P*: A Model of Pilot-Abstractions

Andre Luckow
Center for Computation and Technology
Louisiana State University
216 Johnston
Baton Rouge, LA
aluckow@cct.lsu.edu

Mark Santcroos
Bioinformatics Laboratory
Academic Medical Center
University of Amsterdam
Meibergdreef 9
Amsterdam, The Netherlands
m.a.santcroos@amc.uva.nl

Ole Weidner
Center for Computation and Technology
Louisiana State University
216 Johnston
Baton Rouge, LA
oweidner@cct.lsu.edu

Andre Merzky
Center for Computation and Technology
Louisiana State University
216 Johnston
Baton Rouge, LA
amerzky@cct.lsu.edu

Sharath Maddineni
Center for Computation and Technology
Louisiana State University
216 Johnston
Baton Rouge, LA
smaddineni@cct.lsu.edu

Shantenu Jha*
Center for Autonomic Computing
Rutgers University
94 Brett Road
Piscataway, NJ
shantenu.jha@rutgers.edu

ABSTRACT

Pilot-Jobs support effective distributed resource utilization, and are arguably one of the most widely-used distributed computing abstractions, as measured by the number and types of applications that use them, as well as the number of production distributed cyberinfrastructures that support them. Not surprisingly, there are multiple, distinct and incompatible implementations of pilot-jobs. Often these implementations are strongly coupled to the distributed cyberinfrastructure they were originally designed for. Additionally, in spite of broad uptake, there does not exist a well defined, unifying conceptual model of Pilot-Jobs which can be used to define, compare and contrast different implementations. This presents a barrier to extensibility and interoperability. This paper is an attempt to (i) provide a minimal but complete model (P*) of Pilot-Jobs, (ii) establish the generality of the P* Model by mapping various existing and well known Pilot-Job frameworks such as Condor and DIANE to P*, (iii) derive an interoperable and extensible API for the P* Model (Pilot-API), (iv) validate the implementation of the Pilot-API by concurrently using multiple *distinct* pilot-job frameworks on distinct production distributed cyberinfrastructures, and (v) apply the P* Model to Pilot-Data.

1. INTRODUCTION AND OVERVIEW

The seamless uptake of distributed infrastructures by scientific applications has been limited by the availability of extensible, pervasive and simple-to-use abstractions at multiple levels – at development, deployment and execution stages of scientific applications [1]. Even where meaningful abstractions exist, the challenges of providing them in an extensible, reliable and scalable manner so as to support multiple applications and are usable on different infrastructures are formidable. The lack of appropriate implementations has in fact resulted in "one-off" solutions that address challenges in a highly customized manner. Tools and implementations are often highly dependent on and tuned to a specific execution environment, further impacting portability, reusability and extensibility. Semantic and interface incompatibility are certainly barriers, but so is the lack of a common architecture and conceptual framework upon which to develop similar tools a barrier.

This general state of affairs also captures the specific state of the abstractions provided by *Pilot-Jobs (PJ)*. Pilot-Jobs have been one of the most successful abstractions in distributed computing. Distributed cyber/e-infrastructure is by definition comprised of a set of resources that is 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 distributed cyberinfrastructure (DCI) efficiently. As a consequence of providing a simple approach for decoupling workload management and resource assignment/scheduling, PJ provide an effective abstraction for dynamic execution and resource utilization in a distributed context.

The fundamental reason for the success of the PJ abstraction is that PJ liberate applications/users from the challenging requirement of mapping specific tasks onto explicit heterogeneous and dynamic resource pools. PJ also thus shields application from having to load-balance tasks across such resources. The Pilot-Job abstraction is also a promising route to address specific requirements of distributed scientific applications, such as coupled-execution and application-level scheduling [2, 3].

A variety of PJ frameworks have emerged: Condor-G/ Glide-in [4], Swift [5], DIANE [6], DIRAC [7], PanDA [8], ToPoS [9], Nimrod/G [10], Falkon [11] and MyCluster [12] to name a few. Although they are all, for the most parts, functionally equivalent – they support the decoupling of workload submission from resource assignment – it is often im-

*Author for correspondence

possible to use them interoperably or even just to compare them functionally or qualitatively. The situation is reminiscent of the proliferation of functionally similar yet incompatible workflow systems, where in spite of significant a posteriori effort on workflow system extensibility and interoperability (thus providing post-facto justification of its needs), these objectives remains difficult if not infeasible.

We present the P* Model in §2. Our objective is to provide a minimal, but complete model – hence referred to as P* Model, for Pilot-Job abstractions. The P* Model provides a conceptual basis to compare and contrast different PJ frameworks – which to the best of our knowledge is the first such attempt. In §3 we validate this claim by analyzing well-known PJ frameworks (BigJob, Condor-G/Glide-in, DIANE, Swift-Coaster) using the P* Model.

§4 of this paper motivates and describes the Pilot-API; we discuss how existing and widely used Pilot-Job frameworks, can be used through the Pilot-API. §5 describes the experiments and performance measurements used to characterize the workings of the Pilot-API and to demonstrate interoperability across middleware, platform and different PJ frameworks. To further substantiate the impact of P*, we will demonstrate interoperability between different PJ frameworks – BigJob and DIANE. We believe this is also the first demonstration of concurrent interoperation of different Pilot-Job implementations. Performance advantages arising from the ability to distribute part of a data-intensive workload are discussed; interoperable capabilities increase flexibility in resource selection and optimization.

We investigate generalizations to the base P* Model in §6. A natural and logical extension of the P* Model arises from the need to extend it to include data in addition to computational tasks. This leads to analogous abstraction to the Pilot-Job: the *Pilot-Data (PD)* abstraction. The potentially consistent treatment of data and compute suggests symmetrical compute and data elements in the model; thus we refer to this model as the P* Model ("P-star").

It is worth noting that Pilot-Jobs are used on every major national and international DCI, including NSF/XSEDE, NSF/DOE Open Science Grid, EU EGI and others, to support hundreds and thousands of tasks daily, thus we believe the impact and validation of this paper lies in its ability to not only influence but also bridge the theory and practice of Pilot-Jobs, and thus multiple domains of science dependent on distributed cyberinfrastructure.

2. THE P* MODEL OF PILOT-ABSTRACTIONS

To provide a common analytical framework to understand the most commonly used Pilot-Jobs, we present the P* Model of pilot-abstractions. The P* model is derived from an analysis of many Pilot-Job implementations; based upon this analysis, we first present the common *elements* of the P* Model, followed by a description of the *characteristics* that determine the interaction of these elements and the overall functioning of a Pilot-Job framework that is consistent with the P* Model.

Before we proceed to discuss the P* Model, it is important to emphasize that there exist a plethora of terms — abstraction, model, framework, and implementation, that are overloaded and overlapping, and often even used inconsistently in the literature; thus we establish their context and usage in this paper.

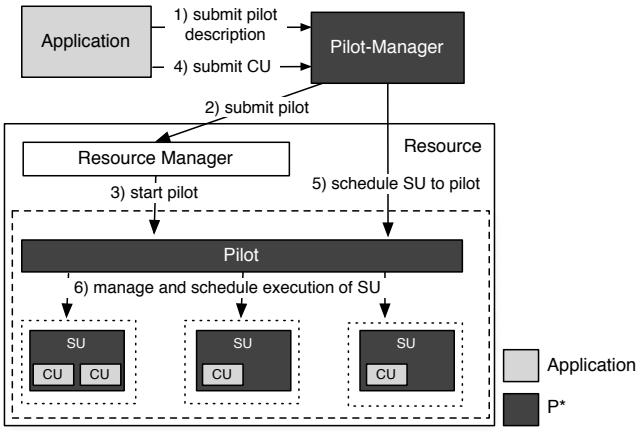

Figure 1: P* Model: Elements, Characteristics and Interactions: The manager has two functions: it manages 1) Pilots (step 1-3) and 2) the execution of CUs. After a CU is submitted to the manager, it transitions to an SU, which is scheduled to a Pilot by the PM. The Pilot then schedules the SU to an available resource.

