



Dataflow Processing Engine Based on Data Distribution Service

Svetlana Shasharina and **Nanbor Wang**

Tech-X Corporation

[sveta,nanbor}@txcorp.com](mailto:{sveta,nanbor}@txcorp.com)

Feb 13, 2012



Outline

- Background
- Common features of scientific workflows
- Expressing workflows
- DDSFlow workflow engine: design, status and future steps
- Support for Python in DDS



Background

- Collaboration with FNAL on dataflow processing and monitoring for workflows (originally for JDEM and LQCD). FNAL drove the requirements and are shaping the workflow language
 - Erik Gottschalk
 - Jim Kowalkowski
 - Marc Paterno
- Work is funded by Phase II SBIR from HEP/DOE
- Goals:
 - Identify key requirements of targeted workflows
 - Find ways to express them
 - Leverage Data Distribution Service for implementation
 - Develop a working system



Definitions

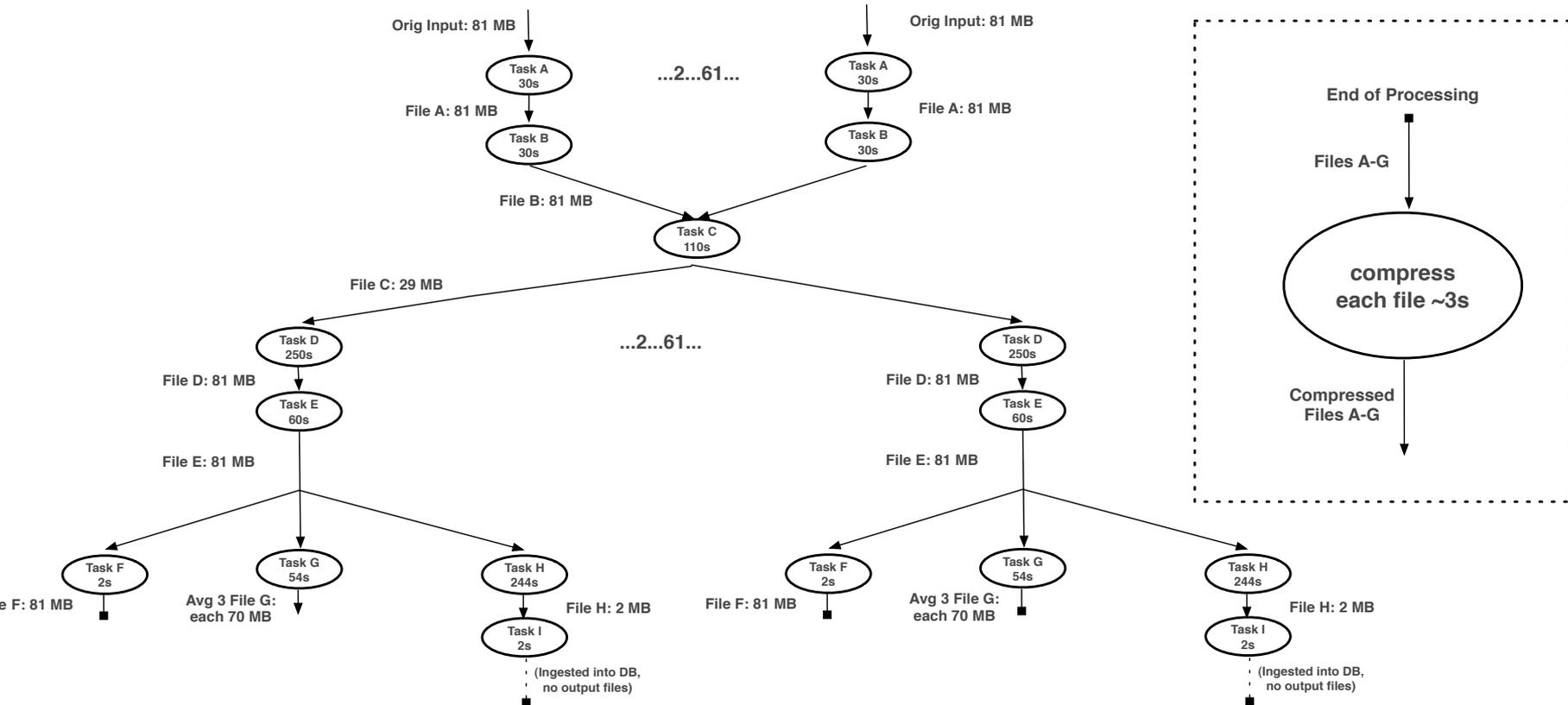
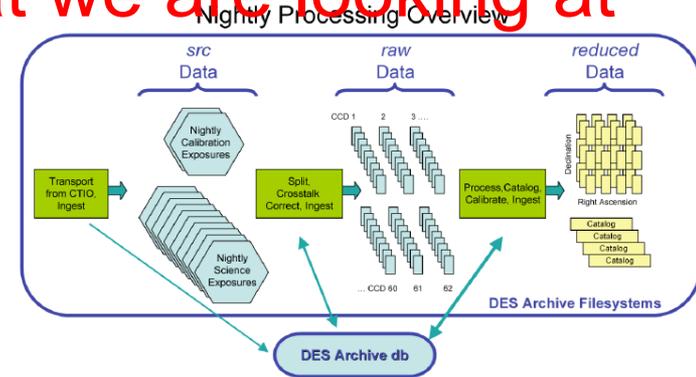
- Actor : a minimal component a workflow (executable taking some input and producing output). Other names: participant and module
- Composite actor: several actors connected via workflow control patterns, that mimics actors
- Workflow: Composition can be specified using the patterns or some other language (for example, a functional language) or graphical representation (a la Kepler). In and outs should be matched. Technically: a composite actor
- Dataflow is a data-driven workflow
- Job is associated with some input data (explicitly specified) and a particular workflow



