P*: An Extensible Model of Pilot-Abstractions for Dynamic Execution

Andre Luckow[1], Mark Santcroos[2,1], Sharath Maddineni[1], Andre Merzky[1], Shantenu Jha[3,1*]

[1]*Center for Computation & Technology, Louisiana State University, USA*

[2]*Bioinformatics Laboratory, Academic Medical Center, University of Amsterdam, The Netherlands*

[3] *Rutgers University, Piscataway, NJ 08854, USA*

[*]*Contact Author:* shantenu.jha@rutgers.edu

Abstract

Distributed cyberinfrastructures (CI) and applications require the ability to determine and utilize resource selection at runtime (dynamically), and not just before execution (statically). Pilot-Jobs have been notable in their ability to support dynamic resource utilization – in the number of applications that use them, in the scope of usage, as well as the number of CI that support them. In spite of broad uptake, there does not exist however, a well defined, unifying conceptual framework for pilot-jobs which can be used to define, compare and contrast different implementations. This paper is an attempt to (i) provide a minimal but complete model (P) of Pilot-Jobs, (ii) extend the basic model from compute to data, (iii) introduce TROY as an implementation of this model using SAGA, i. e. consistent with its API, job-model etc., (iv) establish the generality of the P* model by mapping various existing and well known pilot-jobs such as DIANE to P*, (v) establish and validate the implementation of TROY by concurrently using multiple distinct pilot-jobs.*

I. Introduction and Overview

Distributed CI almost by definition is comprised of a set of resources that is fluctuating – growing, shrinking, changing in load and capability. The ability to utilize such a dynamic resource pool is thus an important attribute of any application that needs to utilize distributed CI efficiently; this is in contrast to a static resource utilization model characteristic of parallel and cluster computing.

In addition, the evolution or internal dynamics of an application may vary, thereby changing the resource requirements. For example, different solvers, adaptive algorithms and/or implementations, can also require applications to utilize different set/amounts of resources. We will define dynamic applications to have the ability to respond to a fluctuating resource pool, i. e., the set of resources utilized at time (T), $T = 0$ is not the same as $T > 0$.

Multiple approaches exist to support dynamic resource utilization, for example, advanced scheduling (without preemption) provides essentially a guarantee of resources at a sufficiently far out time window. Another common approach for decoupling workload management and resource assignment/scheduling are *pilot-jobs (PJ)*. Along with empirical evidence, our experience suggests that distributed applications that are able to use tools, abstractions and services that break the coupling between workload management and resource assignment/scheduling have been more successful at efficiently utilizing distributed resources. In addition, the ability to provide these capabilities in user-space leads to the basic concept of a pilot-job.

Interestingly there exist many implementations of the PJ abstraction, wherein different projects and users have rolled-out their own. The fact that users have voted with their feet for PJs reinforces that the Pilot-Job is both a useful and correct abstraction for distributed CI; the fact that it has become an "unregulated cottage industry" reaffirms the lack of common nomenclature, integration, interoperability and extension.

Our work is motivated by the status of the usage and availability of the pilot abstraction vis-à-vis the current landscape of distributed applications and CI. To achieve our objectives, we attempt to provide a minimal, but complete model for the pilot abstraction, which can be used to provide an analytical framework to compare and contrast different pilot implementations. This is, to the best of our knowledge, the first such attempt.

A natural and logical extension of the PJ model, arising from the need to treat data as a first-class schedulable entity, is a concept analogous to PJ: the *pilot-data (PD)* abstraction. Given the consistent treatment of data and compute, as potentially equal components in a model to support dynamic resource and execution, we refer this model as the P* model ("P-star"). We present and discuss the P* model in §II.

In §III we introduce TROY – A Tiered Resource OverlaY – as an implementation of the P* model using the SAGA API. The specific pilot-job and pilot-data implementations in TROY are referred to as BigJob and BigData. As we will discuss, consistent with the goals and aims of SAGA, there can be multiple instances of BigJob for different backends. Thus, we posit that the TROY API can be used for most, if not all pilot-jobs. Before validating

this claim, in §IV of this paper, we analyze and discuss several well-known pilot-jobs (DIANE, Swift-Coaster and Condor Glide-in) using the analytical framework provided by the P* model.