Terms and Usage: The *abstraction* of a Pilot-Job generalizes the reoccurring concept of utilizing a placeholder job as a container for a set of compute tasks; instances of that placeholder job are commonly referred to as *Pilot-Jobs* or *pilots*. The P* *model* provides a comprehensive description of Pilot-Job abstractions based on a set of identified elements and their interactions. The P* Model can be used as a *conceptual model*, for analyzing different implementations of the Pilot-Job abstraction. The *Pilot-API* exposes a sub-set of the P* elements and characteristics to applications. It is important to distinguish P* which provides a conceptual (abstract) model from an implementation of the P* Model. A *Pilot-Job framework* refers to a specific instance of a Pilot-Job implementation and associated infrastructure that provides the complete Pilot-Job functionality (e. g. Condor-G/Glide-in and DIANE).

2.1 Elements of the P* Model

This sub-section defines the elements of the P* Model:

• **Pilot (Pilot-Compute):** The Pilot is the entity that actually gets submitted and scheduled on a resource. The PJ provides application (user) level control and management of the set of allocated resources.

• **Compute Unit (CU):** A CU encapsulates a self-contained piece of work (a task) specified by the application that is submitted to the Pilot-Job framework. There is no intrinsic notion of resource associated with a CU.

• **Scheduling Unit (SU):** SUs are the units of scheduling, internal to the P* Model, i.e., it is not known by or visible to an application. Once a CU is under the control of the Pilot-Job framework, it is assigned to an SU.

• **Pilot-Manager (PM):** The PM is responsible for (i) orchestrating the interaction between the Pilots as well as the different components of the P* Model (CUs, SUs) and (ii) decisions related to internal resource assignment (once resources have been acquired by the Pilot-Job). For example, an SU can consists of one or more CUs. Further, CUs and

SUs can be combined and aggregated; the PM determines how to group them, when SUs are scheduled and executed on a resource via the Pilot, as well as how many resources to assign to an SU.

An application kernel is the actual binary that gets executed. The application utilizes a PJ framework to execute multiple instances of an application kernel (an ensemble) or alternatively instances of multiple different application kernels (a workflow). To execute an application kernel, an application must define a CU specifying the application kernel as well as other parameters. This CU is then submitted to the PM (as an entry point to the Pilot-Job framework), where it transitions to an SU. The PM is then responsible for scheduling the SU onto a Pilot and then onto a physical resource. As we will see in §3, the above elements can be mapped to specific entities in many Pilot-Jobs in existence and use; often more than one logical element may be rolled into a specific entity in a Pilot-Job.

2.2 Characteristics of P* Model

We propose a set of fundamental properties/characteristics that describe the interactions between the elements, and thus aid in the description of P* Model.

Coordination: The coordination characteristics describe how various elements of the P* Model, i. e. the PM, the Pilot, the CUs and the SUs, interact. A common coordination pattern is master/worker (M/W): the PM represents the master process that controls a set of worker processes, the Pilots. The point of decision making is the master process. In addition to the *centralized* M/W, M/W can also be deployed *hierarchically*. Alternatively, coordination between the elements, in particular the Pilots, can be performed so as to be *decentralized*, i. e. without central decision making point.

Communication: The communication characteristics describes the mechanisms for data exchange between the elements of the P* Model: e. g. messages (point-to-point, all-to-all, one-to-all, all-to-one, or group-to-group), streams (potentially unicast or multicast), publish/subscribe messaging or shared data spaces.

Scheduling: The scheduling characteristics describes the process of mapping a SU to resources via a Pilot and potential multiple levels of scheduling. Scheduling has a spatial component (which SU is executed on which Pilot?) but also a temporal component (when to bind?). The different scheduling decisions that need to be made are representative of multi-level scheduling decisions that are often required in distributed environments. For example, when should a SU be bound to a Pilot? An SU can be bound to a Pilot either before the Pilot has in turn been scheduled (*early* binding), whereas *late* binding occurs if the SU is bound after the Pilot has been scheduled. In general, there are multiple-levels at which scheduling decisions, i.e., resource selection and binding, are made.

The term *agent*, although not a part of the P* Model, finds mention when discussing implementations. For the purposes of this paper, an agent refers to a "proxy process" that has some decision making capability, and could aid the implementation of one or more of the characteristics of the P* Model – coordination, communication, scheduling, within a Pilot-Job framework. These agents can be used to enforce a set of (user-defined) policies (e.g. resource capabilities, data/compute affinities, etc.) and heuristics.

2.3 Putting it all together

Figure 1 illustrates the interactions between the elements of the P* Model. First, the application specifies the capabilities of the resources required using a Pilot-Job description (step 1). The PM then submits the necessary number of Pilots to fulfill the resource requirements of the application (step 2). Each Pilot is queued at the resource manager, which is responsible for starting the Pilot (step 3). There can be variations of this flow: while in the described model, the application defines the required resources, the PM could also decide based on the submitted CU workload whether and when it submits new Pilots.

The application can submit CUs to the PM at any time (step 4). A submitted CU becomes an SU, i. e. the PM is now in control of it. In the simplest case one CU corresponds to one SU; however, SUs can be combined and aggregated to optimize throughputs and response times. Commonly, a hierarchical M/W model for coordination is used internally: the PM uses M/W to coordinate a set of Pilots, the Pilot itself functions as manager for the execution of the assigned SUs.

Scheduling decisions can be made on multiple levels. The PM is responsible for selecting a Pilot for an SU (step 5). A Pilot is bound to a physical resource on which it is responsible for a particular resource set. Once a SU has been scheduled to a Pilot, the Pilot decides when and on which part of the resource an SU is executed. Further, the Pilot manages the subsequent execution of an SU (step 6). There can be variations of this flow. PJ frameworks with decentralized decision making e. g. often utilize autonomic agents that accept respectively pull SUs according to a set of defined policies.

3. PILOT-JOB FRAMEWORKS

As more applications take advantage of dynamic execution, the Pilot-Job concept has grown in popularity and has been extensively researched and implemented for different usage scenarios and infrastructure. The aim of this section is to show that our P* Model can be used to explain/understand some of these PJ frameworks. In particular we focus on Condor-G/Glide-in, BigJob and DIANE.

3.1 Condor-G/Glide-in

The Condor project pioneered the concept of Pilot-Jobs by introducing the *Condor-G/Glide-in* mechanisms [4] which allow the temporary addition of Globus GRAM controlled HPC resources to a Condor resource pool.

In this scenario, the Pilot is exposed as a complete Condor pool that is started via the Globus GRAM service of a resource. This mechanism is referred to as Condor Glide-in. Subsequently, jobs (CUs) can be submitted to the Condor Glide-in pool using the standard Condor tools and APIs. Condor utilizes a master/worker coordination model. The PJ manager is referred to as the Condor Central Manager. The functionality of the Central Manager is provided by several daemons: the `condor_master` that is generally responsible for managing all daemons on a machine, the `condor_collector` which collects resource information, the `condor_negotiator` that does the matchmaking and the `condor_schedd` that is responsible for managing the binding and scheduling process. Condor generally does not differentiate between workload, i. e. CU, and schedulable entity, i. e.

