
publisher=myDataspace.get_publisher()
# Instantiate QoS objects and manipulating them using
# standard entity calls for QoS policy manipulations
# and some SimD-like calls

publisherQos = pydds.PublisherQos()
publisher.get_default_publisher_qos(publisherQos)
publisherQos.set_partition("Example Partition")

myTopicQoS = pydds.TopicQos()
myTopicQoS.set_reliable()
myTopicQoS.set_transient()
myTopicQoS.set_keep_last(3)
**Topic Type Definition**

**Management Examples**

- **Parsing IDL Files**

  ```python
  # Getting a hold to the Type Manager
  idlFilePath=os.getcwd()+'/HelloWorld.idl'
  ddsTypeManager=pydds.get_type_namager()
  ddsTypeManager.parseIDL(idfFilePath)
  
  # Creating a Type Factory
  typebuilder=ddsTypeManager.DDSTypeFatory(['module','list'],['topic name', sourceURL])
  typebuilder.addPrimitive(DDSTYPES_LONG, 'userID')
  typebuilder.addPrimitive(DDSTYPES_STRING, 'message')
  typebuilder.add_keys(['userID'])
  helloWorldMetaClass = typebuilder.complete_type()
  ```

- **Or, Constructing a type programmatically**

  ```python
  # Getting a hold to the Type Manager
  idlFilePath=os.getcwd()+'/HelloWorld.idl'
  ddsTypeManager=pydds.get_type_namager()
  ddsTypeManager.parseIDL(idfFilePath)
  
  # Creating a Type Factory
  typebuilder=ddsTypeManager.DDSTypeFactory(['module','list'],['topic name', sourceURL])
  typebuilder.add_primitive(DDSTYPES_LONG, 'userID')
  typebuilder.add_primitive(DDSTYPES_STRING, 'message')
  typebuilder.add_keys(['userID'])
  helloWorldMetaClass = typebuilder.complete_type()
  ```

- **Current status**
  - Support all primitive types and strings
  - Need supports for sequences, arrays, nested structs
Using the Topic Type Metaclasses

- Get a hold of a metaclass and inquire about the type information

```python
# acquire the metaclass object from the type manager
helloWorldType=ddsTypeManager.getTypeByName('HelloWorldData::Msg')
topicTypeName=helloWorldType.pydds__getTypename()
keyStr=helloWorldType.pydds__getKeys()  # Comma-separated keys
```

- All type-specific operations use type metaclasses

```python
# Create an object instance of the type
helloWorld = helloWorldType()
helloWorld.userID=1001
helloWorld.message=‘Hello World’
# Create a sequence of HelloWorld Objects
helloWorldSeq=pydds.ObjectSeq(helloWorldType)
helloWorldSeq[0].userID=1002
helloWorldSeq[0].message=‘Hello again!’
```

- Similarly, registering the type definition with DDS

```python
dp.register_type(helloWorldType, topicTypeName)
```
Type-specific Operation Examples: Create Topics, Readers, Writers

# Creating/Finding a topic in the data space
# Last argument specifies the URI of the topic structure
helloTopic = dp.create_topic('HelloWorld_Msg', topicTypeName, HelloWorldTopicQos)

# Creating topic-specific reader/writer:
helloReader = subscriber.create_reader (helloTopic, readerQoS)
helloWriter = publisher.create_writer (helloTopic, writerQoS)
More Type-specific Operation Examples: Simple Writing and Reading DDS Samples

```python
# creating a sample
helloSample=helloWorldType()
helloSample.userID=1001
helloSample.message="Wow!"

# publishing the sample
status = helloWriter (helloSample)

# Simple read is straightforward
sampleSeq=pydds.ObjectSeq(helloWorldType)
infoSeq = pydds.SampleInfoSeq()
status=helloReader.take(sampleSeq,
    infoSeq,
    pydds.LENGTH_UNLIMITED,
    pydds.ANY_SAMPLE_STATE,
    pydds.ANY_VIEW_STATE,
    pydds.ANY_INSTANCE_STATE)
```
Building a WaitSet Example