DES workflow is an example of a data processing workflow that we are looking at

FinalSE Pipeline

All numbers are estimates with some more accurate than others
Codes use less than 2G memory (Most less than 1G)
Extra calibration files not illustrated ~18G total
Misc smaller output files not illustrated
62 pieces almost independent trivially-parallel pipelines (exception task C)





Common features of scientific workflows

- Data driven execution coordination
 - Actions triggered by data availability
 - Capable to handle “stream of data”
- Dynamic
 - Actual flow depends on the result of the previous step
 - Resources allocated as needed dynamically and flow is changed depending on outcome and number of outputs
- Automatic management of transient data
 - Without the need to explicitly specify locations and names
 - Selectively archived
- Expression for workflow should be editable: text-based
 - GUI could be useful to set up a simple prototype or show the flow
 - GUIs do not work well in distributed environments)
 - Standard notations that are reusable but easy to use



Other requirements

- Configurable resource allocation/task scheduling strategies
- Tolerance for partial failures
 - Allow dataflow to continue when less than some % subset of data cannot be processed
- Recoverability from software/hardware failures (from a single dataflow or a set of dataflows)
- Provenance – tracking software packages/versions used in a dataflow processing
- Parameter managements (physics/algorithmic/execution parameters)



There are many types of workflows: how to limit?

- Choosing particular control patterns
- Particular syntax to express the workflow
- Compatibility with existing applications
- Target run-time



Dataflow execution can be defined using common workflow patterns

<http://www.workflowpatterns.com/patterns/control/>

- Sequence
- And-split or parallel split (called map for identical threads)
- And-merge or synchronization (called reduce for identical threads)
- Or-split (exclusive)
- Or-merge (exclusive)
- Condition/loop



Functional language syntax for describing dataflow

- Functional language treat $y = f \bullet g$ as a function definition, not assignment. Thus y is a functor (function that can be treated as an object). Predefined methods (for example, map) allow easy creating of composite actors. FLs allow
 - Unbounded stream instead of one argument so a function gets applied to a list (x , followed by remainder)
 - Lazy evaluation: happens only when the elements are available (a must if you are dealing with streams or in dynamic flows when we do not know the number of outcomes in a step)
- DDSFlow
 - We adopt the syntax from functional languages for describing dataflow processing



Dataflow processing definitions in DDSFlow

Borrow from FLs, we maps functional definitions to dataflow processing definitions

Each function defines an executable that take some input data and produce some output data

$$y=F(x)$$

- Input/output data are typed, we currently assume they are files containing data of specific type in some format
- In DDSFlow, we need to associate variuys parameters with F

A sequence of functions: $y=F(G(H(x)))$ can be written as:

$$y=FGH(x)$$

Map is a special function that hints at possible parallelization:

$$y[]=map(F, x[])$$

Split/fork merge/join will be inferred through temporarily variables



More functions under consideration (FNAL suggestion)

- Map applies a function to every element of a list:

$\text{map } (f, [x_0, x_1, \dots, x_{N-1}]) = [f(x_0), \dots, f(x_{N-1})]$

- Reduce collapses a list into a single value by applying a binary operator * cumulatively from left to right:

$\text{reduce } (*, [x_0, x_1, x_2, x_3]) = ((x_0 * x_1) * x_2) * x_3.$

- Scan accumulates all the intermediate results of reduce operation:

$\text{scan } (*, [x_0, x_1, x_2, \dots]) = [x_0, x_0 * x_1, (x_0 * x_1) * x_2, \dots].$

- Filter selects elements of the list for which a function applied returns true:

$\text{filter } (f, [x_0, x_1, \dots, x_{N-1}]) = [x_i, \dots]$ for all i for which $f(x_i) = \text{true}$.