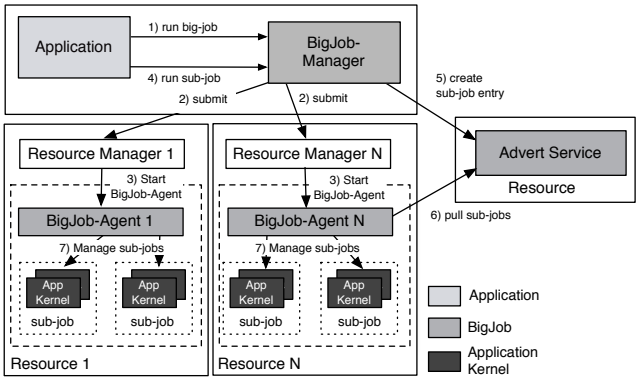

Figure 2: BigJob Architecture and Mapping to P*: The BJ architecture resembles many elements of the P* Model. The BigJob-Manager is the central Pilot-Manager, which orchestrates a set of Pilots. Each Pilot is represented by a decentral component referred to as the BigJob-Agent. Sub-job – the CUs– are submitted via the PM. CUs are mapped 1:1 to SUs.

SU. Both entities are referred to as job. However, it supports late binding, i. e. resources a job is eventually submitted to are not required to be available at submission time. The scheduler matches the capabilities required by a CU to the available resources. This process is referred to as matchmaking. Further, a priority-based scheduler is used. For communication between the identified elements Condor utilizes point-to-point messaging using a binary protocol on top of TCP.

Different fault tolerance mechanisms, such as automatic retries, are supported. Further, Condor supports different security mechanisms: for authentication it integrates both with local account management systems (such as Kerberos) as well as grid authentication systems such as GIS. Communication traffic can be encrypted.

3.2 BigJob: A SAGA-based PJ Framework

BigJob (BJ) [13, 14] is a SAGA-based PJ framework. BJ has been designed to be general-purpose and extensible. While BJ has been originally built for HPC infrastructures, such as XSEDE and FutureGrid, it is generally also usable in other environments. This extensibility mainly arises from the usage of SAGA as a common API for accessing distributed resources. SAGA [15, 16] provides a simple, POSIX-style API to the most common grid functions at a sufficiently high-level of abstraction so as to be independent of the diverse and dynamic grid environments.

Figure 2 illustrates the architecture of BJ and its mapping to P*. The architecture reflects all P* elements: The BJ-Manager is the Pilot-Manager responsible for coordinating the different components of the frameworks. The BigJob-Agent is the actual Pilot that is submitted to a resource. CUs are referred to as sub-jobs. Internally CUs are mapped 1:1 to SUs.

BJ implements the following P* characteristic: As coordination model the M/W scheme is used: The BJ-Manager is the central entity, which manages the actual Pilot, the BJ-Agent. Each agent is responsible for gathering local information, for pulling sub-jobs from the manager, and for executing SUs on its local resource. The SAGA Advert Service is used for communication between manager and agent. The Advert Service (AS) exposes a shared data space that can be accessed by manager and agent, which use the AS to realize a push/pull communication pattern, i. e. the manager pushes a SU to the AS while the agents periodically pull for new SUs. Results and state updates are similarly pushed back from the agent to the manager. Further, BJ provides a pluggable communication & coordination layer and also supports other c&c systems besides the AS, e. g. Redis [17] and ZeroMQ [18].

BJ currently uses a simple scheduling mechanism based on an internal queue: each CU submitted to the BigJob framework is mapped to a SU. Binding can take place at submission time (early binding) or delayed in case of multiple Pilots (late binding). For scheduling, a simple FIFO queue is used (see also table 2).

In many scenarios it is beneficial to utilize multiple resources, e. g. to accelerate the time-to-completion or to provide resilience to resource failures and/or unexpected delays. BigJob allows dynamic resource additions/removals as well as late binding. The support of this feature depends on the backend used. To support this feature on top of various BigJob implementations that are by default restricted to single resource use, the concept of a BigJob pool is introduced. A BigJob pool consists of multiple BJs (each BigJob managing one particular resource). An extensible scheduler is used for dispatching CUs to one of the BJs of the pool (late binding). By default a FIFO scheduler is used.

3.3 DIANE

DIANE [6] is a task coordination framework, which was originally designed for implementing master/worker applications, but also provides PJ functionality for job-style executions. it utilizes a single hierarchy of worker agents and a PJ manager referred to as `RunMaster`. For the spawning of PJs a separate script, the so-called submitter script, is required. For the access to the physical resources the GANGA framework [19] can be used. Once the worker agents are started they register themselves at the RunMaster. For communication between the RunMaster and worker agents point-to-point messaging based on CORBA [20] is used. CORBA is also used for file staging.

DIANE is primarily designed with respect to HTC environments (such as EGI [21]), i. e. one PJ consists of a single worker agent with the size of 1 core. To use DIANE on HPC environments, a so-called multinode submitter script can be used: the scripts starts a defined number of worker agents on a certain resource. However, CUs will be constrained to the specific number of cores managed by a worker agent. By default a CU is mapped to a SU; application can however implement smarter allocation schemes, e. g. the clustering of multiple CUs into a SU.

DIANE includes a simple capability matcher and FIFO-based task scheduler. Plugins for other workloads, e. g. DAGs or for data-intensive application, exist or are under development. The framework is extensible: applications can implement a custom application-level scheduler.

DIANE is, like BJ, a single-user PJ, i. e. each PJ is executed with the privileges of the respective user. Also, only CUs of this respective user are executed by DIANE. DIANE supports various middleware security mechanisms (e. g. GSI,

P* Element	BigJob	DIANE	Condor-G/Glide-in	Swift/Coaster
Pilot-Manager	BigJob Manager	RunMaster	condor_master, condor_collector, condor_negotiator, condor_schedd	Coaster Service
Pilot	BigJob Agent	Worker Agent	condor_master, condor_startd	Coaster Worker
Compute Unit (CU)	Task	Task	Job	Application Interface Function (Swift Script)
Scheduling Unit (SU)	Sub-Job	Task	Job	Job

Table 1: Mapping P* elements and PJ Frameworks: While each PJ framework maintains its own vocabulary, each of the P* elements can be mapped to one (or more) components of the different PJ frameworks.

X509). For this purpose it relies on GANGA. Further, DIANE supports fault tolerance: basic error detection and propagation mechanisms are in place. Further, an automatic re-execution of CUs is possible.

3.4 Swift/Coaster

Swift [5] is a scripting language designed for expressing abstract workflows and computations. The language provides, amongst many other things, capabilities for executing external applications, as well as the implicit management of data flows between application tasks. The runtime environment handles the allocation of resources and the spawning of the compute tasks. Swift supports e. g. Globus, Condor and PBS resources. By default, Swift uses a 1:1 mapping for CUs and SUs. However, Swift supports the grouping of SUs as well as PJs. For the PJ functionality, Swift uses the Coaster [22] framework. Coaster relies on a master/worker coordination model; communication is implemented using GSI-secured TCP sockets. Swift and Coaster support various scheduling mechanisms, e. g. a FIFO and a load-aware scheduler. Additionally, Swift can be used in conjunction with Falkon [11], which also provides Pilot-like functionality.

3.5 Discussion

P* provides an abstract model for describing and understanding PJ frameworks. Table 1 summarizes how P* can be applied to BigJob, DIANE, Condor-G/Glide-in and Swift/-Coaster. While each of the frameworks maintains its own vocabulary, all share the common P* elements. Table 2 summarizes the P* characteristic and other properties of these frameworks. While most of these frameworks share many properties, such as the M/W coordination model, they differ in characteristics, such as the communication model or scheduling.

As we will see further in §V (Experiments), in spite of a common Pilot-API each PJ framework has a rather different usage modality; this is reflective of the fact that typically, PJ frameworks "evolve in" and are "native to" specific infrastructure. Native infrastructure refers to the infrastructure for which a PJ framework has been primarily developed for and on which infrastructures it is mainly used, e.g., Condor-G/Glide-in is the native PJ framework of the Open Science Grid and its use is heavily coupled with WMS; that does not mean, however, that these frameworks do not work on other infrastructures.

