Exploring Dynamic Enactment of Scientific Workflows using Pilot-Abstractions

Mark Santcroos, Barbera DC van Schaik,
Shayan Shahand and Silvia Delgado Olabarriaga
Bioinformatics Laboratory
Academic Medical Center (AMC)
University of Amsterdam, The Netherlands
{m.a.santcroos | b.d.vanschaik | s.shahand | s.d.olabarriaga}@amc.uva.nl

Andre Luckow and Shantenu Jha
RADICAL
Rutgers University
Piscataway, NJ, USA
andre.luckow@gmail.com
shantenu.jha@rutgers.edu

***Abstract*— Current workflow abstractions in general lack: (a) an adequate approach to handle distributed data and (b) proper separation between logical tasks and data-flow from their mapping onto physical location. As the complexity and dynamism of data and processing distribution have increased, optimized mapping of logical tasks to physical resources have become a necessity to avoid bottlenecks. We argue that the management of dynamic data and compute should become part of the runtime system of workflow engines to enable workflows to scale as necessary to address big data challenges and fully exploit distributed computing infrastructures (DCI). In this paper we explore how the P* model for pilot-abstractions, which proposes a clear separation between the logical compute and data units and their realization as a job or a file in some physical resource, could provide these capabilities for such a runtime environment. The Pilot-API provides a general-purpose interface to pilot-abstractions and the ability to assign compute and data resources to them. We share our experience of using the case study of a DNA sequencing pipeline, to re-implement the workflow using the Pilot-API. This first exercise, which resulted in a running application that is discussed here, illustrates the potential of this API to address (a) and (b). We thereby infer that the pilot abstractions (as captured by the P* Model) offer an interesting approach to explore the design of a new generation of workflow management systems and runtime environments that are capable of intelligently deciding on application-aware late binding to physical resources.**

I. Introduction

A wide range of Biomedical applications have been successfully ported and executed on distributed computing infrastructures (DCI) using workflow technology. Workflow management systems can hide details of the underlying infrastructure, and serve as excellent abstraction layers to carry out high throughput experiments [1]. By lowering the barriers to use such complex infrastructures, workflow management systems have been valuable allies in the realization of the e-science vision.

At the Academic Medical Center of the University of Amsterdam (AMC), a workflow-based software platform has been adopted for four years now to enable medical imaging [2] and DNA sequencing [3] research. Workflows are extensively used as the primary abstraction for programming and running applications on the Dutch production grid infrastructure, facilitating access to advanced and novice users. With this approach, data processing "pipelines" can be easily described into grid workflows, and high throughput performance can be achieved by splitting the datasets and distributing their processing on the DCI. Running computations on the DCI has become a trivial exercise on this platform.

With the growth of the data volumes, the solution provided by the platform turned out to be insufficient to address the increasing complexity and dynamism of data and processing distribution [3]. The main challenges shifted from processing to data, and the coordination between the two became more important. Much more than before, it is now necessary to optimize the mapping of logical tasks to physical resources to maintain high throughput. This includes for example, workload balancing to avoid bottlenecks, but also clustering tasks together to minimize data transfers. Furthermore we know from experience that one size does not fit all, and that one approach can be an optimization for one application and a pessimization for another one.

Ideally the location of data and processing should take into account the dynamic availability of resources and the data flow requirements derived from a given workflow execution. In practice we have seen that such optimization is hard to achieve using workflow abstractions as we know today. Optimization attempts often found their way into the workflow descriptions, for example, by early binding a given computation or dataset to resources that were known in advance to have sufficient capacity. In this way the workflow descriptions became "polluted" with all types of DCI-dependent information, and their execution became limited to a subset of the available resources in runtime. Users (the workflow developer or the workflow executors) became responsible for the optimizations that the workflow management system was unable to do.

Although our hands-on experience is limited to a couple of workflow management systems, we argue that this is a fundamental characteristic in most workflow management systems due to (a) the lack of an explicit approach to handle distributed data on a workflow and (b) the lack of proper abstraction to separate logical tasks and data flow from their mapping into physical location on a DCI. Although (b) has been partially addressed by using pilot job framework as back-end of workflow systems, to our knowledge handling

data distribution has not been properly addressed yet in the context of workflow systems. Based on our observations from the backstage of various workflow systems, we realize that implementing our vision of the "ideal case" is very complex and requires some out-of-the-box thinking and looking into fresh alternatives.

Currently we are exploring the P* model for pilot-abstractions [4], which proposed a clear separation between the logical compute and data units and their realization as jobs or files in some physical resources. This model is accompanied by an API – the Pilot-API, which provides an interface to Pilot-Job frameworks that adhere to the P* Model, and which supports programming of distributed applications that can implement complex and dynamic scheduling of resources. We believe that this API has powerful features to address (a) and (b), forming an interesting basis to explore for the construction of a new generation of workflow management systems that are more capable of intelligently deciding on application-aware late binding to physical resources.

In this paper we first characterize the problems we observe in our daily usage of a grid workflow system, and then generalize them in the context of grid workflow systems. The requirements that follow from that are well met by the characteristics of the P* Model. For the sake of completeness, we briefly describe the basis of the P* model and the related Pilot-API. We then present how a concrete workflow for DNA sequence alignment has been manually rewritten using the Pilot-API. Based on a execution of this example we illustrate and discuss the potential of the Pilot-API to implement specific complex and dynamic data/compute distribution case. We conclude the paper with a discussion about how the findings for this specific case could be generalized to form the basis for the implementation of a new generation of workflow systems.