- Zip merges two lists into one with elements that are tuples from the original lists:

$\text{zip } [a,b,c\dots] [z,y,x\dots] = [(a,z), (b,y), (c,x)\dots]$

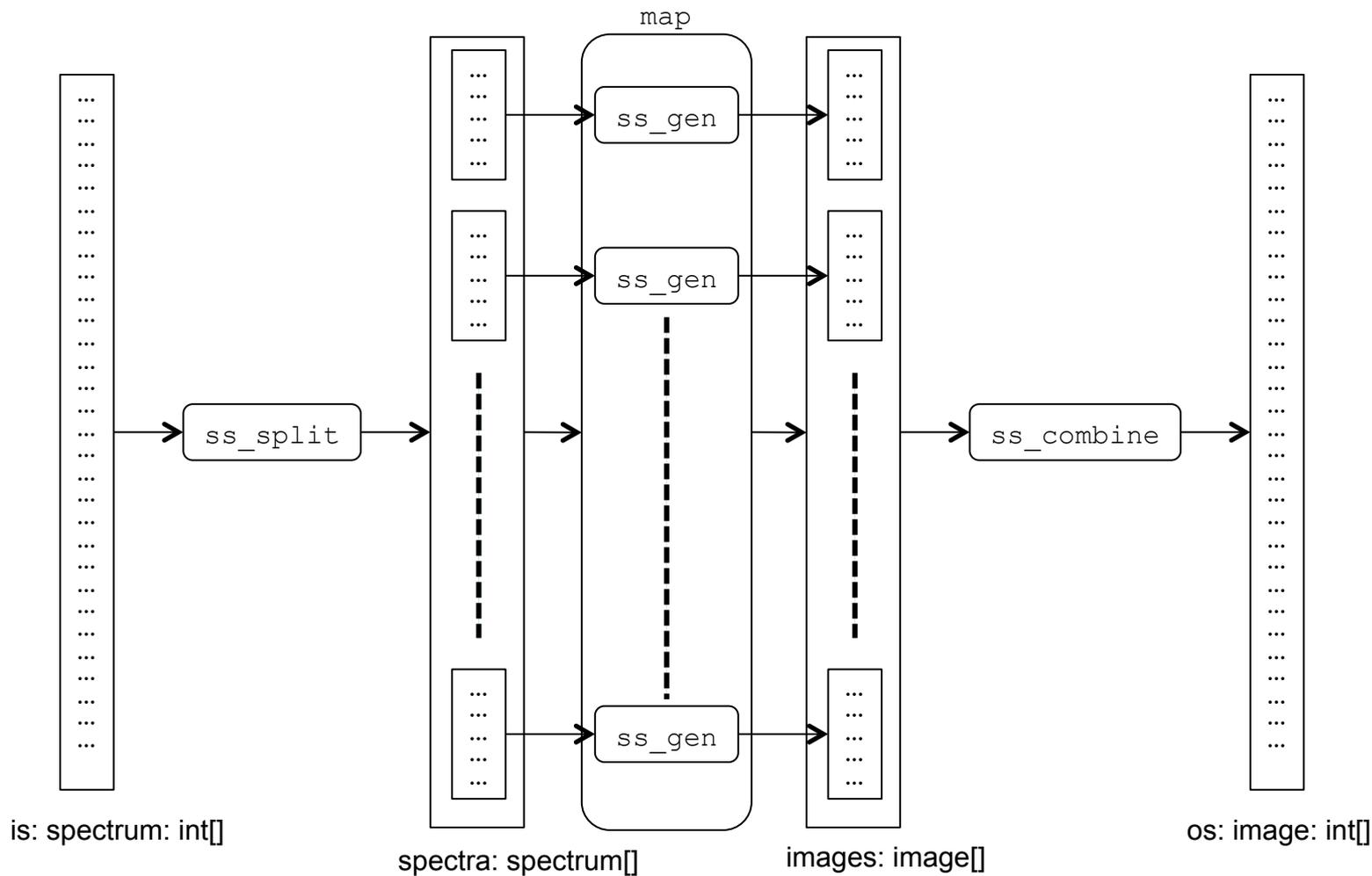
- Unzip takes a list of tuples, and breaks them apart into a tuple of lists

$\text{unzip } [(a,b),(c,d)] = ([a,c], [b,d])$

- Cycle is the same as map but until some condition is true.