In §V we validate the TROY framework by demonstrating how DIANE [16] – an existing and widely used pilot-job implementation, can be given a well defined API via the TROY API. To further substantiate the impact of TROY, we will demonstrate interoperability between different PJs by using the same API – a native SAGA based PJ, referred to as BigJob-SAGA, and DIANE. We believe this is also the first demonstration of interoperation of different pilot-job implementations.

II. P* Model: A Conceptual Framework for Pilot-Abstractions for Dynamic Execution

The uptake of distributed infrastructures by scientific applications has been limited by the availability of extensible, pervasive and simple-to-use abstractions which are required at multiple levels – development, deployment and execution stages of scientific applications. The PJ abstraction has been shown to be an effective abstraction to address specific requirements of distributed scientific applications. There are many implementations of the PJ abstraction. Although they are all for the most parts functionally equivalent – they support the decoupling of workload submission from resource assignment – it is often impossible to use them interoperably or even just to compare and contrast them.

Terms and Usage: It is important to emphasize that the terms abstraction, model, framework, and implementation are overloaded and often used inconsistently; thus we establish their usage in this paper. The pilot *abstractions* generalizes the reoccuring concept of utilizing a placeholder job as a container for a set of compute tasks. The P* *model* provides a specific, comprehensive description of pilot-job abstractions based on a set of identified elements and characteristics. The P* model can be used as a *conceptual framework*, for analyzing different implementations of the pilot abstraction. *TROY* is a specific implementation of the P* framework; it is also often referred to as *software framework* for pilot-jobs.

In an attempt to provide a common analytical framework using which most, if not all commonly used pilot-jobs can be understood, 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 properties that must be assigned and that determine the interaction of these elements and the overall functioning of any pilot-job that is described by the P* model. Further, we will show that these elements and interactions can be used to describe a pilot-data model.

A. Elements of the P* Model

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

- **Pilot-Job (PJ):** The PJ (or 'pilot') is the entity that actually gets submitted and scheduled onto a resource using the local resource management system. The pilot often has the role of an agent: it collects local information, manages the resources allocated to it and exchanges data (such as resource information, states) with the other elements of the PJ framework.
- **Work Unit (WU):** A work unit encapsulates a self-contained piece of work that is submitted to the PJ framework. The PJ framework in turn has flexibility to bind and schedule the WU, i. e. to determine when and how-many resources the WU will receive.
- **Scheduling Unit (SU):** SUs are the units of scheduling used inside the PJ framework. They get scheduled and executed on a resource via the pilot. An SU consists of one or more WUs that are combined inside the PJ framework. Note that that this can either be a simple grouping or a smart aggregation of WUs (e.g. two sets of parameters merged into one).
- **Pilot-Manager:** The PJ manager is responsible for coordinating the different components of the framework, i. e. the pilots, WUs and SUs. A PJ manager can generally manage multiple pilots.

The application itself is not strictly part of the core P* Model. The term application generally refers to the upper layer of the stack. The application utilizes the PJ framework to execute multiple instances of an application kernel (an ensemble) or alternatively instances of multiple different application kernels (a workflow). An application kernel is the actual binary that gets run. To execute an application kernel, an application must define a WU specifying the application kernel as well as other parameters. This WU can then be submitted to the PJ framework. Typically this is done by the PJ-Manager which is then responsible for binding one or more WUs to a SU and for scheduling the SU. As we will see in §IV, the above elements can be mapped to specific entities in many pilot-jobs in existence and use.