4. PILOT-API: A UNIFORM API TO HETEROGENEOUS PJ FRAMEWORKS

In the previous two sections we presented successively the P* Model and existing Pilot-Job frameworks. Before we present the Pilot-API – which provides an abstract interface to Pilot-Job frameworks that adhere to the P* Model, we will motivate the need for such an API.

4.1 Motivation

At a high-level, there exist two approaches towards interoperability: (i) deep integration of systems (system level interoperability), and (ii) the use abstract interfaces (application level interoperability). Approach (ii) requires a certain level of semantic harmonization between the systems, and is (in principle and technically) hard to achieve post-facto, even if the respective systems inherently implement the same abstract model (here: the P* Model). While interoperation via an abstract interface (here: Pilot-API) is a semantically weaker approach than (i), it does allow for interoperability with minimal (application level) effort [14, 23].

The current state of workflow (WF) systems [24, 25] provides a motivating example for the P* Model and the Pilot-API: even though many WF systems exist (with significant duplicated effort), they provide limited means for extensibility and interoperability. We are not naive enough to suggest a single reason, but assert that one important contributing fact is the lack of the right interface abstractions upon which to construct workflow systems; had those been available, many/most WF engines would have likely utilized them (or parts thereof), instead of proprietary solutions. Significant effort has been invested towards WF interoperability at different levels – if nothing else, providing post-facto justification of its importance. The impact of missing interface abstractions on the WF world can be seen through the consequences of their absence: WF interoperability remains difficult if not infeasible. The Pilot-API in conjunction with the P* Model aims to prevent similar situation for Pilot-Jobs.

We appreciate the difficulty balancing the level of semantic expressivity for API's designed to work on multiple semantically heterogeneous systems: defining the API as 'smallest common denominator' is often too simplifying and misses large numbers of 'edge' use cases; defining the API as 'greatest common factor' clutters the API with non-portable semantics and making the API too complex [26]. The Pilot-API uses the Pareto principle as a guideline for a balanced abstraction level.

Properties	**BigJob**	**DIANE**	**Condor-G/Glide-in**	**Swift/Coaster**
Coordination	Master/Worker	Master/Worker	Master/Worker	Master/Worker
Communication	Advert Service	CORBA	TCP	GSI-enabled TCP
Scheduling Binding	FIFO, custom Early/Late	FIFO, custom Late	Matchmaking, priority-based scheduler Late	Load-aware scheduler, CU grouping Late
Agent Submission	API	GANGA Submission Script	Condor CLI	Resource Provider API
End User Environment	API	API and Master/Worker Framework	CLI Tools	Swift script
Fault Tolerance	Error propagation	Error propagation, retries	Error propagation, retries	Error propagation, retries, replication
Resource Abstraction	SAGA	GANGA/SAGA	Globus	Resource Provider API/Globus CoG
Security	Multiple (GSI, Advert DB Login)	Multiple (GSI)	Multiple (GSI, Kerberos)	GSI

Table 2: P* Characteristics and Properties of Different Pilot-Job Frameworks: The properties in bold-face correspond to the P* characteristics; other items are general properties. The PJ frameworks share many P* characteristics and properties, e. g. the common usage of the M/W scheme or of a resource abstraction layer. However, they also differ in aspects, such as the coordination model or the used communication framework.

4.2 Understanding the Pilot-API

The Pilot-API[1] supports two different usage modes (i) it provides a unified API to various PJ frameworks (e. g. BigJob , DIANE and Condor-G/Glide-in), and (ii) it enables the concurrent usage of multiple PJ implementations.

The Pilot-API classes and interactions are designed to reflect the P* elements and characteristics. The API exposes the primary functionality of the Pilot-Manager using two classes: the `PilotComputeService` for the management of Pilots and the `ComputeUnitService` for the management of CUs. As defined by P*, a CU represents a primary self-containing piece of work that is submitted through the Pilot-API.

Figure 4 shows the interactions between the P* elements. The Pilot-API decouples workload management and resource scheduling by exposing two separate services: The `PilotComputeService` and `ComputeUnitService`. The `PilotComputeService` serves as a factory for instantiating Pilots. Also, the `PilotComputeService` can be used to query for currently active `PilotCompute` instances. A `PilotCompute` instance is returned as result of the `create_pilot()` method of the `PilotComputeService` (step 1). Analogous to job creation in SAGA, the instantiation of the `PilotCompute` instance is done by using a `PilotComputeDescription`. The description can be reused and has no state, while the `PilotCompute` instance has state and is a reference for further usage.

```
pcs = PilotComputeService()
pc_desc = PilotComputeDescription()
pc_desc.total_core_count = 8
pc = pcs.create_pilot('gram://queenbee',
                          pc_desc, 'bigjob')
```

Listing 1: Instantiation of a Pilot Service using a Pilot Compute Description.

The `PilotCompute` object represents a Pilot instance and allows the application to interact with it, e. g. to query its state or to cancel it. The process of `PilotCompute` creation is depicted in step 1-2 of figure 4 and in listing 1.

Listing 2 shows the creation of a `ComputeUnitService`. Having created a `ComputeUnitService` instance, `PilotComputeService` instances can be added and removed at any time. This enables applications to respond to dynamic resource requirements at runtime, i. e. additional resources can be requested on peak demands, and can be released if they are no longer required.

```
cus = ComputeUnitService()
cus.add(pcs)
```

Listing 2: Instantiation of a `ComputeUnitService` using a reference to the `PilotComputeService`.

[1]We use Python syntax for describing the Pilot-API.

```
cud = ComputeUnitDescription()
cud.executable = '/bin/bfast'
cud.arguments = ['match', '-t4', '/data/file1']
cud.total_core_count = 4
cu = cus.submit(cud)
```

Listing 3: Instantiation and submission of a `ComputeUnitDescription`.

The `ComputeUnitService` is responsible for managing the execution of CUs. Regardless of the state of the `PilotComputeService`, applications can submit CUs to a `ComputeUnitService` at anytime (listing 3 and step 3 in figure 4). Once the `ComputeUnitService` becomes responsible for a CU, the CU transitions to an SU. SUs are internally processed (e. g. they can be aggregated) and are then scheduled to the Pilot-Job Framework (step 4). The PJ framework is responsible for the actual execution of the SU on a resource. Note that multiple levels of (hierarchical) scheduling can be present, commonly a SU is scheduled inside a PJ framework and the model allows it to be present in multiple layers.

Each `ComputeUnit` and `PilotCompute` object is associated with a state. The state model is based on the SAGA job state model [16]. Applications can query the state using the `get_state()` method or they can subscribe to state update notifications (by setting callbacks).

Finally, an application can have any number of `PilotComputeService` or `ComputeUnitService` instances. Multiple `PilotComputeService` instances can be associated to a `ComputeUnitService`, and a `PilotComputeService` can be associated to multiple `ComputeUnitService` instances. A `ComputeUnitService` can manage multiple `ComputeUnit` instances, but a `ComputeUnit` can only be managed by one `ComputeUnitService`. So can a `PilotComputeService` manage multiple `PilotCompute` instances, but can a `PilotCompute` only be managed by one `PilotComputeService`.