II. The AMC e-BioInfra experience

At the AMC the "e-BioInfra" platform is used daily since 2008 to interface biomedical research applications with the Dutch e-science production infrastructure. At its core there is a workflow management system, MOTEUR [5], which enacts workflows described in the GWENDIA language [6] on a gLite infrastructure using the DIANE [7] Pilot-Job framework. Other services such as monitoring and provenance are implemented at the platform based on the workflow layer. Details about the platform and its components are described elsewhere [8], [9].

The basic principle of this platform is that all applications are ported to the DCI as grid workflows. Advanced users develop and execute their own workflows, whereas novice users execute existing workflows made available through a web interface. Workflows have been chosen because they provide a powerful yet easy-to-use abstraction to port applications to DCIs, a feature recognized by many [1]. The choice for MOTEUR was natural at the time (2007) because it was one of the few workflow systems that had been successfully deployed for medical imaging on a large production grid infrastructure (EGEE). The execution model of MOTEUR fits our usage very well: translation between workflow tasks and grid jobs is done automatically; the grid-enabled workflow components or processes are wrapped into grid jobs using GASW [10]; data transfer between processes is done by reference (URLs for grid storage are passed around by the workflow engine), and the iteration operators enable easy description of workflows that can sweep on lists of files or parameters. The workflow developers could concentrate on the data analysis workflow, and MOTEUR would take care of the DCI details. Similar workflow-based functionality can be found today also in various other systems, e.g. gUSE/WS-PGRADE [11], ASKALON [12] and GWES [13].

With the e-BioInfra, the development and execution of workflows for high-throughput computing became straightforward, and the platform became heavily used at the AMC for medical imaging [2] and DNA sequencing [3] data analyses[1]. This translates into more than 40 different workflows for data analysis so far, of which many more versions exist. Some of them are available at the workflow repository of the SHIWA project[2].

Based on this activity we observed that, although the workflow abstraction provided by MOTEUR is very powerful for describing the flow of processes and data, it is limited regarding optimization of scheduling of data and compute on the DCI because of the fixed or implicit execution pattern. In fact the workflow description contains information that binds them to specific DCIs or resources, which for MOTEUR is concentrated on the component descriptors (GASW descriptors). These descriptors mix definition of the task (inputs, outputs) and its realization as a grid job (files, dependencies, command-line parameters, job requirements, etc). Whereas the first part is resource-agnostic, the second binds the component to a certain type of DCI and even resource (e.g., the executable needs to match the OS and architecture of the target resource, or a particular credential is needed to access the file containing the executable).

The situation is similar to other workflow management systems, as it can be seen in the SHIWA repository: various workflow implementations exist for the same application and workflow engine, but for different DCI configurations. Workflows are presented and positioned as an ideal way to describe scientific experiments that require complex computation, and a literature review of the field of workflow languages and systems shows a clear bias on functional characteristics [1]. In practice the design and implementation of individual workflow systems are often tied to the application type and, more importantly, to the execution scenario of workflows at the computing resources that the developers had in mind. In one way or the other, at some point in a workflow a task is defined that needs to be executed on some resource. Often details of this actual execution are very scarce or non-existent at all in the papers presenting the systems. While from a workflow language perspective this is not the most compelling part, for

[1]From January to July 2012 alone 25 users used the platform to perform 1500 workflow runs

[2]SHIWA: http://www.shiwa-workflow.eu/, repository: http://shiwa-repo.cpc.wmin.ac.uk/shiwa-repo/

an end user it is certainly important.

So in practice such "early binding" is common in workflow systems. Note that such early binding mechanisms provide some control to the workflow developer to "plan" the workflow execution. This feature has been used by the advanced e-bionfra users to address the problems caused by the growth of data volumes, e.g., by manually pre-selecting resources and distributing/replicating the data/compute. Such early binding also clearly limits any (later) optimization based on dynamic resource availability. We argue that late binding – the dynamic coupling to resources (compared to statically in advance) – is a necessary feature of the runtime system to execute scalable applications that fully exploit the dynamic features of the infrastructure. For workflow systems this would enable the dynamic enactment of workflows.

III. Pilot Abstractions

DCIs are by definition comprised of a set of resources that are fluctuating – growing, shrinking, changing in load and capability, in contrast to a static resource utilization model of traditional parallel and cluster computing systems. The ability to utilize a dynamic resource pool is thus an important attribute of any application that needs to utilize DCIs effectively and efficiently. Pilot-Jobs consist of a simple approach for decoupling workload management and resource assignment/scheduling, providing an effective abstraction for dynamic execution and resource utilization. In essence, a Pilot-Job is placeholder job as a container for a set of compute tasks. Not surprisingly, Pilot-Jobs have been very successful abstractions in distributed computing because they liberate applications and/or users from the challenging requirement of mapping specific tasks onto explicit heterogeneous and dynamic resource pools. Pilot-Jobs thus shield applications from having to load-balance tasks across such resources.

A. The P* Model

The P* model introduced in [4] provides a unified model for describing and analyzing common elements of Pilot-Job implementations. The P* approach to pilots has the following natural advantages: (i) permits late binding of workloads to resources and (ii) decoupling tasks from resource management can be extended to data. In [14] the P* model is extended with Pilot-Data.