B. 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 characteristic describes how various elements of the P* Model are coordinated. Commonly, a distributed coordination mechanism such as master/worker (M/W) is utilized. A metric for describing coordination is the point and process of decision

making (e. g. about binding and scheduling of compute tasks): in the **central** model, information about available resources (aka pilots) are collected centrally, and decisions are centrally made by a manager process. In the **hierarchical** model the decision making process is divided up into a hierarchy of distributed agents. Each of them coordinates a defined aspect (e. g. a certain set of resources). In the **decentral** model, control is distributed among the different components. PJs with decentralized decision making often utilize autonomic agents that accepts respectively pull SUs according to a set of defined policies.

Communication: The communication characteristic describes the mechanisms for data exchange between the elements of the 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.

Binding: The binding characteristic defines how and at what time the assignment of a WU to a pilot is done. For example, a WU can be bound to a pilot either before the pilot has in turn been scheduled (early binding), whereas late binding can occur if the WU is bound after the pilot has been scheduled. Furthermore, the time of binding influences the point at which the binding decision is made, e. g. in the case of early binding, the decision could be made by the application, while in late binding mode the decision is often made by the PJ framework.

Scheduling: The scheduling characteristic describes the process of mapping a SU to physical resources via a pilot. Scheduling has a spatial component (which SU is executed on which pilot?) and a temporal component (at what time?). The different scheduling decisions that need to be made are representative of multi-level scheduling decisions that are often required in distributed environments. Commonly a set of (user-defined) policies (e.g. resource capabilities, data/compute affinities, etc.) or the implementation of a custom scheduler is supported.

Putting it all together: Figure 1 illustrates the interactions between the elements of the P* model. Although there can be variations, a typical usage scenario consists of the following steps: The application specifies the capabilities of the resources required (step 1). During the instantiation of a PJ and the assignment of resources to the PJ, pilots are queued and started via the local resource manager (step 1-4). Having initialized the PJ, WUs can be submitted to it (step 5).

Most PJs utilize a M/W coordination model, i. e. the manager tightly controls the actions of the worker (the agent). However, the integration of more decision capabilities into the agent can yield to a more autonomic and adaptive behavior as well as a better scalability. The application submits a WU to the PJ framework in most cases via the Manager component, which is then responsible for binding the WU to a SU and then scheduling the SU to a running pilot. In the simplest case one WU corresponds to one SU. The Manager then schedules a SU to specific PJ according to a specified policy or a user provided scheduler function (step 6). The SU gets executed to a physical resource on which the pilot is operating (step 7).

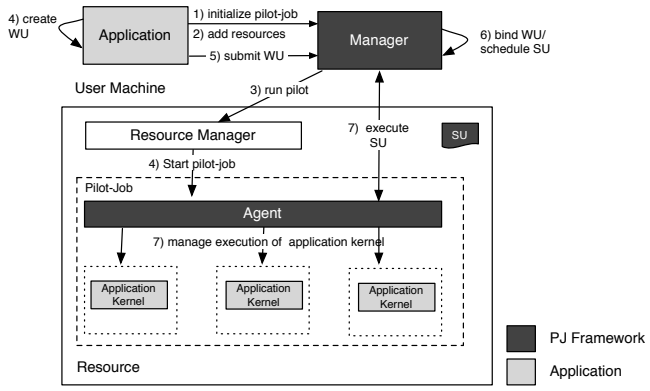

Fig. 1: **P* Model: Elements and Interactions:** The core of the model is the manager who is responsible for managing pilot-jobs and WUs. After a WU is assigned to the manager, the manager binds it to a SU and schedule the SU to an available resource.

C. Pilot-Data: Extension of P* to Dynamic Data

Dynamic execution is at least equally important for data-intensive applications: applications must cope with various challenging issues e. g. varying data sources (such as sensors and/or other application components), fluctuating data rates, optimizations for different queries, data-/compute co-location etc. Thus, having defined the P* model, we propose to extend it with data so as to facilitate an understanding of DE. This will motivate 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*.

Extension of P Model Elements:* A Pilot-Data Framework facilitates the late-binding between data units and physical storage resources, the so called pilot-stores. The elements defined by P* (in section II-B) can be extended by the following elements. By symmetry each element can be mapped to an element in the PJ model.