A common dataflow processing scenario – parallel pipeline using map/reduce



```
os = ss_combine(map ss_gen (ss_split is))
```



How such a simple parallel map/ reduce dataflow look like

```
ss_split :: Spectrum -> [Spectrum]
```

```
ss_generate :: Spectrum -> Image
```

```
ss_combine :: [Image] -> Image
```

```
spectra = ss_split input_spectrum
```

```
images = map ss_generate spectra
```

```
result = ss_combine images
```

OR

```
my_workflow :: Spectrum -> Image
```

```
my_workflow is = ss_combine (map ss_generate (ss_split is))
```



Functional definitions to DDSFlow execution entity mappings: Simple actors

Internally, DDSFlow uses XML documents to describe dataflow

A simple function is mapped to an actor

- Maps to an external executable
- Has predefined input/output types
- Can be associated with a configuration file
- Work as an insertion point for provenance and Q/A

```
<actor>
  <id>foo</id>
  <exec>foo.sh</exec>
  <inport>
    <type>Type1</type>
  </inport>
  <outport>
    <type>[Type2, Type3]</type>
  </outport>
  <config>foo.cfg</config>
</actor>
```



Composite actors

- Sequential actors
 - Represent a series of actors into one actor
 - Can refer to other composite actors
 - Define a sequential firing rules

```
<compositeActor>  
  <type> sequence </type>  
  <id> denoiser3 </id>  
  <actors> denoiser1, denoiser2 </actors>  
</compositeActor>
```

- Map actors
 - Can refer to a composite actor
 - Define a possible dynamic parallel firing rules
 - Define a synch point at the end

```
<compositeActor>  
  <type>map</type>  
  <id>apply_filter</id>  
  <actor>filter_seq</actor>  
</compositeActor>
```



Forming dataflow processing definitions

- Reduce is like a regular actor

- Types must match

```
<dataflow> // Not sure about this?
```

```
<actorId> fullworkflow </actorId>
```

```
<indata> inputfilenames </indata>
```

```
<outdata> outputfilenames </outdata>
```

```
</dataflow>
```

- Yet to define

- Split defines a firing rule to trigger concurrent actors to fire

- Merge defines a synch point

- Dataflow definition invokes a actor (usually a composite one)