In the extended model two fundamental abstractions are defined: Pilot-Compute and Pilot-Data. The abstraction of a Pilot-Compute (PC) generalizes the reoccurring concept of utilizing a placeholder job as a container for a set of compute tasks or Compute Units (CU). Instances of that placeholder job are commonly referred to as Pilot-Jobs or pilots. Analogous to Pilot-Compute, the Pilot-Data (PD) abstraction has been introduced to provide a placeholder for data as a container for a set of application-level logical Data Units (DU), separately from their physical allocation. A Compute Unit represents a primary self-containing piece of the processing to be carried out (e.g. a workflow task), while a Data Unit represents some data (e.g., input or output files for a task). The Pilot-Compute and Pilot-Data abstractions enable application- (user-) level control and management of the set of allocated resources, with late binding of Compute Unit and Data Unit to pilots. Figure 1 shows the architecture of the model.

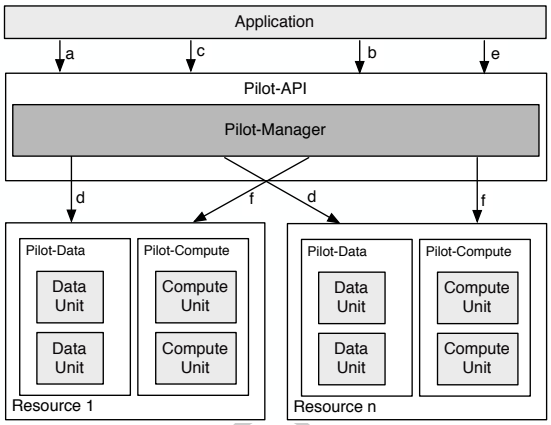

Fig. 1. **P* Architecture:** The application allocates Pilot-Compute (a) and Pilot-Data (b) resources through the Pilot-API. The application also describes the Data Units and passes these to the Pilot-Manager (c). The Pilot-Manager is responsible for transferring (d) the Data Units to their physical locations (Pilot-Data). When the data is in place, the Application can submit Compute Units (e) that will run in Pilot-Computes (f).

The model also defines the notion of *affinity* between units (data and compute), which can be used to reason about the distribution of units.

B. The Pilot-API

The Pilot-API is an interoperable and extensible API that exposes the core functionalities of a Pilot framework to an application via a unified interface that can be used across multiple infrastructures [14]. The API provides five core classes: the *PilotComputeService* for the management of Pilot-Computes, *PilotDataService* for the management of Pilot-Data and the *ComputeDataService* for the combined management of *ComputeUnits* and *Data Units* . The API also defines the *Pilot-Compute* (PC) and *Pilot-Data* (PD) entities. Whereas a PC functions as a placeholder job that can be used for a set of CUs, a PD functions as a placeholder object that reserves storage spaces for a set of DUs. The binding between units and a particular pilot is resolved by the *ComputeDataService* both for compute and data.

IV. Case Study: DNA Sequence Alignment

As a first use case we re-implemented an existing and frequently used data analysis pipeline that performs sequence alignments of short DNA fragments against a reference database with the Burrows-Wheeler Aligner (BWA) program [15]. The alignment is the basis for subsequent analysis such as to find the differences between the genomes of patients and healthy individuals, or for studying gene expression levels.

The dataset used in the experiments was generated by a paired-end sequencing experiment where the protein coding part of the genome of a human patient has been sequenced. In a paired-end sequence experiment, DNA fragments of a

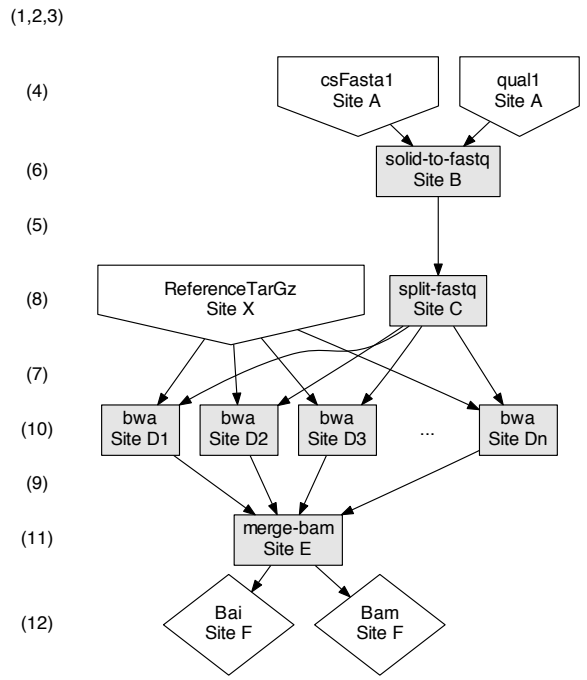

Fig. 2. A workflow that performs sequence alignments of short DNA fragments against a reference database with the BWA program. Numbers in parentheses correspond to code listings. "Site X" labels denote file or process locations. Notation: Pentagons = inputs, diamonds = outputs, and boxes = grid processors.

particular length (e.g. 500 nucleotides) are sequenced from both ends, resulting in two pairs corresponding to "reverse" and "forward" reads. The DNA sequencing machine has produced 4 files (81 GB in total) for this experiment, two containing the DNA sequences (`*.csFasta`) and two containing the nucleotide quality scores (`*.qual`). The result file (7.6 GB) after alignment is provided in `bam` format and contains information about the position of the reads on the human genome, differences compared to the reference genome, alignment quality information, and the original data. The `bam` file contains the necessary information for further analysis steps.

Below we describe the original implementation in GWENDIA followed by the implementation using the Pilot-API.