A. **Data Unit (DU):** DU is the base unit of data used by the framework, e. g. a data file or chunk.
B. **Pilot-Data (PD):** PD allows the logical grouping of DUs and the expression of data-data affinities. This collection of files can be associated with an extensible set of properties. One of these properties is affinity.
C. **Pilot-Store (PS):** A PS functions as a placeholder object that reserves the space for pilot-data objects. By

assigning a PD to a PS, the PD is bound to a physical resource. A PS facilitates the late-binding of data and resource and is equivalent to the pilot-job.

D. **Pilot-Data Manager** is analogous to the PJ-Manager responsible for managing DU, PD and PS elements.

In summary, a PD is a logical container that describes the properties of a group of DUs. A DU correspond to a WU in the PJ model. The PD object provides a higher level of semantics to the end-user for expressing relationships between DUs for which there is currently no equivalent in the PJ model. Users of a PJ must manage WU-level dependencies on application-level. Further, the PD model does not assume an internal scheduling unit. While there is some value in introducing a SU to PD, we assume that a PD is the unit of internal scheduling. Having instantiated a PD, it can be assigned to a PS. A PS is a placeholder reserving a certain amount of storage, i. e. it corresponds to a pilot in the pilot-job model. By associating a PD to a PS the data is actually moved to the physical location associated with the PS.

Extension of P Model Characteristics:* While the PD model introduces new elements, the characteristics remain to a great extent the same. The coordination characteristic describes how the elements of PD interact; the communication characteristic maps the identified communication patterns to the PD framework. The remaining two characteristics can be extended with respect to data: Binding describes the intelligent grouping and assignment of PDs and WUs to facilitate an optimal performance. The scheduling component particularly needs to consider affinities, i. e. user-defined relationships between PJs and/or PDs, e. g. data-data affinities exist if different DUs must be present at the same compute element; data-compute affinities arise if data and compute must be co-located for a computation, but their current location is different. The decision of where to place the data and compute is made by the scheduler based on the defined policies, affinities, available dynamic resource information.

PJ and PD encapsulate cross-cutting properties across data and computation. The P* implementation will optimize data- and computing according to the defined affinities and policies. Both the PJ and PD define similar elements that can be mapped to each other. Nevertheless, compute and data model sometimes require a different treatment. With the maturing of both models and implementations, we expect that both the PJ and PD model converge in the future. In the following, we discuss further implementation considerations for P*.

D. Considerations

To implement the P* model, there are additional considerations are necessary, e. g. the definition of the exposed end-user abstraction and usage model etc. The API usage model defines how resources are allocated as well as how

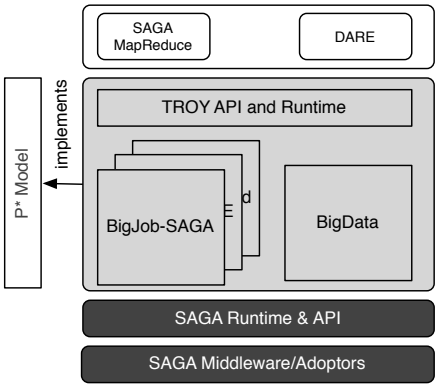

Fig. 2: **TROY – An Implementation of the P* Model:** TROY provides an API for managing PJs and PDs. BigJob and BigData are realizations of the actual PJ and PD functionality. BJ-SAGA and BD-SAGA rely on SAGA for implementation of the PJ/PD.

WUs are described, submitted and monitored. Further, non-functional properties such as fault tolerance and security must be considered. PJ implementations differ in their support for authorization, authentication and accounting. Commonly grid technologies e. g. GSI, VOMS, MyProxy, etc. are used. Further, it can be differentiated between single- and multi-user: The former PJ runs under the identity of a single user and is only able to accept jobs from this user, while the latter PJ is able to accept jobs from different users.