5. EXPERIMENTS AND RESULTS

In this section we analyze the performance and scalability of different PJ frameworks. It is important to note that our experiments do not try to identify the "fastest" PJ framework, as this is dependent on different factors, e. g. particularly the infrastructure used – the experiments are conducted on different production (XSEDE, EGI, OSG) and

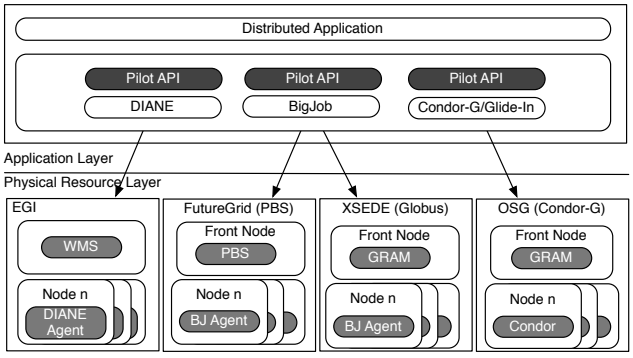

Figure 3: Pilot-API and PJ frameworks: The Pilot-API provides a unified interface to utilize the native Pilot-Job capabilities of different infrastructures, e. g. BigJob for XSEDE/FutureGrid resources, DIANE for EGI and Condor for OSG resources.

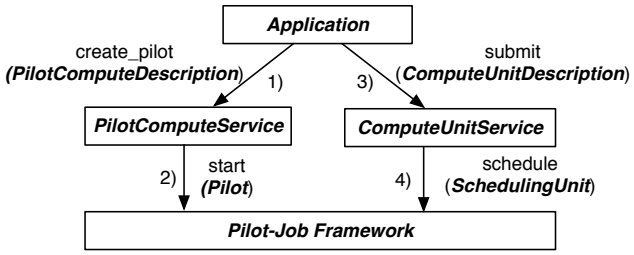

Figure 4: Control Flow Pilot-API and PJ Frameworks: The functionality of pilot-jobs are exposed using two primary classes: The `PilotComputeService` for the management of Pilots, and the `ComputeUnitService` for the management of CUs.

research (FutureGrid) infrastructures.

As discussed in §3, the investigated PJ frameworks can be mapped to the P* Model, which enables their collective usage via the Pilot-API. In section 5.1 we investigate the performance and scalability of coordination in BigJob and DIANE, executing thousands of tasks but with zero workload. We establish that for up to thousands of tasks, existing coordination infrastructures suffice. In section 5.2 we show the effectiveness of our approach (Pilot-API/P* Model) by executing *real application workloads* – in this case a genome alignment application – on multiple distinct production infrastructures. Further, we demonstrate interoperability by using multiple PJ frameworks concurrently on multiple infrastructures using the Pilot-API – see section 5.3.

5.1 Coordination in PJ Frameworks

Different PJ implementation follow different design objectives – that also reflects in the designs of their communication & coordination (c&c) sub-systems. The primary barrier for performance and scalability is not the CU submission, but the internal coordination of the elements of a PJ implementation. There are many factors that influence the overall performance, e. g. the degree of distribution (local (LAN) vs. remote (WAN)), the communication pattern (1:n versus n:n) and the communication frequency. In the following we investigate the impact of different c&c related factors on the overall performance and scalability of PJ systems. We evaluate different BigJob configurations, and compare and contrast them with DIANE.

BigJob's c&c was originally based on a shared, centralized data space, the SAGA Advert Service (AS) [27]. The communication between all components is done via this shared data space, which decouples BJ-Manager and BJ-Agent and allows both entities to operate at their own pace, optimizing the overall throughput. A particular issue during distributed runs is the latency between the application and the (remote) Advert Service. Another challenge is that this design introduces a potential single-point-of-failure and scalability bottleneck if the centralized data space is not carefully designed and operated.

BigJob also provides two alternative c&c sub-systems: Redis [17] and ZeroMQ [18]. Redis is a lightweight key/value store. Redis is used in a similar way as the AS, i. e. all communication between BJ-Agent and BJ-Manager is channeled through it. In contrast, the ZeroMQ sub-system utilizes a client-server architecture, which is similar to the CORBA-based approach of DIANE – in this architecture, the PM

maintains the overall state. Clients connect to the PM to request new SUs or to report state updates. An advantage of this architecture is that it does not require a separate infrastructure deployment for the c&c sub-system. Both the data space and the client-server c&c sub-systems can be combined with a publish/subscribe mechanism, i. e. instead of polling, an agent can receive notifications when a new SU arrives.

For our c&c evaluation, we conducted several experiments on FutureGrid [28]. We executed a different number of very short running (i. e. zero workload) CUs on Alamo and Sierra, running up to 128 CUs concurrently. This enables us to focus on the overhead induced by the PJ framework and the c&c subsystem. Each experiment is repeated at least 10 times.

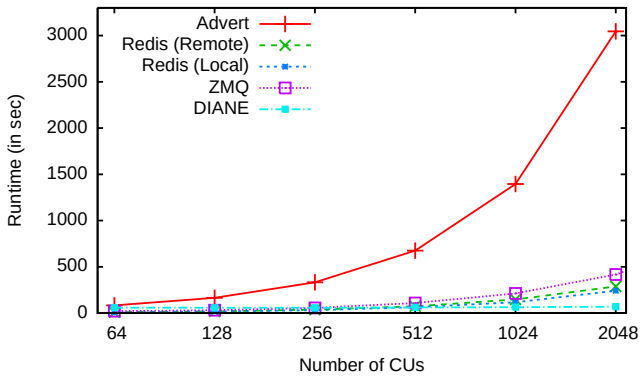

Figure 5: BigJob and DIANE Performance (1): The runtime increases linearly with the number of CUs in most cases. The overhead per CU imposed by the AS is generally higher than for other backends. The runtime of the DIANE CUs only increases moderately mainly due to the aggregation of the CUs.

Figure 5 illustrates the performance and scalability of different BJ configurations and DIANE with respect to the number of CUs. Clearly, the c&c sub-system has a great impact on the overall performance: the Redis backend shows the best performance for small CU counts. The difference between local and remote coordination is moderate (about 20 %). While ZeroMQ is very fast and lightweight, it requires a careful implementation, in particular concerning synchronization and throughput optimization. The overall performance is slightly worse than for Redis. The Advert Service currently has some limitations which will be discussed later. DIANE shows a higher startup overhead, which is particularly observable for smaller CU numbers. However, the runtime increases only slightly (about 10 sec) when going from 64 to 2048 CUs. One reason is that DIANE aggregates SUs: for 2048 CUs only a single task description is created, which the framework efficiently distributes to its agents. BigJob in contrast maintains a separate description for each CU.

Figure 6 illustrates the performance scalability with respect to the number of cores per Pilot. For this purpose, we execute 4 CUs per core, i. e. between 32 and 512 CUs. In particular, the Redis (Local) configuration show an almost linear scalability up to 128 cores. The Redis (Remote) setup again imposes some overhead (about 14 %). ZeroMQ performs very well with lower core counts; with larger core counts the runtimes increase, indicating a potential scalability bottleneck. DIANE shows, in particular for lower core counts, a longer runtime, again due to the higher startup overhead. With higher core counts DIANE behaves similar to BigJob/ZeroMQ, showing a greater increase of the overall runtime. This increase can likely be attributed to the single central manager in the client-server architecture. As in the last experiment, the Advert c&c sub-system showed a significantly lower performance.

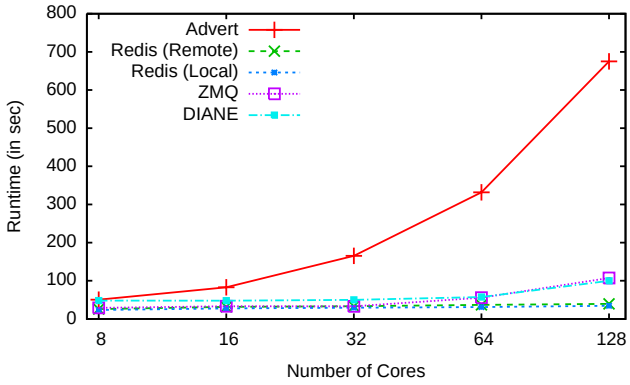

Figure 6: BigJob and DIANE Performance (2): The runtime of a workload of 4 CUs per core, i. e. 32 - 512 CUs. For BJ-Redis the runtime increases only moderately, the client-server-based implementations BJ-ZMQ and DIANE show particularly a steep increase when going from 64 to 128 cores. The Advert backend is currently unoptimized and shows a significant slowdown.