A. Original GWENDIA workflow

The workflow is illustrated in Fig. 2. Besides the BWA alignment step, it contains data conversion steps to transform from the DNA sequencing machine format (`*.csFasta`) to the BWA format (`*.fastq`), as well as to split/merge the sequences to allow for parallel processing. The alignment of each data chunk is performed against the human genome reference database (version hg19)[3], which is provided to the BWA component as a compressed archive (5.2 GB).

[3]http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/

First a data conversion step takes place for the paired-end files with the `solid-to-fastq` component, where the sequence and quality information are combined into two `fastq` files, one for the forward and one for the reverse reads. Since the datasets are relatively large, these files are split into smaller chunks with the `split-fastq` component. The user can define how large the chunks should be, and the files are split accordingly and transferred back to grid storage. These chunks are then used as input to the sequence alignment step (`bwa`), which is executed in parallel on each chunk of data.

The results of the parallel jobs are stored onto a single directory on grid storage. After all alignments have been performed the intermediate files are passed on to the merge component (`merge-bam`). This last component retrieves the files from grid storage and combines all alignment results into one file, which is the final output of the workflow. (The workflow actually contains two instances of the `split-fastq` for the forward and reverse sequence reads respectively.)

B. Implementation using the Pilot-API

This section presents a walk-through of the implementation of this workflow using the Pilot-API. Although the workflow processes both forward and reverse reads of the sequence, for brevity we will present here only the code listings for forward reads.

```
# Instantiate a PilotComputeService running on localhost
pilot_compute_service = PilotComputeService('localhost')
# Define a PilotComputeDescription
pilot_compute_desc = {
  'service_url': 'cream://gb-ce-amc.amc.nl/cream-pbs-medium'
}
# Create a \pilotcompute through the PilotComputeService
pilot_compute_service.create_pilot(pilot_compute_desc)
```

Listing 1. Creation of a *Pilot-Compute* on the specified compute resource endpoint.

The application starts with acquiring compute resources in listing 1. The *PilotComputeService* is a factory for creating *Pilot-Compute* objects, where the latter is the individual handle to the resource. The application specifies the compute resource where the Pilot should run (in this case CREAM[4]) and can request the size of the resources that should be allocated to the Pilot (e.g., 4 cores). There can be any number of *pilotcompute* instantiated depending on the compute resources available to the application.

```
# Instantiate a PilotDataService
pilot_data_service = PilotDataService()
# Define a PilotDataDescription
pilot_data_description = {
  'service_url': 'srm://tbn18.nikhef.nl/home/mark/pilotdata/'
}
# Create a PilotData
pilot_data_service.create_pilot(pilot_data_description)
```

Listing 2. Creation of a *PilotData* on the specified storage resource.

In listing 2 a *PilotData* is created following the same principle as for the *Pilot-Compute* . Here the application can specify a physical resource (in this case SRM[5] and the required

[4]http://grid.pd.infn.it/cream/

[5]http://sdm.lbl.gov/srm/

storage size, and the service will verify that the resource is available and that the application has the rights to use it. An individual handle is returned to the created *Pilot-Data* .

```
# Instantiate a ComputeDataService
cd_service = ComputeDataService()
# Connect PilotComputeService to ComputeDataService
cd_service.add_pilot_compute_service(pilot_compute_service)
# Connect PilotDataService to Compute Data Service
cd_service.add_pilot_data_service(pilot_data_service)
```

Listing 3. Creation of a *ComputeDataService* and the assignment of the *PilotComputeService* and *PilotDataService* to it.

So far we have allocated resource for both compute and data. Because we need an entity that is able to manage them in a holistic way, in listing 3 we create a *ComputeDataService* and attach both the *PilotComputeService* and *PilotDataServices* to it. The `cd_service` now becomes the handle for the application to deal with these resources.

The previous listings dealt with the generic setup of resources for the application. The following listings are workflow-specific.

```
# Define DataUnitDescription for  short read files
solid_dud = {'file_urls': [
  'Test42_F3.csfasta.bz2',
  'Test42_F3_QV.qual.bz2',
  'Test42_R3.csfasta.bz2',
  'Test42_R3_QV.qual.bz2']
}
# Instantiate the \dataunit and add it to a \pilotdata,
# so that it becomes under control of the system.
solid_du = DataUnit(solid_dud, static=True)
static_pilot_data.add_data_unit(solid_du)
```

Listing 4. Definition and population of a *Data Unit* with files that are already on the remote storage system.

In listing 4 the input *Data Unit* is instantiated and added to a Pilot-Data so that it becomes under control of the system. (The creation of the static Pilot-Data is not shown.) The `solid_du` handle is a logical reference to the input data for the application, and is fully decoupled from the actual physical location.

```
# \dataunit for forward fastq output file
data_unit_description = {'file_urls':
  ['Test42_fwd_read.fastq.gz']
}
fwd_fastq_du = DataUnit(data_unit_description)
```

Listing 5. Definition and creation of the *Data Unit* that will hold the output of the forward solid-to-fastqPE *Compute Unit* .

While the code in listing 4 refers to existing data, the code in listing 5 creates a logical reference to data that currently does not exist yet, namely the file that will be generated by the first conversion step in the workflow.

```
# Define ComputeUnitDescription for solid-to-fastqPE
compute_unit_description = {
  'executable': 'solid-to-fastqPE',
  'arguments': [<...APPLICATION_ARGUMENTS...>],
  'input_data': [solid_du],
  'output_data': [fwd_fastq_du, rev_fastq_du],
}
# Submitting solid-to-fastqPE computeunit
cd_service.submit_compute_unit(compute_unit_description)
```

Listing 6. Definition and submission of the data conversion *Compute Unit* specifying binaries and scripts in the sandbox, the `solid_du` *Data Unit* as an input dependency, and that output results are to be stored into the `fwd_fastq_du` and `rev_fastq_du` *Data Units* .