III. TROY: A SAGA-based Implementation of the P* Model

TROY is an implementation of the P* Model (§II). It consists of the TROY API [4], a runtime environment and a set of PJ implementation referred to as BigJobs. As shown in Figure 2, the TROY implementation is based on SAGA, which is utilized for job management, data transfer and coordination. SAGA [18], [12] 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. The TROY API itself was designed to be similar to SAGA in appearance and philosophy: it re-uses many of the well defined and standardized semantics and syntax of the File, Job and Advert API. Further, we aimed for a minimal but complete API.

The TROY API renders the artifacts defined by the P* model, as defined in §II. It defines two description classes that extend SAGA's Job Description: the PJ description and the WU description. A TROY manager class represents a pool of resources – resources can be added by submitting a PJ description to the TROY manager, using the `add_resource()` method. Subsequently, WUs can be submitted to the TROY manager. Resources can be added and removed at any time (i.e. at runtime). TROY

supports different usage modes i) it provides stand-alone PJ functionalities, ii) it provides a unified API to various PJ implementations (e. g. Condor Glide-In and DIANE), and iii) it enables the concurrent usage of multiple PJ implementations. Further, we show in section III-B how TROY is extended to support dynamic data and affinity-based scheduling.

The SAGA inspired approach to TROY's API design, and its also SAGA inspired adaptor based architecture, leverage on the design experiences of SAGA, and seems to appeal our pilot-job user community. Also, the chosen designs allow to very easily exchange the actual PJ implementation and to concurrently use multiple PJ implementations, as will be shown below. TROY thus functions as common access layer for different PJ implementations, providing interoperability and portability of PJ applications. To some extent, the TROY API can be considered to be a prototype of a PJ-like API extension to SAGA (see future work, §VI).

A. BigJob: A Pilot-Job for TROY

PJ implementations in TROY are called BigJob (BJ) [3]. BigJob-SAGA [15] is a BJ implementation which uses SAGA for all remote activities. BJ-SAGA supports a wide range of application types, and is usable over a broad range of infrastructures, i. e. it is general-purpose and extensible. BJ-SAGA is integrated into the TROY runtime environment (see Figure 3). In addition to BJ-SAGA, various other BigJob implementations exist, e. g. there are specific BJ flavors for cloud resources such as Amazon EC2 and Microsoft Azure that are capable of managing set of VMs, as well as a BJ with a Condor Glide-In based backend. In the scope of this paper, BJ generally refers to the BJ-SAGA implementation though.

BJ-SAGA utilizes a M/W coordination model: The BigJob Manager is responsible for the orchestration of pilots, for the binding of WUs and for the scheduling of SUs. For submission of the pilots, BigJob-SAGA relies on the SAGA Job API, and thus can be used in conjunction with different SAGA adaptors, e. g. the Globus, PBS, Amazon Web Service and other adaptors. Each pilot initializes a so called BigJob agent component. The agent is responsible for gathering local information and for executing tasks (SUs) on its local resource. The SAGA Advert Service API 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.

BJ-SAGA currently uses a simple binding mechanism: each WU (a task) submitted to the BigJob framework is mapped to one SU (a so called sub-job). Binding takes places at submission time (early binding). For scheduling, a simple FIFO scheduler is used (see also Table II).

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. The TROY API allows for 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 (e. g. BJ-SAGA), 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 WUs to one of the BJs of the pool (late binding). By default a FIFO scheduler and an affinity-aware scheduler (i. e. a scheduler that considers relationships between work and data units managed by BigJob and BigData, see below) are provided. Other backends (such as DIANE and Condor) natively support elasticity, but can nevertheless be combined into a BJ pool.

B. BigData: Pilot-Data for TROY

BigData-SAGA (BD-SAGA) is the SAGA-based implementation of the Pilot-Data abstraction – in the scope of this paper it is referred to as simply BigData (BD). Figure 4 gives an overview of the architecture. The system consists of two components: the BD manager, and the BD agents deployed on the physical resources. The coordination scheme used is again M/W with some intelligence that is located de-centrally at the BD agent. As communication mechanism the SAGA Advert Service is used, in a similar push/pull mode as for BJ.