- Types can be inferred from the actors

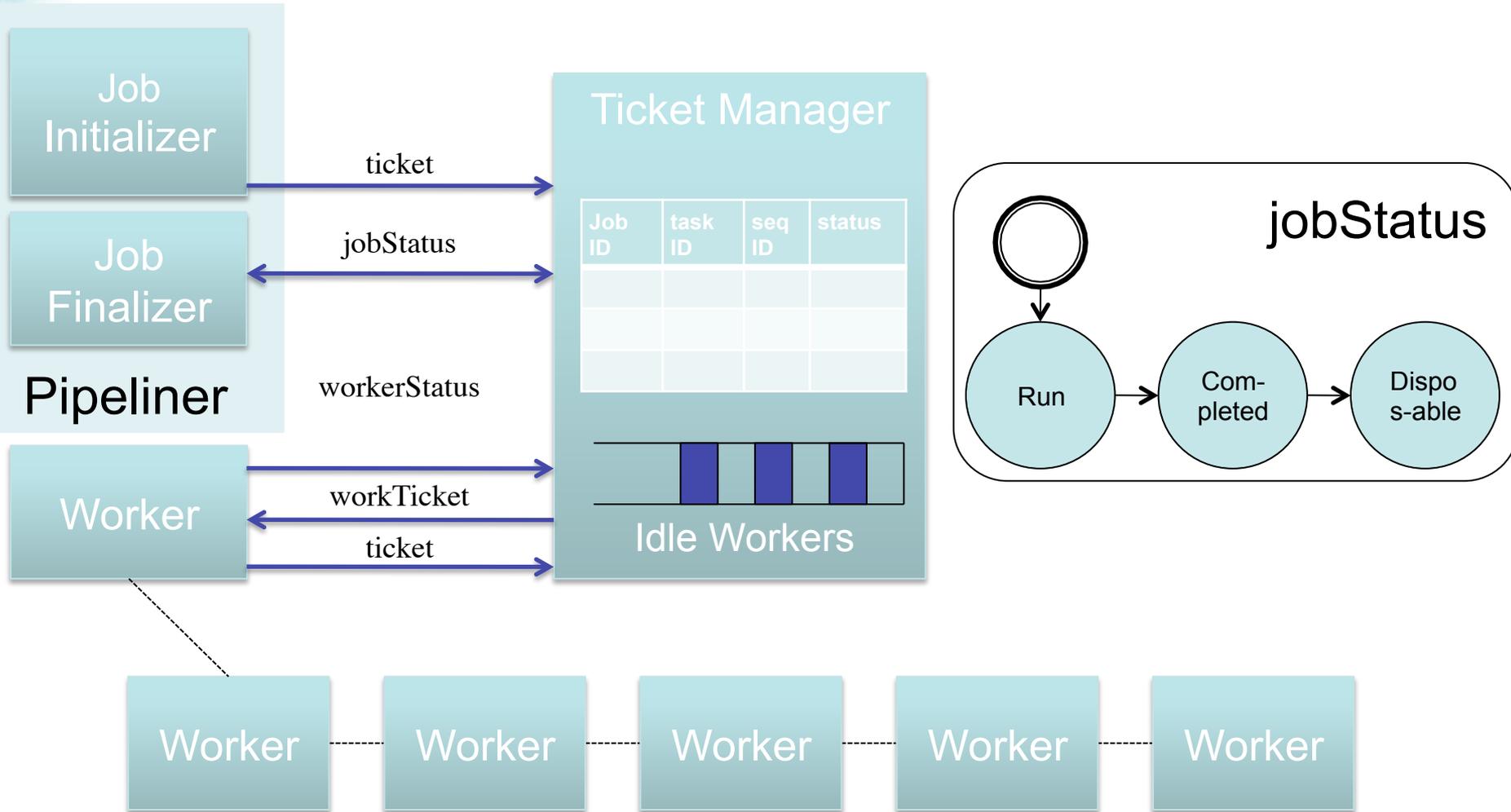


Underlying assumptions

- There are many working processes available for the workflow
- A process can be specialized (do just one type of action) or not (agreeing to whatever should be done next)
- Loose coupling: action triggered upon files generation manifested by events
- Detect failed nodes/processes
- Elimination of a single point of failure calls for a architecture with distributed dataflow processing conductors



DDSTFlow provides runtime environment to coordinate and schedule actor executions





Details in dataflow processing

Internally, a task/ticket represents the execution of a simple actor
Incoming data trigger a dataflow (using pipeliner) to submit a **job**
representing the overall dataflow for the piece of data

A composite actor often is translated into a new job. The
composite actor will wait for the job to finish before firing the next
task(s)

Jobs are independent to one another when it comes to
scheduling (we may need more comprehensive scheduling
strategies later)

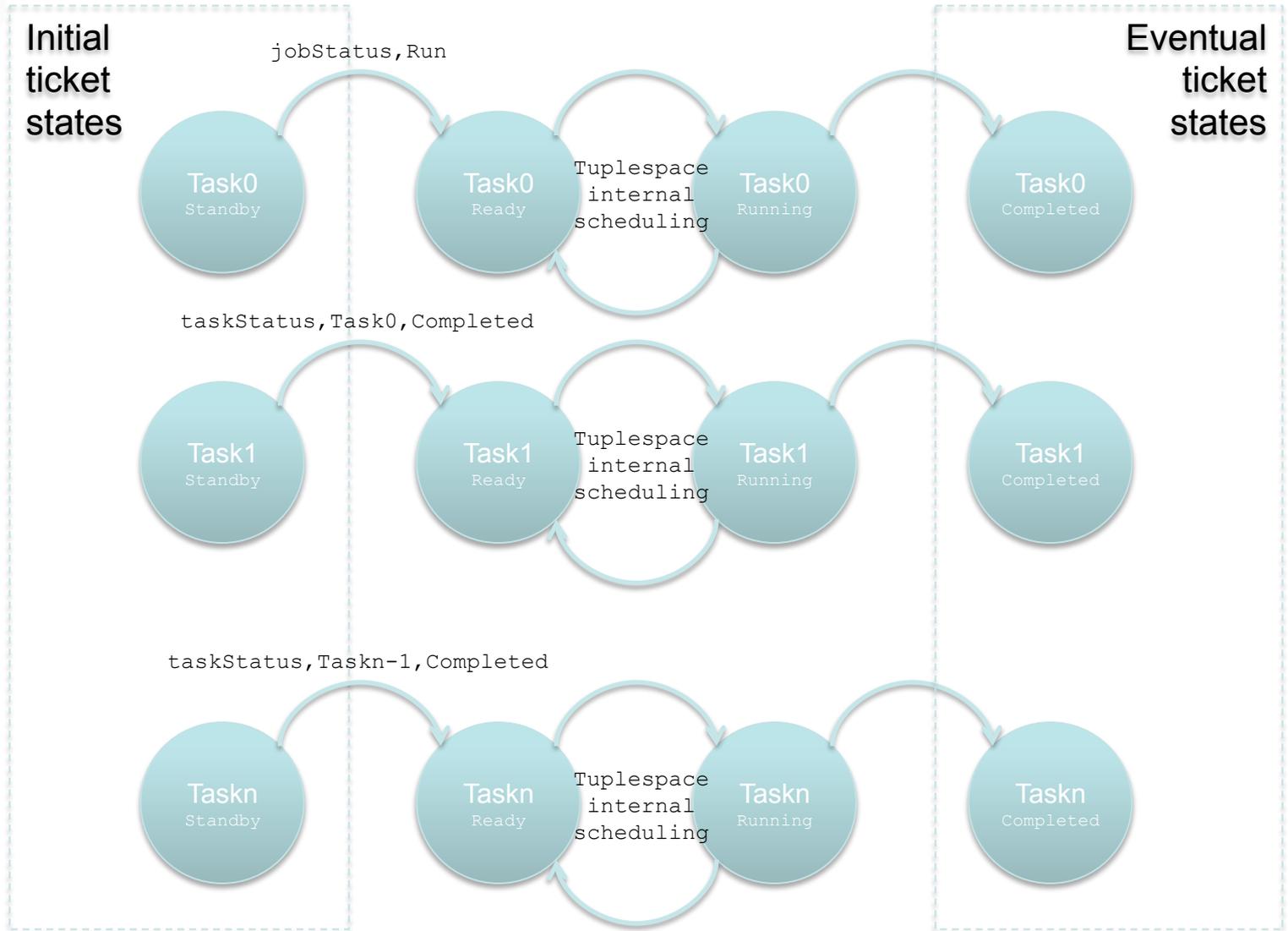
We do not cover stage-in/out of data so far