Once the in- and output are defined, the first *Compute Unit* is defined and submitted in listing listing 6. Along with the execution details for the data conversion step, the application passes the logical handles to the input and output *Data Units* . During the `submit_compute_unit()`, call the *ComputeDataService* determines that the execution is to take place at the existing Pilot-Compute resource. Additionally, it resolves the logical handle `solid_du` to a physical location, and the data is made available to the compute task on that compute resource. When the compute task is complete, the generated files corresponding to the `fwd_fastq_du` and `rev_fastq_du` *Data Units* are automatically stored at the physical location reserved by the corresponding *Pilot-Data* .

```
# DataUnitDescription for fastq chunks from fwd fastq file
data_unit_description = {'file_urls':
  ['Test42_fwd_read*.fastq.gz']
}
fwd_fastq_chunks_du = DataUnit(data_unit_description)
```

Listing 7. Definition and creation of a *Data Unit* that will hold the output of the SplitFastq *Compute Unit* .

The `fwd_fastq_chunks_du` *Data Unit* is created in listing listing 7 to hold an (in advance) unknown number of output files by specifying the output file names using a wildcard '*'. It is up to the *Compute Unit* to create output files with this pattern.

```
# Define ComputeUnitDescription for split-fastq
compute_unit_description = {
  'executable': 'split-fastq',
  'arguments': [<...APPLICATION_ARGUMENTS...>],
  'input_data': [fwd_fastq_du],
  'output_data': [fwd_fastq_chunks_du]
}
# Submitting forward split-fastq \computeunit
cd_service.submit_compute_unit(compute_unit_description)
```

Listing 8. Definition and submission of a *Compute Unit* for splitting the fastq input file into a number of fastq chunk files. In addition to the execution details, this *Compute Unit* takes the `fwd_fastq_du` *Data Unit* as an input dependency. The *Compute Unit* is configured to store its output result into the `fwd_fastq_chunks_du` *Data Units* .

The code in listing 8 is analogous to listing 6, as it describes a task and submits it through the *ComputeDataService*. Note that the input *Data Unit* `fwd_fastq_du` has been instantiated by the *Compute Unit* in listing 6, and will be resolved here by the *ComputeDataService* to its physical location. This task creates an unspecified and unknown number of chunks (files), and stores them in `fwd_fastq_chunks_du`.

```
# List to track all output bam du's
all_bam_dus = []

# Loop over all chunks in pairs (fwd, rev)
for fwd_chunk_du, rev_chunk_du
    in zip(fwd_fastq_chunks_du.split(),
           rev_fastq_chunks_du.split()):

  # Get the filename out of the chunk du
  fwd_chunk = fwd_chunk_du.description['file_urls'][0]
  rev_chunk = rev_chunk_du.description['file_urls'][0]

  # Instantiate the output du for the bwa-short-paired-read
  result_bam = '%s_%s.bam' % (fwd_chunk, rev_chunk)
  data_unit_description = {'file_urls': [result_bam]})
  bam_du = DataUnit(data_unit_description)

  # Add this bam DU to the list of tracked DUs
  all_bam_dus.append(bam_du)
```

Listing 9. This code splits the `fwd_fastq_chunks_du` *Data Unit* on Pilot-API level and generates output data units to hold individual results of parallel jobs.

In the next step in the workflow BWA is run with every individual chunk generated as output of listing 8, and therefore the `fwd_fastq_chunks_du` needs to be split into individual *Data Units* . The loop in listing 9 iterates over pairs of chunks (both forward and reverse reads), extracts the chunks file names to be used as inputs, and creates an output *Data Unit* to hold the result of every individual BWA execution. These units are collected into a list of all created output *Data Units* .

```
# Define ComputeUnitDescription for bwa-short-paired-read
compute_unit_description = {
  'executable': 'bwa-short-paired-read',
  'arguments': [<...APPLICATION_ARGUMENTS...>],
  'input_data': [reference_du, parameters_du,
                 fwd_chunk_du, rev_chunk_du],
  'output_data': [bam_du]
}
cd_service.submit_compute_unit(compute_unit_description)
```

Listing 10. Definition and submission of a *Compute Unit* to run the bwa-short-paired-read task. The task takes a number of *Data Units* as input and stores its output in the `bam_du` *Data Unit* .

The *Compute Unit* in listing 10 is submitted inside the loop of listing 9. Its input and output *Data Units* are defined in the loop. The code for creation of `reference_du` and `parameters_du` is similar to the code in listing 4, but left out for brevity.

```
# Define ComputeUnitDescription for merge-bam
compute_unit_description = {
  'executable': 'merge-bam',
  'arguments': [<...APPLICATION_ARGUMENTS...>],
  'input_data': all_bam_dus,
  'output_data': [merged_du]
}
# print 'Submitting merge-bam computeunit ...'
cd_service.submit_compute_unit(compute_unit_description)
```

Listing 11. Definition and submission of a *Compute Unit* for the merge task. The task takes a list of *Data Units* that were the output of the N bwa-short-paired-read tasks and stores its merged output in the `merged_du` *Data Unit* .

At this stage all individual BWA executions have created an output *Data Unit* that are collected in the `all_bam_dus` list. This list of *Data Units* is passed to the merge-bam *Compute Unit* in listing 11. The `merged_du` output *Data Unit* is created in a similar way as in listing 5 and not included in the paper. It will execute once and create the final output of the workflow.