The BD manager is responsible for 1) meta-data management, i. e. it keeps track of the pilot stores that a pilot data object is associated with, 2) for scheduling data movements and data replications (taking into account the application requirements defined via affinities), and 3) for managing data movements activities. Similar to BigJob, an agent on each resource is used to manage the physical storage on a resource. The BD scheduler particularly supports affinity-aware scheduling: both BigJob and BigData are tightly integrated to efficiently support compute- and data-related aspects of dynamic execution (see also §IIIA).

IV. Understanding Other Pilot-Jobs

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. There is a variety of PJ implementations: Condor Glide-In [11], SWIFT [20], DIANE [16], DIRAC [9], PanDA [10], ToPoS [6], Nimrod/G [8], Falkon [17] and MyCluster [19] to name a few.

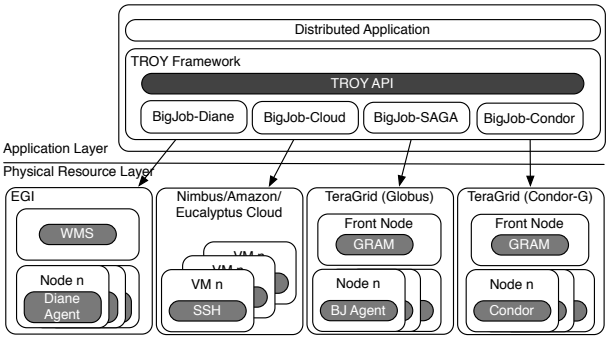

Fig. 3: **BigJob – SAGA-based TROY Implementation:** BigJob is the implementation of the actual PJ functionality for TROY. Various BJ implementation for different grid and cloud backends exist.

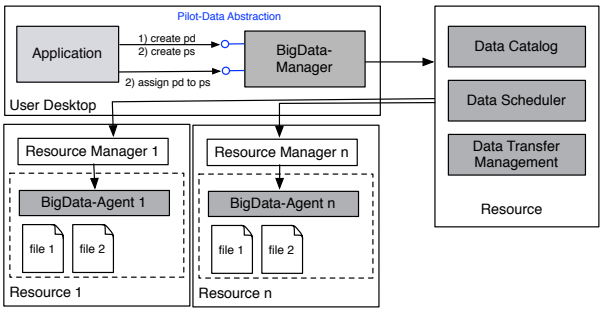

Fig. 4: **BigData Architecture:** The BD Manager exposes TROY's PD API. Application can create group of files and assign files to storage. The BD manager tracks file locations in the data catalog. The scheduler optimizes data-compute co-location. The transfer manager initiates and monitors data movements.

The aim of this section is to show that our P* Model can be used to explain/understand some of these PJ implementations, in particular DIANE as well as Condor Glide-In and Swift. Table I shows how the elements P* model can be mapped to these frameworks. Table II compares the characteristics of the four PJ implementations.

A. DIANE

DIANE [16] is a task coordination framework, which was originally designed for implementing master/worker applications, but also provides PJ functionality for job-style executions. DIANE utilizes a single hierarchy of worker agents as well as 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 [7] can be used. Once the worker agents are started they register themselves at the RunMaster. In contrast to BJ-SAGA, a worker agent generally manages only a single core and thus, by default is not able to run parallel applications (e. g. based on MPI). BJ-SAGA utilizes an agent that is able manage a set of local resources (e. g. a certain number of nodes and cores) and thus, is capable of running parallel applications. For communication between the RunMaster and worker agents point-to-point messaging based on CORBA is used. CORBA is also used for file staging, which is not fully supported by BJ-SAGA, yet.