- Perhaps can be done via actors?



Execution of sequential actors

Tasks executed sequentially



Sequences proceed independently in parallel



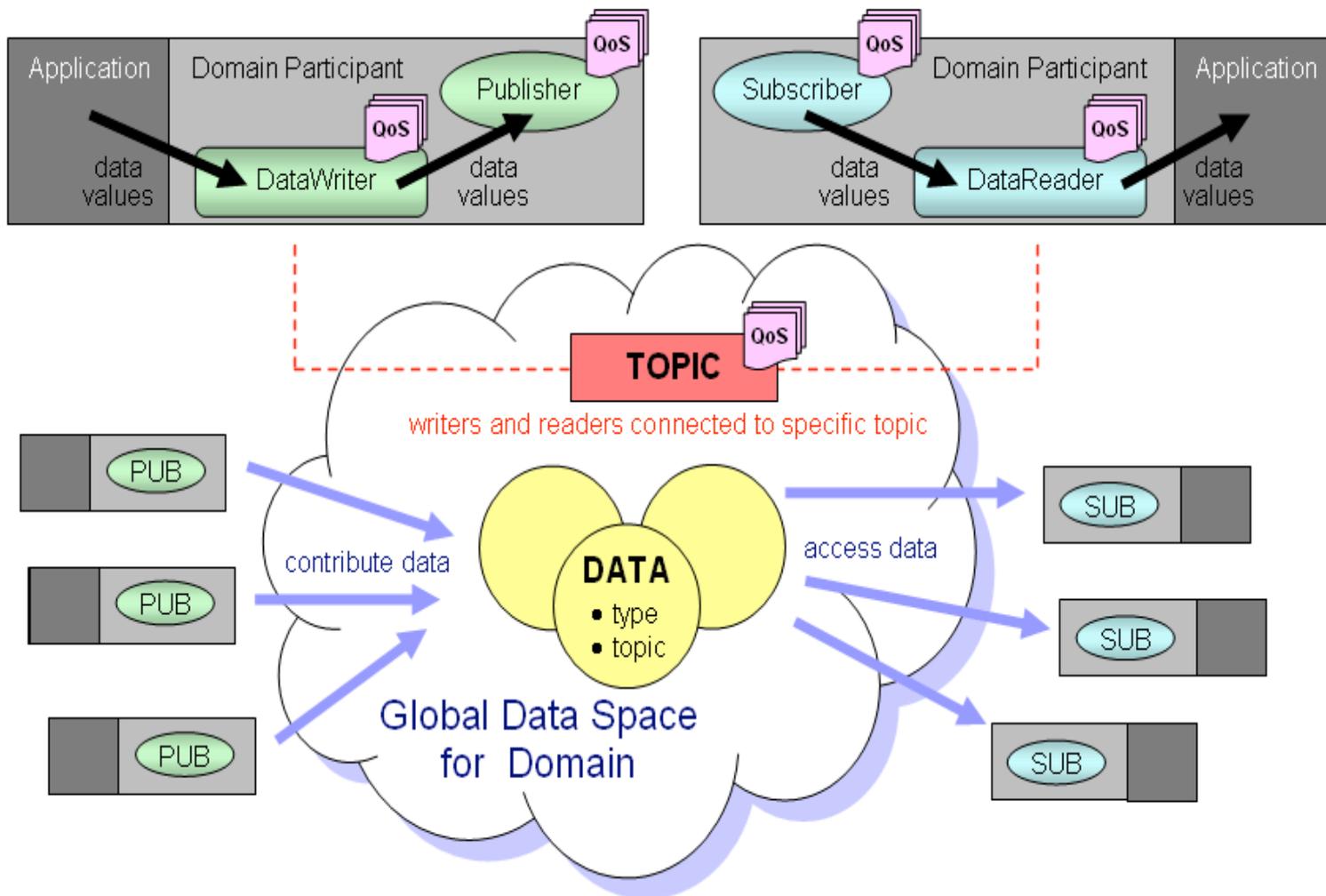
DDSFlow leverages Data Distribution Service (DDS)