V. Validation and Experience

A. Runtime Implementation

DIANE is a lightweight job execution control framework for parallel scientific applications [7]. DIANE has a Master/Worker coordination model where the worker announces itself to the master and requests a job to execute. We implemented Pilot-Data functionality on top of DIANE and mapped the DIANE pilot job and data functionalities to the Pilot-API. Besides exposing the P* semantics to the application programmer, the implementation also offers a straightforward scheduler for Compute Unit and Data Unit placement. The Pilot-Data scheduling is also centrally coordinated.

The binding between Data Units and Pilot-Data is decided in a round-robin over the available Pilot-Datas. The Pilot-Data Manager is responsible for transferring the data from local storage to a remote Pilot-Data, from where it can be further downloaded as input by the Compute Unit that runs on a compute resource.

For Data Units that are already on the infrastructure, the Pilot-Data is implicit by its present location. For output Data Units, the binding to a Pilot-Data is decided at the time of the scheduling of the Compute Unit that will populate the Data Unit. If a Compute Unit has input Data Units, then the Pilot-Data of the output Data Unit will be the same as for the input, otherwise a random Pilot-Data is chosen. Based on these decisions, the Compute Unit will be instrumented with the location to download and upload the Data Units once it runs.

The scheduler implemented by *ComputeDataService* has three modes of operation. Mode *STRICT_AFFINITY* controls whether the scheduler should take Compute Unit and Data Unit co-location in mind. It will only schedule a Compute Unit to the site where the input Data Unit is residing. Mode *LOOSE_AFFINITY* is a variant where the affinity defines the preference for co-location, but if that is not feasible, the Compute Unit will be scheduled somewhere else. The third mode is *RANDOM*, that naively schedules Compute Units and Data Units to random locations. By using the *LOOSE_AFFINITY* and *RANDOM* scheduling mode we can contrast the effect of the scheduling alternatives.

B. Environment

The implementation is tested on the BiG Grid infrastructure[6] with the proof of concept application (see Section IV-B). This is the Dutch part of the European Grid Infrastructure (EGI)[7] and consists of 14 sites that host both a Storage Element (SE) and a Compute Element (CE). We interact with the SE using the Storage Resource Manager (SRM) protocol and submit our Pilot-Computes using the CREAM interface of the CEs. For the proximity of Compute/Data locality we consider only *near*

[6]http://www.biggrid.nl/
[7]http://www.egi.eu/

and *far*. The CE and SE that are on the same site are *near*, all other combinations are *far*. In practice there are multiple degrees of *far*, given network properties, but we have not considered them in this implementation.

The scheduling of Pilot-Compute resources in a HTC Grid is an unpredictable activity. As statistics from the Information System (BDII[8]) are not real-time, they can not be used to (reliably) predict queuing times. Also, as application execution times vary depending on where they run, so it can happen that in one phase a Pilot-Compute finishes its Compute Unit earlier than another one. In this case it could happen that the Pilot-Compute expires. Therefore, in our testing we request a suitable number of Pilot-Computes for every phase of the application workflow spread out over the configured CEs. It is unknown in advance whether these resources will ever become available, and therefore the exact amount and location of resources available to the application varies for every workflow enactment.

Figure 3 shows the lifetime of 20 Pilot-Computes during a test run with the respective Compute Units projected on top of them.[9] In addition it shows the number of Compute Units waiting to be scheduled on an available Pilot-Compute.

VI. Related Work and Discussion

A. Related Work

The overheads involved in accessing distributed resources can lead to poor performance that a workflow system is not able to mitigate. Resource provisioning techniques such as advance reservations, multi-level scheduling, and infrastructure as a service (IaaS) may be used to reduce these overheads. Pros and cons of each technique are explained in [16]. For example, Juve et al. showed that a resource provisioning system based on multi-level scheduling called Corral could improve workflow runtime by reducing scheduling overheads [17]. Similarly, Singh and Deelman [18] showed that the completion time of scientific workflows could be reduced by 50% through task clustering and resource provisioning using advance reservations based on statistics or dynamic provisioning mechanisms. Both of these examples use Pegasus WfMS [19] coupled with HTCondor Glidein [20] for resource provisioning which is a mechanism to add one or more remote grid resources to a local HTCondor resource pool temporarily. It uses the same method as typically used by Pilot-Job frameworks, which is to submit a set up task that creates daemons on a remote grid resource. Once the daemons are created and started, they contact the local pool to fetch and run jobs. HTCondor's matchmaking mechanism is used to map jobs to resources, however, no direct control of the pilots is exposed to the user.

The successes and perceived potential of workflow systems has drawn significant attention from researchers, however, general approaches to extensibility and interopability at the workflow system level remain elusive. We attribute this to the fact that a wide range of workflow systems have been developed often using special approaches and to meet niche requirements; although prevalent for workflow systems, bespoke approaches and usage are representative of distributed applications and tools and they come at the expense of broadly applicable design principles and commonly available abstractions [21].

[8]https://tomtools.cern.ch/confluence/display/IS/Home

[9]The failed downloads in Pilot-Compute 19 triggered us to look into the site and we identified that they had deployed a new data transfer library that contained a bug.

B. Discussion

In light of the above, the exercise of writing the NGS alignment application was not to formally characterize the performance or design an optimized system, but to explore the possibility that the workflow could support late-binding for *both* compute and data through the Pilot-API, and that where the pilot-abstractions can be used, or should be exposed as part of the programming model (whether that be for a tool or an application), the Pilot-API is a good approximation to the needs/requirements of the application/tool.