# creating a WaitSet object
myWaitSet=pydds.WaitSet()

termCond=pydds.GuardCondition()
newMsgCond=helloReader.create_readcondition
    (pydds.NOT_READ_SAMPLE_STATE, pydds.ANY_VIEW_STATE,
     pydds.ALIVE_INSTANCE_STATE)
wrtrCond=helloReader.get_statuscondition()
wrtrCond.set_enabled_statuses(pydds.LIVELINESS_CHANGED_STATUS)
myWaitSet.attach_condition(termCond)
myWaitSet.attach_condition(newMsgCond)
myWaitSet.attach_condition(wrtrCond)

# Waiting and Acting on Waitsets
condList=pydds.ConditionSeq(3)
timeout=pydds.Duration(3,0)
myWaitSet.wait(condList, timeout)
Using A WaitSet Example

```python
# Waiting and Acting on Waitsets
condList=pydds.ConditionSeq(3)
timeout=pydds.Duration(3,0)
myWaitSet.wait(condList, timeout)
for i in range(len(condList)):
    if (termCond.is_same_condition(condList[i])):
        # Handle terminate signal
    elif (newMsgCond.is_same_condition(condList[i])):
        # Handle new sample available
    elif (wrtrCond.is_same_condition(condList[i])):
        # Handle writer joining/leaving
    else:
        # something is wrong....

# Cleaning up
myWaitSet.detach_condition(wrtrCond)
...```
Simple Event-based Listener Examples

- PyDDS provides several “listener objects” for calling Python callback functions

```python
# Getting a hold to a DataReaderListener
exListener = pydds.DataReaderListener()

- Implementing Python callbacks as functions

```python
def data_handler(reader):
    dataSeq = pydds.ObjectSeq(helloWorldType)
    infoSeq = pydds.SampleInfoSeq()
    reader.read(dataSeq, infoSeq, ...) # No need to downcast!!

    ...

exListener.set_on_data_available(data_handler)
# each callback can be set separately
listenerMask = pydds.DATA_AVAILABLE_STATUS | pydds.REQUESTED_DEADLINE_MISSED
helloReader.set_listener(exListener, listenerMask)
```
More Event-based Listener Examples

- Alternative, implement a set of related callbacks as member functions in a class

```python
class appEventLogic:
    def __init__(self, more_args):
        # define and initialize internal states
        def deadline_missed(self, reader, status):
            ...
        def liveliness_changed(self, reader, status):
            ...
        def newdata(self, reader):
            ...

    applicationLogic = appEventLogic(more_args)
exListener.set_on_deadline_missed(lambda r, s:
    applicationLogic.data_handler(r, s))
exListener.set_on_data_available(lambda r:
    applicationLogic.newdata(r))```
Example Applications under Developments

- The flying shape example using Pygame
  - Show interoperability with other DDS implementations
- HPC/multi-physics simulation monitoring/steering applications
  - Tech-X has vast expertise in high-performance, heterogeneous, parallel, multi-physics simulations
  - *Python’s fast prototyping capability often helps* integrating these computational/simulation modules
  - Similarly, Python has shown to help GPU/OpenCL kernel development by handling all the error-prone, architecture dependent configuration code
  - Coupled with DDS, we can provide monitoring and steering capabilities to real-time data processing

For more information, please visit: [http://www.txcorp.com/](http://www.txcorp.com/)
Conclusion and Future Work

• DDS for Python marries two robust and dynamic technologies

• Currently, we support most key DDS features over OpenSplice Community Versions but leverage key Python dynamic language features

• Will be available soon

• Further work includes:
  • Support for most standard API
  • Support for compound data types

Please come to see our talk on scientific DDS applications During OpenSplice’s Users Meeting
Future Work: Enhancing PyDDS API

- Provide Higher-level of abstractions
  - Better dataspace support
  - Configuration can be done outside the code using XML files
  - Waitset abstraction – automatic clean up and traversal dispatch of handlers
  - Single handler class support for Listeners
    - Allow whole-sale replacement of all listener callbacks

- Enhance Python API
  - Borrow more from the new C++ PSM
    - Allow tweaking QoS with a list of policy objects
      ```python
tqos.set([History.keep_last(3), Reliability.Reliable()])
```
  - More Pythonism
    - More use of exceptions
    - E.g., read operations can return a tuple
      ```python
      (dataSeq, infoSeq) = reader.read(max_length, conditions)
      ```