- OMG standard for real time publish-subscribe systems
- Free implementations are available and good
- Rich quality-of-service features (resource and time aware, time sequences): to get just what you want, how much you want and when you want
- Works over WAN
- Security support
- C++ and Java binding (we are working on a Python binding)
- Tested in DOD applications (military ships, air traffic control)
- Topics – events with data (can be used to exchange data in memory, we tried FITS data)



DDS decouples participants and delivers information of interest

Data Distribution Service has Data-Centric Publish-Subscribe Interface





Key features in DDSFlow enabled via DDS

- Dynamic workflows
- Pub/sub model enables us to form federations (many clusters on LAN or WAN) easily
 - Other than the ability to pool more resources
- In-memory data exchange (avoiding files)
- Detect processes that have stopped working in a pre-determined time (single node/worker failure)
- Support for auto-snapshot of ticket states
- More resilient to faults
- Enable restart of interrupted dataflow processes (cluster/framework failure)
- Streamlined collection of provenance (through global information space), QA, and system monitoring data via DDS topics



Leveraging various run-time environments

- Currently targeting a single cluster
 - Head-node runs the ticketing server
 - Worker nodes runs the worker daemon
 - Assuming executables are available for all workers
- Can easily extend to support federation of clusters
 - Ticketing servers need to be aware of other server, this can easily implemented with DDS
- Grid environment?
 - Deploy our services using OSG/Teragrid software stack



DDSTFlow current status

- Support execution of sequences and map/reduce's
- Support for dynamic map (variable number of parallel elements)
- Depending on an internal XML-based actor definitions



DDSTFlow plan

- Extend implementation to more control functions and recursive maps (now all ~works for map-reduce systems)
- More support for configuration files
- Better error and status information management
- Support for provenance
- Translate functional dataflow definitions into XML
- Intervention based on data quality or other status events
- Demonstrate that we can implement given examples
- Type verifications