DIANE is primarily designed with respect to HTC environments (such as EGI [2]), i. e. one PJ consists of a single worker agent with the size of 1 core. BJ-SAGA in contrast is designed for HPC systems such as TG, where a job usually allocates multiple nodes and cores. To address this issue a so-called multinode submitter script can be used: the scripts starts a defined number of worker agents on a certain resource. However, WUs will be constrained to the specific number of cores managed by a worker agent. A flexible allocation of resource chunks as with BJ-SAGA is not possible. By default a WU is mapped to a SU; application can however implement smarter allocation schemes, e. g. the clustering of multiple WUs 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 as BJ-SAGA a single-user PJ, i. e. each PJ is executed with the privileges of the respective user. Also, only WUs of this respective user can be executed by DIANE. DIANE supports various middleware security mechanisms (e. g. GSI, X509 authentication). For this purpose it relies on GANGA. The implementation of GSI on TCP-level is possible, but currently not yet implemented. Further, DIANE supports fault tolerance: basic error detection and propagation mechanisms are in place. Further, an automatic re-execution of WUs is possible.

B. Condor Glide-In

Condor GlideIn pioneered the Pilot-Job concept [11]. The pilot is actually a complete Condor pool that is started using the Globus service of a resource. Subsequently, jobs can be submitted to this 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 conder_master that is generally responsible for managing all daemons on a machine, the conder_collector which collects resource information, the conder_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. WU, and schedulable entity, i. e. SU. Both entities are referred to as job. However, it supports late binding, i. e. resources a job is submitted to must generally not be available at submission time. The scheduler matches the capabilities required by a WU to the available resources.

P* Element	BigJob-SAGA	DIANE	Condor	Swift-Coaster
Manager	BigJob Manager	RunMaster	condor_master, condor_collector, condor_negotiator, condor_schedd	Coaster Service
Pilot-Job	BigJob Agent	Worker Agent	condor_master, condor_startd	Coaster Worker
Unit of Work	Task	Task	Job	Application Interface Function (Swift Script)
Unit of Scheduling	Sub-Job	Task	Job	Job

TABLE I: P* Elements and Pilot-Job Frameworks

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.

C. SWIFT-Coaster

SWIFT [20] is a scripting language designed for expressing abstract workflows and computations. The language provides among many things capabilities for executing external application as well as the implicit management of data flows between application tasks. For this purpose, SWIFT formalizes the way that applications can define data-dependencies. Using so called mappers, these dependencies can be easily extended to files or groups of files. The runtime environment handles the allocation of resources and the spawning of the compute tasks. Both data- and execution management capabilities are provided via abstract interfaces. SWIFT supports e. g. Globus, Condor and PBS resources. The pool of resources that is used for an application is statically defined in a configuration file. While this configuration file can refer to highly dynamic resources (such as OSG resources), there is no possibility to manage this resource pool programmatically. By default a 1:1 mapping for WU and jobs is used. However, SWIFT supports the grouping SUs as well as PJs. For the PJ functionality the Coaster [5] framework is used. Coaster relies on a master/worker coordination model; communication is implemented using GSI-secured TCP sockets. SWIFT and Coaster supports various scheduling mechanisms, e. g. a FIFO and a load-aware scheduler.

V. Experiments and Results

To validate the abstractions we have presented we conducted a series of experiments. We execute BFAST [13] - a genome sequencing application. To manage the workflow we use the DARE [14] framework which builds upon TROY. On the LONI [1] cluster Oliver we utilize a total of 16 cores distributed across four nodes. We make use of two BigJob implementations: BJ-SAGA and BJ-DIANE. We define eight WUs for BFAST. We run the experiments in three different setups: (i) BJ-SAGA only, (ii) BJ-DIANE only and (iii) both BJ-SAGA and BJ-DIANE. In case (iii) on each backend four WUs are executed.