The AS implementation currently shows some performance limitations, mainly caused by the prototypical nature of its implementation. Further, it must be noted that the Advert Service was deployed remotely (mainly due to deployment constraints). Even so, the discrepancy between AS and Redis (Remote), which has been deployed on the same remote network, is significant. A reason for that is the used remote access protocol, which is basically the native remote PostgreSQL database protocol, which is not optimized for WAN connections. Transactions e. g. are very latency sensitive and require several roundtrips. Further, the API is based on a hierarchical namespace, which does not easily map to relational databases – in deeply nested namespaces exhibit an insufficient query and update performance. In contrast, data is stored in memory for Redis and ZeroMQ, which partially explains the significant performance gains.

5.2 Characterizing PJ Frameworks on DCI

Production Infrastructure

To validate the abstractions developed, such as Pilot-API, we conducted a series of experiments on various production infrastructure. We executed BFAST [29] using three different PJ frameworks (BigJob, DIANE and Condor) on XSEDE [30], FutureGrid [28], EGI [21] and OSG [31].

Specifically, we utilized the following resources: XSEDE: Kraken (1.17 PFlop Cray XT5 / 9,408 nodes / 112,896 cores / Torque) and QueenBee (Linux Cluster / 668 nodes / 5,344 cores / PBS); FutureGrid: India (Linux Cluster / 108 nodes

/ 864 cores / PBS); European Grid Initiative (EGI): Resource federation of 364,500 cores; OSG: Condor pool (via the *Engage* VO, Glide-in WMS, 20,000 Glide-ins).[2]

Experimental Configuration

We define an experimental configurations as a set comprised of the production infrastructure used, the specific machine/resource configuration, and the utilized PJ framework. We run the experiment on four different configurations: (B1) BigJob/XSEDE, (B2) BigJob/FutureGrid, (B3) Condor/OSG, (B4) DIANE/EGI. As discussed in §4, the Pilot-API provides a unified way for accessing these infrastructures using the native PJ capabilities for all three PJ frameworks, i. e. BigJob, DIANE and Condor (see figure 3). For BJ we use the PBS/SSH plugin to access India/FutureGrid, and the SAGA-Torque adaptor to access Kraken/XSEDE. On OSG, we use SAGA and the SAGA-Condor adaptor to interface directly with OSG's dynamic Glide-in-WMS resource pool. Further, we utilize DIANE on EGI.

The investigated workload consists of of 128 CUs. Each CU executes a BFAST matching process, which is used to find potential DNA sequence alignments. The scenario requires about 7 GB input data: 1.7 GB for the reference genome and index files, and 5.4 GB *short read* files, generated by a DNA sequencing machine (32 files, 1.68 GB each).

In total, 128 CUs are defined and submitted to the `ComputeUnitService`. Each BFAST CU requires 1 core; depending on the resource a different number of cores maybe assigned for each BFAST CU, e. g. on Kraken 4 cores are reserved for each CU to provide sufficient IO and memory. Each CU is associated with a set of input files which is pre-staged in configuration B1 and B2. For the HTC infrastructures EGI and OSG (B3 and B4), these file are transferred by DIANE respectively Condor before running each CU.

Experimental Results

Figure 7 shows the results of the experiments. In addition to the runtime (T_r) we measured the queuing time (T_q), the time to transfer input files (T_s), and the actual runtime of the BFAST CU T_c. T_q includes both pilot-external, i. e. queuing system (PBS, Torque, Condor) waiting times, as well as Pilot-internal waiting times.

Generally, the performance of BFAST is heavily dependent on the available I/O bandwidth. Both the index and read files (7 GB in this scenario) need to be loaded into memory. Kraken and India use shared network filesystems (Lustre for Kraken, NFS for India), which are utilized by all jobs running on these machines – the collective performance of multiple BFAST CU thus degrades significantly on larger machines, such as Kraken, where a potentially large number of jobs access the filesystem concurrently (the runtime on Kraken is about 30 % slower than on India). On EGI and OSG, the BFAST CU performs best: on OSG about two times faster than on Kraken. This can mainly be attributed to the use of local storage.

Another factor is T_q. In B1 and B2, T_q is very low and mainly caused by BJ internal sub-job queueing. In B3, T_q is primarily caused by the launch of the 128 worker agents – each of these agents needs to be submitted and started via the gLite WMS; on each node, a DIANE worker agent must be downloaded, installed and started. In total, T_q for each CU was ∼50 min (in cmp. T_q for BJ on India was ∼9 min). Thus, even if T_s is not considered, T_r is ∼2.3 times longer on EGI (DIANE) than on FutureGrid (BJ). The unpredictability of T_q also contributes to the higher deviation in T_r. On OSG, T_q is ∼12 minutes, comparable to T_q on XSEDE::Kraken.

In B1 and B2, no file staging is used. On EGI and OSG resources, however, no shared filesystem is available; thus, files need to be transferred to the executing system for B3 and B4. For each CU, the reference genome, the index files and one read file need to be staged (3.38 GB) – which consumes more than 50 % of the overall runtime. Thus, the runtime on these two infrastructures is about 4-5 times longer than on Kraken and India. If file staging is not considered, OSG shows the best performance.

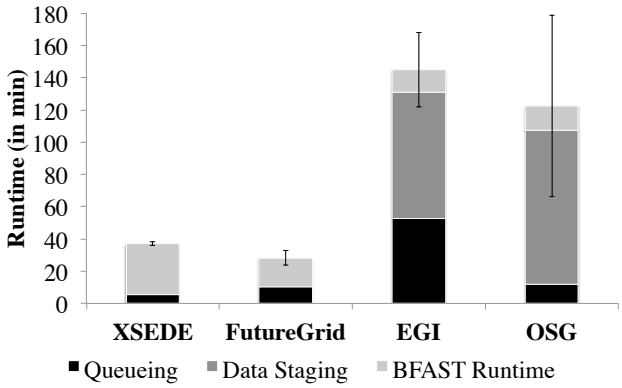

Figure 7: PJ Framework Performance on XSEDE, FutureGrid, EGI and OSG: Running 128 BFAST match tasks on 128 cores. Each experiment is repeated at least 3 times. The longer runtimes on EGI and OSG are mainly caused by the longer queuing times. Additionally, both infrastructure require the staging of all input files.

5.3 PJ Framework Interoperability

In principal, two different kinds of interoperability between Pilot-Jobs and infrastructure exist: the first is the usage of BJ in conjunction with different SAGA adaptors and infrastructures, and the second is PJ framework interoperability, i. e. the usage of different PJ frameworks via the Pilot-API. In configuration C1 we utilize SAGA-based interoperability by running BigJob concurrently on FutureGrid::India and XSEDE::Kraken. C2 and C3 show PJ framework interoperability by concurrently running BigJob and DIANE on FutureGrid::India and EGI as well as Condor and BigJob on OSG and XSEDE::QueenBee. For all scenarios, we run the same BFAST application described in §5.2 with 64 CUs on each infrastructure, i. e. in total 128 CUs.

Figure 8 shows the results of the interoperability tests. In C1, one Pilot is submitted to Kraken and one to India. Files are pre-staged before the run. Needless to say, the overall runtime is determined by the slower resources. Consistent

[2]Glide-in WMS is a software system built on top of Condor-G/Glide-in, which can, based on the current and expected number of jobs in the pool, automatically increase or decrease the number of active Glide-ins (Pilots) available to the pool. While the average number of active Glide-ins on OSG is around 20,000, more than 40,000 Glide-ins can be active and available for job execution during peak load.