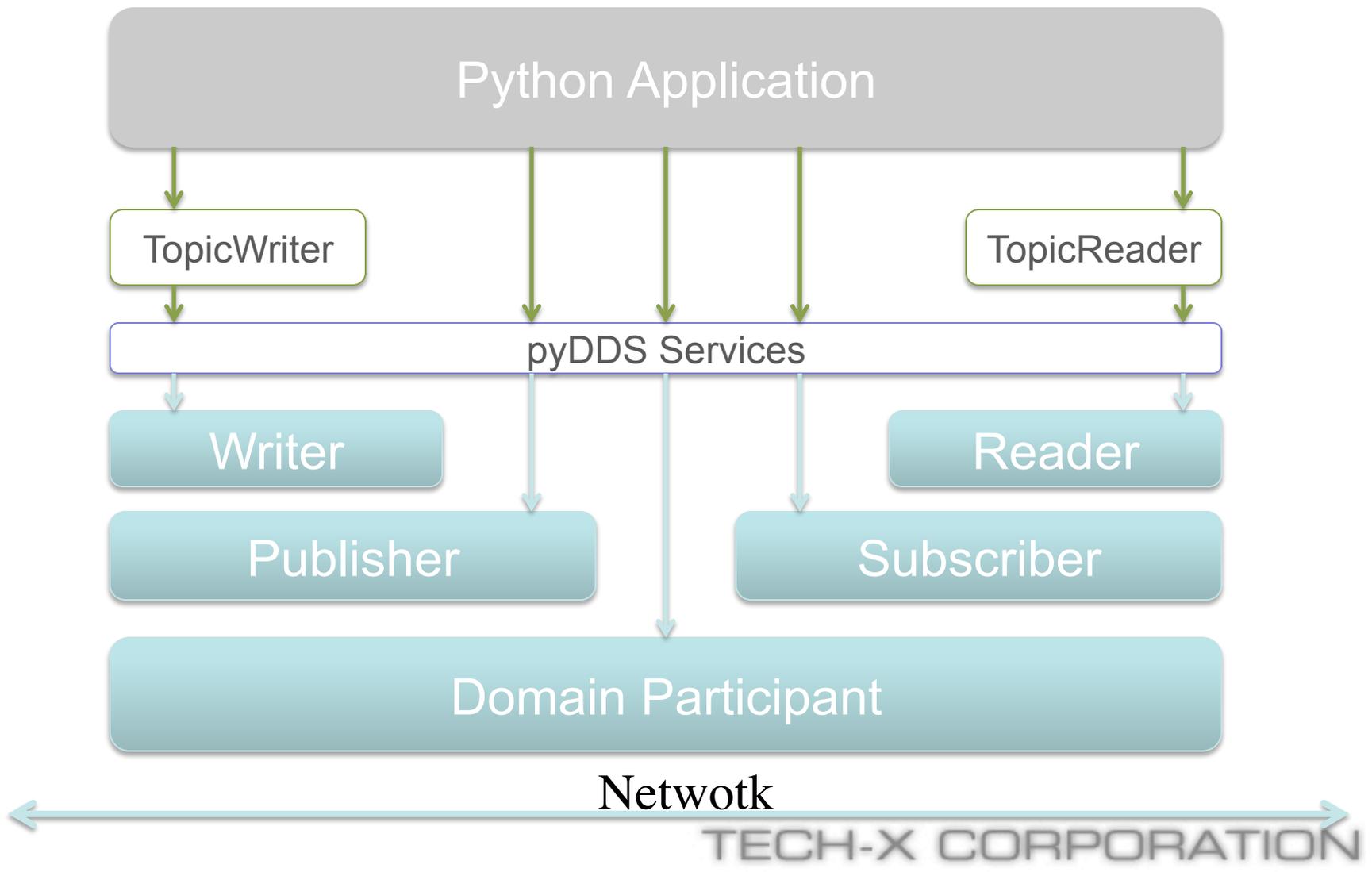


PyDDS: bringing Python programmers to DDS

- Implement prototype Python mappings for DDS which would allow Python applications to participate in DDS data exchanges easily
- Currently, no standardized DDS Python mapping available and for developers to use DDS in their Python applications
- Allow Python developers to interact with DDS data spaces directly
 - Eliminate the need to generate topic-specific Python wrappers of C/C++ mapping codes



PyDDS wrappings





Typical development steps using PyDDS

- Import pydds in Python code
- Dynamically generate topic specific DDS objects using services provided by pydds in python
- Interact with DDS subsystem directly thru 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
 - Natural Python development flow



Example: joining a data domain/partition

```
# A one-stop interface into the pydds global factory methods
import pydds

# We should allow users to define their dataspace/runtime and pass it in as
# an argument to various operations that need it. (See later)
myDataspace = pydds.connect_dataspace("Domain name", "Partition name")

# Perhaps pydds can hold a default dataspace object. If some operations
# that need to know a dataspace object but are not provided one, they will
# fish out the default (nameless, partition less) dataspace and use it instead.

# Likewise, a Dataspace object should contains some
# default subscriber/publisher objects with default QoS policies.
```



Example: manipulating QoS policy sets

```
myQoS = pydds.create_qos()  
myQoS.set_reliable (3000000)  
myQoS.set_transient()  
myQoS.set_keep_last (3)
```



Example: create topics, readers, writers

```
# Creating/Finding a topic in the data space
# Last argument specifies the URI of the topic structure
helloTopic = pydds.create_topic
              ("TopicName",
               myDataspace,
               myQoS,
               "file:///HelloWorld.idl#HelloTopic")

# Creating topic-specific reader/writer:
helloReader = helloTopic.create_reader (readerQoS)
helloWriter = helloTopic.create_writer (writerQoS)
```



Example: writing and reading DDS samples

```
# creating a sample
helloSampl = helloTopic.create_sample
              (message="John Smith", repeat=3)

# publishing the sample
status = helloWriter (helloSample)

# Simple read is straightforward
[ samples, infos ] = hellowReader.read()
sys.stdout.write(samples[0].topicFieldOfInterest...)

# Will look at Twistd and/or Trellis to design the Listener
# callback
```



Design and implementation status

- Working with OpenSplice Community Edition version 5.4.1
- Currently implementing the core OpenSplice communication layer
- We are creating a thin wrapper layer on top of the OpenSplice C APIs
 - OpenSplice's C++ classes are designed for their topic-specific C++ mappings
 - Many are derived from other classes, have multiple layers of inheritance and multiple inheritances
 - All the levels of inheritance need to be taken care of in the bindings, and including all the classes, there are about 150 classes for which Python bindings are needed
 - Using Boost.Python to minimize dependencies to external software



PyDDS next steps

- Develop lightweight C++ wrapper class for Boost.python
- Develop pyDDS API - Pything bridge for OpenSplice communication API in pyDDS
- Implement example Python applications with hand-crafted marshaling and demarshaling
- Define API for dynamic code generation of Python mappings
- Implement such tool