We successfully implemented a workflow using the Pilot-API. This workflow contains representative elements, such as data conversion, data splitting, distribution of processing, and combination of results. The implementation is functionally complete, as a consequence of building upon an existing implementation of Pilot-API for the DIANE Pilot-Job framework. In addition a simple implementation of the Pilot-API for Pilot-Data on SRM has been developed to complete the runtime backend for this application. We will present details of the semantic mapping and other implementation details in future work.

The main feature of this implementation is that, by defining the output data decoupled from its physical location, we allow the system to decide on its most optimal physical location taking into account dynamic conditions. It does not require much imagination to envision a workflow system exploiting these characteristics. In fact, we anticipate several workflow systems such as bio-Kepler, Swift and Moteur to implement such (if not P*) capabilities in the near future.

The backend used in the implementation described here has a simple implementation, without most of the discussed possible intelligence for job and data placement. Notwithstaning this basic implementation, it was already easier to manage the files generated by the tasks; for example, there was no need to specify in advance the precise location of the files because the details are sorted out at runtime by the pilot layer. Straightforward additions to this implementation would be to add more compute and data resources, and let balancing be obtained with random binding of CUs and DUs to the resources. For example, the decision of where to store the files of a given DU could be determined by the location of the consuming CU. Ultimately, the relationship between DU and CU could also be further defined on application-level and used by the ComputeDataService for optimization.

Note however that enhancements to the capabilities of the runtime would not require any modification to the NGS workflow implementation.

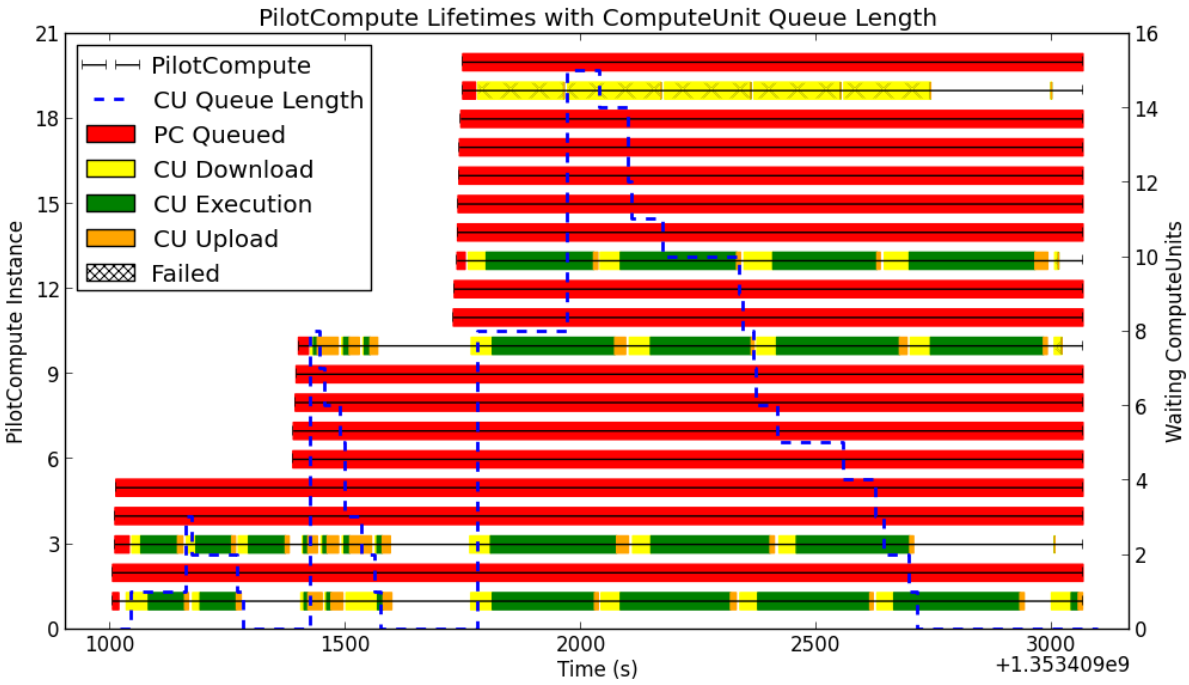

Fig. 3. The lifetime of a workflow from a Pilot-Compute perspective. The horizontal lines are Pilot-Compute lifetimes. They all start in red, meaning that they are queued and waiting to be started. Yellow and orange is download and upload respectively. Green is the actual execution of the application. On the right Y scale with the blue dashed line are the the Compute Units that are waiting to be scheduled on a Pilot-Compute. If a CU block is hatched it means that the action did not complete successfully.

Although Pilots are primarily abstractions for resource management, we leave a discussion of our experience with resource management out of this paper, for effective and interoperable runtime-management has been established [4]. With workflows in general, and specifically with Pilot-Computes involved, the number of resources to acquire and maintain is not a straightforward decision.

VII. Conclusions

As heavy users of workflows to enable e-Science we observed that the scalability and flexibility of workflows is limited by early binding between logical tasks and physical resources and data. In earlier work we have established the use of Pilot-Jobs for workflow execution and this has overcome some of the the computing-caused obstacles. We argue that in order to achieve the scalability required by distributed processing of big data on dynamic infrastructures the introduction of late binding of data into workflow systems is also required.

In this paper we have explored how pilot abstractions can serve as the substrate for implementing late binding in workflows for both compute and data, and illustrated the concept with a representative example. The results are preliminary, but nevertheless indicative of the possibilities.