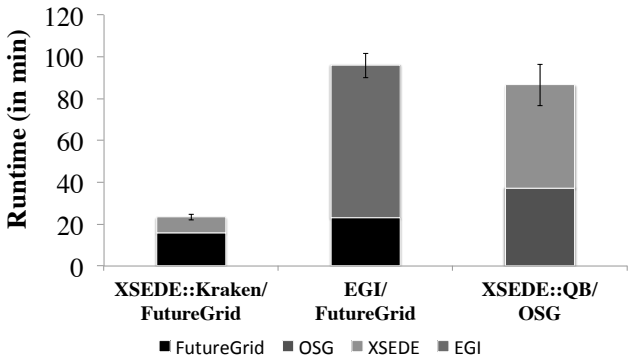

Figure 8: PJ Framework Interoperability: Runtime of 128 BFAST CUs on different infrastructures. CUs are equally distributed across the two infrastructures. In particular, slower resources, such as Kraken can benefit from offloading CUs to faster infrastructures.

with previous results the Pilot on India finished before the Pilot on Kraken. However, T_r of the distributed run improved about 6 % compared to a Kraken only run mainly due to the usage of the faster India resource for half of the CUs.

The middle bar in figure 8 shows C2 and demonstrates that two PJ frameworks can be utilized concurrently using the Pilot-API. Files are pre-staged for FutureGrid; they need to be staged for the DIANE/EGI CUs. The performance in this scenario is significantly better than in the DIANE only case for 128 CUs (see section 5.2), mainly due to the fact the overhead induced by file staging is only applicable to half of the CUs. However, it must also be noted that the performance of the BJ Pilot is about 30 % worse for the EGI/FG run compared to the XSEDE/FG run. This can be mainly attributed to the distributed coordination necessary in a high latency environment. All communication is conducted via a Redis instance deployed on FG, while the BJ manager is deployed on EGI; thus, for each CU several cross-atlantic roundtrips are necessary.

Finally, C3 (right bar in figure 8) shows the result of the OSG and XSEDE::QB run. Since QueenBee is an older XSEDE machine, T_r on this machine is much longer than T_r on OSG despite the necessity of file staging.

In summary, the Pilot-API enables the usage of a diverse set of distributed, heterogeneous infrastructures via a unified API. As shown, the Pilot-API does not represent a barrier to scalability, but provides the ability to overcome scalability limitations of certain infrastructure caused e. g. by IO, memory, and/or bandwidth bottlenecks. By using the capabilities of the Pilot-API, i. e. the decoupling of workload submission from resource assignment, applications can scale-out on multiple and possibly heterogeneous resources in a flexible way. While scale-out introduces some overheads (e. g. in particular for data-intensive applications, large amounts data need to be moved), it enables application to improve performance, e. g. by offloading parts of the computation to a faster resource (see C1), and to scale its problem set, e. g. by utilizing more resources at the same time.

6. P* AS A MODEL FOR PILOT-DATA

Many scientific applications have immense data requirements, which are projected to increase dramatically in the near future [32]. The small genome alignment tool scenario presented above e. g. operates on a input data set of > 7 GB. While Pilot-Jobs efficiently support late-binding of Compute Units and resources, the management of data in distributed systems remains a challenge due to various reasons: (i) the placement of data is often decoupled from the placement of Compute Units and Pilots, i. e. the application must often manually stage in and out its data using simple scripts; (ii) heterogeneity, e. g. with respect to storage, filesystem types and paths, often prohibits or at least complicates late binding decisions; (iii) higher-level abstraction that allow applications to specify their data dependencies on an abstract, logical level (rather than on file basis) are not available; (iv) due to lack of a common treatment for compute and data, optimizations of data/compute placements are often not possible. For example, in scenario (B3) and (B4) presented in section 5.2, even though the 50 % of the input data set is shared for all CUs, the complete data is staged for each CU.

In addition, applications must cope with various other challenging, data-related issues, e. g. varying data sources (such as sensors and/or other application components), fluctuating data rates, transfer failures, optimizations for different queries, data-/compute co-location etc. While these issues can be in principal handled in an application-specific way, the usage of higher-level abstractions, such as a common Pilot-based abstraction for compute and data is preferable.

This motivates an analogous abstraction that we call *Pilot-Data (PD)*. PD provides late-binding capabilities for data by separating the allocation of physical storage and application-level data units. Further, it provides an abstraction for expressing and managing relationships between data units and/or work units. These relationships are referred to as *affinities*.

P Model Elements for Data*

The elements defined by P* (in section 2.2) can be extended by the following elements:

- **Pilot (Pilot-Data):** A Pilot-Data (PD) functions as a placeholder object that reserves the space for data units. PD facilitates the late-binding of data and resource and is equivalent to the Pilot in the compute model.
- **Data Unit (DU):** DU is the base unit of data assigned by the application, e. g. a data file or chunk. Multiple DUs can be aggregated within a Data Unit Set.
- **Scheduling Unit (SU):** is an internal unit of scheduling (as in the compute case). The Pilot framework can aggregate or split DUs into one or more SUs.
- The **Pilot-Manager (PM)** is the same as in the compute model and implements the different characteristics of the P* Model. It is responsible for managing DUs and SUs. Data is submitted to the framework via the PM. The PM which is responsible for mapping DUs to SUs and for conducting decision regarding resource assignments. SUs are placed on physical resources via the Pilot.

Note, each element can be mapped to an element in the P* Model by symmetry, e. g., a DU correspond to a CU in the original P* Model; a PD is a placeholder reserving a certain amount of storage on a physical resource and corresponds

to the Pilot in the P* Model.

P Model Characteristics for Data*

While the extended P* Model introduces new elements, the characteristics however, remain the same to a great extent. The coordination characteristic describes how the elements of PD interact, e. g. utilizing the M/W model; the communication characteristic can be applied similarly. The scheduling characteristics must be extended to not only meet compute requirements, but also to support common data access patterns. Data and compute placement decisions are made by the scheduler based on defined policies, affinities & dynamic resource information.

Pilot-API for Data

Analogous to the Pilot-API for Compute, the Pilot Data API [33] defines the `PilotDataService` entity as an abstraction for creating and managing pools of storage. A `PilotData` instance represents the actual physical storage space. The `ComputeDataService` entity functions as an application-level scheduler, which accepts both `ComputeUnits` and `DataUnits`. It resolves necessary dependencies (e.g. data/data or data/computer affinities), and is responsible for managing the execution of DUs and CUs.

7. DISCUSSION AND FUTURE WORK

The primary intellectual contribution of this work has been the development of the P* Model, the mapping of P* elements to PJ frameworks such as DIANE and Condor-G/Glide-in and the design and development of the Pilot-API – that reflects the P* elements and characteristics.

The P* Model provides a common abstract model for describing and characterizing Pilot-abstractions. We validate the P* Model by demonstrating that the most widely used PJ frameworks, viz., DIANE and Condor-G/Glide-in can be compared, contrasted and analyzed using this analytical framework. Furthermore we demonstrate the use of the Pilot-API – which provides a common access layer to different PJ frameworks, with multiple PJ frameworks over distributed production cyberinfrastructure, such as XSEDE, OSG, EGI and FutureGrid. The Pilot-API also enables the concurrent use of multiple PJ frameworks, thus providing interoperability and extensibility. Although the aim of our experiments is the demonstration of the interoperable use of hitherto distinct and disjoint Pilot-Jobs, in the process we highlight the performance advantages that can emanate from the ability to seamlessly distribute (I/O intensive) workloads in a scalable manner.

