**Virtual Earthquake and seismology Research Community e-science environment in Europe**
**Project 283543 – FP7-INFRASTRUCTURES-2011-2 – www.verce.eu – info@verce.eu**

# The VERCE Science Gateway: Dispel4Py Down to the Basics

(dispel4py training)

15-16 October 2014

# Outline

- What is dispel4py
- What is a stream
- What is a processing element (PE)
- What is a instance
- What is graph
- What I need for constructing a dispel4py workflow
- How to implement PEs
- Types of PEs:
  - GenericPE
  - IterativePE
  - SimplePE
  - Create_iterative_chain
- How to connect PEs
  - Create an object from a PE
  - Create a graph

# What is dispel4py

- Dispel4Py is a library used to describe **abstract workflows for distributed data-intensive applications**
- The language used for describing the data-flow and the processing elements of the workflow is **Python**.
- It is dataflow-oriented rather than control-oriented:
  - No required specification of how data should be produced or consumed

# What is a stream

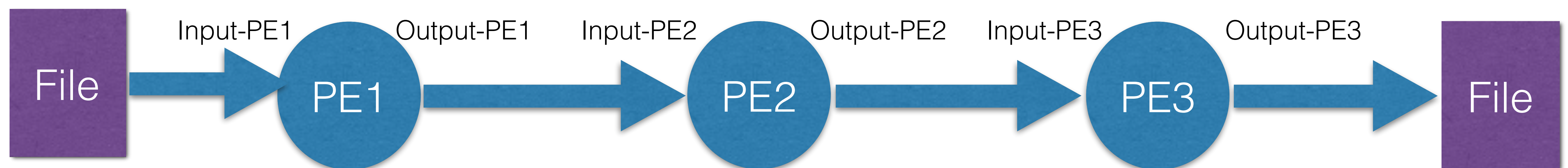
- A stream is a sequence of data elements made available over time
- It also can be defined as a flow of input or output data.
- It delivers data to its destination or takes data from a source.

# What is a processing element (PE)

- PE is a computational activity which encapsulates an algorithm, services and other data transformation processes.

- PEs represent the basic computational blocks of any Dispel4Py workflow.

- Each PE has:
  - inputs & outputs
  - computational activity.
- PEs are connected by streams, and not by writing or reading files



# What is a instance

- It is the executable copy of a PE that runs in a process.
- During execution time, each PE is translated into one or more (MPI & Multiprocess) instances.
- A connection streams data between PE instances

# What is a graph

- It defines the way throughout which PEs are connected and data can be streamed.
- topology of the workflow
- No limitations of type of graphs

# What I need for constructing a dispel4py workflow

- Users **only** have to i**mplement their PEs** (in python) and connect them as they desire in **graph**.:
  - We need to learn how to implement PEs
  - We need to learn how to connect them.

# How to implement PEs

- Each PE must indicate:
  - Inputs & Output streams
  - Computational activity for processing data-blocks —> “process” menthod.

# Types of PEs

| Type | Inputs | Outputs | When to use it |
|---|---|---|---|
| **GenericPE** | *n* inputs | *n* outputs | many inputs and/or many outputs |
| **IterativePE** | 1 input named 'input' | 1 output named 'output' | processing one data block and producing one in each iteration |
| **SimpleFunctionPE** | 1 input named 'input' | 1 output named 'output' | only implement process method |
| **create_iterative_chain** | 1 input named 'input' | 1 output named 'output' | pipeline of functions processing sequentially; creates a composite PE |

# GenericPE example

```
# this PE consumes data in the format [url] - a list with one element
from obspy.core import read
from dispel4py.core import GenericPE
class StreamAndStatsProducer(GenericPE):
    def __init__(self):
        GenericPE.__init__(self)
        self._add_input('input')
        self._add_output('output_stream')
        self._add_output('output_stats')
    def process(self, inputs):
        data = inputs['input']
        filename = data
        st = read(filename)
        # This PE returns two outputs: the output stream and the trace stadistics (metadata).
        return {'output_st': stream, 'output_stats': st[0].stats}
```

This PE reads a file (input) that contains seismological traces and returns two outputs: obspy stream (output_stream) and metadata (output_stats).

What we have learnt:

- We can add several outputs with different names
- The process method gets values from the input streams
- The process method returns both streams

# IterativePE example

```
class StreamProducer(IterativePE):
    def __init__(self):
        IterativePE.__init__(self)
    def _process(self, data):
        # this PE consumes one input
        self.log(data)
        filename = data
        st = read(filename)
        return st
```

This PE also reads a file (‘input’) that contains seismological traces and returns one output (‘output’): obspy stream.

What we have learnt:

- We don’t need to specify the input and output
- The parameter to the _process method is a tuple
- _process returns the value that is written to the output stream

# SimpleFunctionPE

```
def stream_producer(data):
    filename = data
    st = read(filename)
    return st
#For using this function as a PE we need to use 'SimpleFunctionPE' before defining the graph:

streamProducer = SimpleFunctionPE(stream_producer)
```

This function reads a file that contains seismological traces  and returns it as an obspy stream.

What we have learnt:

- Only implement the processing function
- The easiest but the most restrictive way
- The **function cannot store state between calls**; for example you can't implement SUM or AVG with it
- 1 input called 'input', 1 output called 'output'.

# create_iterative_chain

```
def decimate(data, sps):
    st = data[0]
    st.decimate(int(st[0].stats.sampling_rate/sps))
    return st
def detrend(data):
    st = data[0]
    st.detrend('simple')
    return st
def demean(data):
    st = data[0]
    st.detrend('demean')
    return st
# For using this function as a PE we need to use 'creative_iterative_chain' before defining the graph.
preprocess_trace = create_iterative_chain([(decimate, {'sps':4}), detrend, demean])
```

What we have learnt:

- We can create a *composite* PE which processes several function in a sequence
- Creates a pipeline of SimpleFunctionPEs
- It's the easiest way to create a pipeline but the most restrictive
- 1 input called 'input', 1 output called 'output'.

# How to connect PEs: Create an object from a PE

- Create an object of PE WHEN is **GenericPE** or **IterativePE** type:

stream_producer1=StreamAndStatsProducer()

stream_producer2=StreamProducer()

- Create an object of PE WHEN is a **function**:

stream_producer3 = **SimpleFunction**(stream_producer)

** imagine that stream_producer has an input parameter called 'name' —>

def stream_producer(data, name):

……

The object will be:

stream_producer3 = **SimpleFunction**(stream_producer, {'name':'station1'})

- Create an object of PE WHEN is a **compositePE**:

preprocess_trace = **create_iterative_chain**([(decimate, {'sps':4}), detrend, demean])

# How to connect PEs: Create a graph

- **Import WorkflowGraph**

from dispel4py.workflow_graph import WorkflowGraph

- **Create the objects ….**

stream_producer1=StreamAndStatsProducer()

stream_producer3 = SimpleFunctionPE(stream_producer)

preprocess_trace = **create_iterative_chain**([(decimate, {'sps':4}), detrend, demean])

- **Create the graph ….**

graph = WorkflowGraph()

graph.connect (<object_PE_source>, ‘<name_output_stream>’,<object_PE_destine>, ‘<name_input_stream>’ )

graph.connect(stream_producer1, ‘output_stream', preprocess_trace, 'input')

graph.connect(streamProducer3, 'output', preprocess_trace, 'input')