Although the workflow presented here has been manually translated into an application encoded using the Pilot-API, it illustrates that compute-data orchestration, coordination and execution in a distributed environment can be expressed and captured using the Pilot-API. Furthermore, several workflow runtime systems, i.e., workflow enactment, can be built which exposes Pilot-API as a widely usable public/application interface. The design of the semantic translation, the system interface and information flow between the pilot layer and the runtime layer is a topic for current and future research.

Acknowledgments

This work is financially supported by the AMC ICT innovation fund, the BiG Grid programme funded by the Netherlands Organisation for Scientific Research (NWO), the COMMIT project “e-Biobanking with imaging for healthcare”, NSF CHE-1125332 (Cyber-enabled Discovery and Innovation), NSF-ExTENCI (OCI-1007115) and US Department of Energy Award No. DE-SC0008591. Experiments were executed on resources provided by BiG Grid, the Dutch e-Science Grid. Last but not least, we thank e-BioScience and RADICAL group members for their support and input.

References

[1] E. Deelman, D. Gannon, M. Shields, and I. Taylor, “Workflows and e-Science: An overview of workflow system features and capabilities,” *Future Generation Computer Systems-The International Journal Of Grid Computing-Theory Methods And Applications*, vol. 25, no. 5, pp. 528–540, 2009.

[2] M. W. A. Caan, S. Shahand, F. M. Vos, A. H. C. van Kampen, and S. Olabarriaga, “Evolution of grid-based services for Diffusion Tensor Image analysis,” *Future Generation Computer Systems*, vol. 28, no. 8, pp. 1194–1204, Oct. 2012.

[3] A. C. Luyf, B. D. van Schaik, M. de Vries, F. Baas, A. H. van Kampen, and S. D. Olabarriaga, "Initial steps towards a production platform for DNA sequence analysis on the grid," *BMC Bioinformatics*, vol. 11, no. 1, p. 598, Dec. 2010.

[4] A. Luckow, M. Santcroos, O. Weidner, A. Merzky, S. Maddineni, and S. Jha, "Towards a common model for pilot-jobs," in *HPDC '12: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*. ACM, Jun. 2012.

[5] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec, "Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR," *International Journal of High Performance Computing Applications*, vol. 22, no. 3, pp. 347–360, 2008.

[6] J. Montagnat, B. Isnard, T. Glatard, K. Maheshwari, and M. Fornarino, "A data-driven workflow language for grids based on array programming principles," *WORKS '09: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, Nov. 2009.

[7] J. Mościcki, "Diane-distributed analysis environment for grid-enabled simulation and analysis of physics data," *Nuclear Science Symposium Conference Record, 2003 IEEE*, vol. 3, pp. 1617–1620 Vol. 3, 2003.

[8] S. Olabarriaga, T. Glatard, and P. de Boer, "A Virtual Laboratory for Medical Image Analysis," *Information Technology in Biomedicine, IEEE Transactions on*, vol. 14, no. 4, pp. 979–985, 2010.

[9] S. Shahand, M. Santcroos, A. H. C. Kampen, and S. D. Olabarriaga, "A Grid-Enabled Gateway for Biomedical Data Analysis," *Journal of Grid Computing*, Oct. 2012.

[10] T. Glatard, J. Montagnat, D. Emsellem, and D. Lingrand, "A Service-Oriented Architecture enabling dynamic service grouping for optimizing distributed workflow execution," *Future Generation Computer Systems*, vol. 24, no. 7, pp. 720–730, Jul. 2008.

[11] P. Kacsuk, Z. Farkas, M. Kozlovszky, G. Hermann, A. Balasko, K. Karoczkai, and I. Marton, "WS-PGRADE/gUSE Generic DCI Gateway Framework for a Large Variety of User Communities," *Journal of Grid Computing*, Nov. 2012.

[12] T. Fahringer, A. Jugravu, S. Pllana, R. Prodan, C. Seragiotto, and H.-L. Truong, "ASKALON: a tool set for cluster and Grid computing," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 143–169, 2005.

[13] A. Hoheisel, "Grid Workflow Execution Service-Dynamic and interactive execution and visualization of distributed workflows," *Proceedings of the Cracow Grid Workshop*, vol. 2, pp. 13–24, 2006.

[14] A. Luckow, M. Santcroos, A. Merzky, O. Weidner, P. Mantha, and S. Jha, "P*: A Model of Pilot-Abstractions," in *8th IEEE International Conference on e-Science 2012*, 2012.

[15] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics (Oxford, England)*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009.

[16] G. Juve and E. e. . e. . I. F. I. C. o. Deelman, "Resource Provisioning Options for Large-Scale Scientific Workflows," in *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, 2008.

[17] G. Juve, E. Deelman, K. Vahi, and G. Mehta, "Experiences with resource provisioning for scientific workflows using Corral," *Scientific Programming*, vol. 18, no. 2, pp. 77–92, 2010.

[18] G. Singh and E. Deelman, "The interplay of resource provisioning and workflow optimization in scientific applications," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 16, Nov. 2011.

[19] E. Deelman, G. Singh, M. H. Su, J. Blythe, and Y. Gil, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific ...*, 2005.

[20] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The Condor experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2âĂŘ4, pp. 323–356, 2005.

[21] S. Jha, M. Cole, D. S. Katz, M. Parashar, O. Rana, and J. Weissman, "Distributed Computing Practice for Large-Scale Science & Engineering Applications," 2012, computing and Concurrency: Practise and Experience (in press).