P* provides significant future development, research & deployment opportunities. We discuss one opportunity along each of these axes: (i) Development and Extension: In order to overcome limitations of the API-based approach we will develop a complete implementation of the P* Model – called TROY (Tiered Resource Overlay). TROY will provide deeper integration of the Pilot-API via a semantic mapping of the P* elements and characteristics to Pilot-Job frameworks. TROY will also provide a unified – compute and data, implementation of P*, e.g., BigData analogous to BigJob; (ii) Research Issues: These developments and extensions will also provide the basis to explore and reason on the relative roles of system versus application-level scheduling, heuristics for dynamic execution and the role of affinity; (iii) Deployment and Uptake: The Pilot-API and TROY are/will be designed to support production scale science on production infrastructure (as demonstrated by the emphasis on production infrastructure in our experimentation). Our experience with fragile grid middleware and infrastructures has taught us to appreciate the significance of the ease of deployment and the need for robustness of tools and software on heterogeneous distributed infrastructures. TROY – which embodies the P* Model, is being designed with usability, reliability and robustness as first-class considerations.

Our fundamental research has immense practical implications and potential: Although current PJ frameworks collectively support millions of tasks yearly on several production distributed infrastructure, extensibility and interoperability remain significant challenges [34]. Given the increasing importance of Pilot-Jobs and the challenges associated with distributed data placement, our work which provides both a conceptual model and practical solutions has the potential to improve this situation. In fact, it is a stated goal of our research to enhance the range of applications and application usage-modes that will benefit from the Pilot abstraction, by deeply integrating Pilot-API/P* and TROY capabilities with multiple production infrastructures [35].

Acknowledgements

This work is funded by NSF CHE-1125332 (Cyber-enabled Discovery and Innovation), HPCOPS NSF-OCI 0710874 award, NSF-ExTENCI (OCI-1007115) and NIH Grant Number P20RR016456 from the NIH National Center For Research Resources. Important funding for SAGA has been provided by the UK EPSRC grant number GR/D0766171/1 (via OMII-UK) and the Cybertools project (PI Jha) NSF/LEQSF (2007-10)-CyberRII-01. SJ acknowledges the e-Science Institute, Edinburgh for supporting the research theme. "Distributed Programming Abstractions" & 3DPAS. MS is sponsored by the program of BiG Grid, the Dutch e-Science Grid, which is financially supported by the Netherlands Organisation for Scientific Research, NWO. SJ acknowledges useful related discussions with Jon Weissman (Minnesota) and Dan Katz (Chicago). We thank J Kim (CCT) for assistance with BFAST. This work has also been made possible thanks to computer resources provided by TeraGrid TRAC award TG-MCB090174 (Jha) and BiG Grid. This document was developed with support from the US NSF under Grant No. 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed".

8. REFERENCES

[1] S. J. et.al., "Critical Perspectives on Large-Scale Distributed Applications and Production Grids (Best Paper Award)," in *The 10th IEEE/ACM Conference on Grid Computing 2009*, 2009, pp. 1–8. [Online]. Available: "http://www.cct.lsu.edu/~sjha/dpa_publications/dpa_grid2009.pdf"

[2] S.-H. Ko, N. Kim, J. Kim, A. Thota, and S. Jha, "Efficient runtime environment for coupled multi-physics simulations: Dynamic resource allocation and load-balancing," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 349–358.

[3] J. Kim, W. Huang, S. Maddineni, F. Aboul-Ela, and S. Jha, "Exploring the RNA folding energy landscape using scalable distributed cyberinfrastructure," in *Emerging Computational Methods in the Life Sciences, Proceedings of HPDC*, 2010, pp. 477–488.

[4] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A Computation Management Agent for Multi-Institutional Grids," *Cluster Computing*, vol. 5, no. 3, pp. 237–246, July 2002.

[5] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011, emerging Programming Paradigms for Large-Scale Scientific Computing.

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

[7] A. Casajus, R. Graciani, S. Paterson, A. Tsaregorodtsev, and the Lhcb Dirac Team, "Dirac pilot framework and the dirac workload management system," *Journal of Physics: Conference Series*, vol. 219, no. 6, p. 062049, 2010.

[8] P.-H. Chiu and M. Potekhin, "Pilot factory – a condor-based system for scalable pilot job generation in the panda wms framework," *Journal of Physics: Conference Series*, vol. 219, no. 6, p. 062041, 2010.

[9] "Topos - a token pool server for pilot jobs," https://grid.sara.nl/wiki/index.php/Using_the_Grid/ToPoS, 2011.

[10] R. Buyya, D. Abramson, and J. Giddy, "Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid," *International Conference on High-Performance Computing in the Asia-Pacific Region*, vol. 1, pp. 283–289, 2000.

[11] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: A Fast and Light-Weight TasK ExecutiON Framework," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[12] E. Walker, J. Gardner, V. Litvin, and E. Turner, "Creating personal adaptive clusters for managing scientific jobs in a distributed computing environment," in *Challenges of Large Applications in Distributed Environments, 2006 IEEE*, 0-0 2006, pp. 95–103.

[13] "SAGA BigJob," http://faust.cct.lsu.edu/trac/bigjob, 2011.

[14] A. Luckow, L. Lacinski, and S. Jha, "SAGA BigJob: An Extensible and Interoperable Pilot-Job Abstraction for Distributed Applications and Systems," in *The 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2010, pp. 135–144.

[15] "The SAGA Project," http://www.saga-project.org.

[16] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, A. Merzky, J. Shalf, and C. Smith, "A Simple API for Grid Applications (SAGA)," Open Grid Forum," OGF Recommendation Document, 2007.

[17] Redis, http://redis.io/, 2011.

[18] ZeroMQ, http://www.zeromq.org/, 2011.

[19] J. M. *et al*, "Ganga: A tool for computational-task management and easy access to grid resources," *Computer Physics Communications*, vol. 180, no. 11, pp. 2303 – 2316, 2009.

[20] *Common Object Request Broker Architecture: Core Specification*, Object Management Group, M 2004.

[21] EGI, http://www.egi.eu/, 2011.

[22] "Coasters," http://wiki.cogkit.org/wiki/Coasters, 2009.

[23] S. Jha, H. Kaiser, A. Merzky, and O. Weidner, "Grid Interoperability at the Application Level Using SAGA," in *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 584–591.

[24] "Final Report of NSF Workshop on Challenges of Scientific Workflows," 2006, http://www.isi.edu/nsfâĂŘworkflows06.

[25] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, *Workflows for e-Science: Scientific Workflows for Grids*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[26] Joel Spolsky, "The Law of Leaky Abstractions," http://www.joelonsoftware.com/articles/LeakyAbstractions.html.

[27] A. Merzky, "SAGA API Extension: Advert API," OGF Document Series 177, http://www.gridforum.org/documents/GFD.177.pdf, 2011.

[28] "FutureGrid: An Experimental, High-Performance Grid Test-bed," https://portal.futuregrid.org/, 2012.

[29] N. Homer, B. Merriman, and S. F. Nelson, "BFAST : An alignment tool for large scale genome resequencing," *PLoS One*, vol. 4, no. 11, p. e7767, 2009.

[30] "XSEDE: Extreme Science and Engineering Discovery Environment," https://www.xsede.org/, 2012.

[31] R. P. *et al*, "The open science grid," *Journal of Physics: Conference Series*, vol. 78, no. 1, p. 012057, 2007.

[32] T. Hey, S. Tansley, and K. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Redmond, Washington: Microsoft Research, 2009.

[33] Pilot API, https://github.com/drelu/BigJob/blob/master/pstar/api/.

[34] "Extenci: Extending science through enhanced national cyberinfrastructure," https://sites.google.com/site/extenci/.

[35] "How to Run BigJob on XSEDE," https://github.com/drelu/BigJob/wiki.