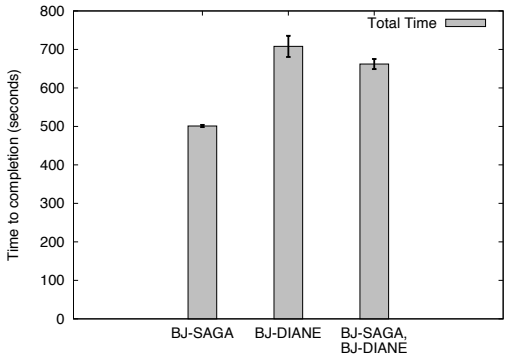

Fig. 5: **Performance TROY with BJ-SAGA and BJ-DIANE:** Running BFAST on a 4 node cluster with a total of 16 cores. The time-to-completion for BJ-DIANE is higher than for BJ-SAGA mainly due to the higher startup time required and some light runtime overhead caused by the additional agents required on the resource.

While BJ-SAGA utilizes one BJ agent on the resource, BJ-DIANE currently requires the spawning of one worker agent per WU that must be executed in parallel. The TROY API marshals these differences, i. e. while the API remains the same for both backends, the different semantics in the PJ implementation are handled by the TROY backend.

Each BFAST application-kernel requires two cores; a total of eight WUs allocating two cores each is specified and assigned to TROY. Each WU is associated with a set of input files. After submission of the WU to the PJ manager, the TROY runtime takes care of binding and scheduling the WUs to the pilots. The application can monitor the state of the PJs and WUs using the TROY API.

Figure 5 shows the results of the experiments. The time-to-completion for BJ-DIANE is about 207 sec longer than for BJ-SAGA. The main contributor for this increased runtime is the deployment time required for DIANE. In a multi-node setup multiple work agents must be used (8 in this case). For each worker agent DIANE must be downloaded, installed and started separately. In total this requires about 178 sec. Further, each DIANE worker

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

TABLE II: P* Characteristics and Pilot-Job Implementations

agent is queued as a separate job at the local resource manager – this contributes the the higher deviation in the measured runtimes. Because BJ-SAGA requires a pre-run installation on each site, it only shows a startup time of 28 sec. Additionally, we also observed a runtime overhead of about 17 sec for the BJ-DIANE scenario. This overhead is likely caused by the additional agents required.

Finally scenario iii) demonstrates that two BJ implementations can be utilized concurrently using the TROY API. The performance in this scenario is slightly better than in the DIANE only case, mainly due to the fact that only four DIANE worker agents need to be started. Also, only half of the WUs are executed on a DIANE node and thus, show a longer runtime. While there are some limitations in the current BJ-DIANE implementation, the aim of this experiment is to emphasizes the possibilities that the TROY API provides to dynamic applications. TROY enables applications to utilize a dynamic resource pool consisting of resources of different infrastructures, e. g. EGI and TG/XD resources. Dynamic applications can utilize the elasticity of the TROY resource pool e. g. to improve the time-to-completion and/or to scale the accuracy of their computations.

VI. Conclusion and Future Work

We established PJ and PD as abstractions for supporting dynamic execution by decoupling workload and resource assignment/scheduling. The P* model provides a common framework for describing and characterizing Pilot-abstractions. We validate the P* model by demonstrating that the most widely used PJ implementations, viz., DIANE, Condor Glide-In and SWIFT can be compared, contrasted and analyzed using this framework, i. e. the architecture and the communication and coordination schemes. TROY is an implementation of the P* model that captures the commonalities between the different PJ implementations via a common API.

Having established the validity of the P* Model and TROY, in the future, the TROY framework will be extended to support advanced autonomic resource management and selection strategies, e. g. by deploying more decentral decision logic into the agents. We will use existing and emerging capabilities of TROY to support the efficient and scalable solution of many scientific applications that involve multiple independent tasks

Acknowledgements

This work is funded by Cybertools project (http://cybertools .loni.org; PI Jha) NSF/LEQSF (2007-10)-CyberRII-01, HPCOPS NSF-OCI 0710874 award, 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). 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 the e-Science Institute, Edinburgh for supporting the research theme. "Distributed Programming Abstractions" & 3DPAS. We thank J Kim (CCT) for assistance with the DNA models. SJ acknowledges useful related discussions with Jon Weissman (Minnesota) and Dan Katz (Chicago). This work has also been made possible thanks to computer resources provided by TeraGrid TRAC award TG-MCB090174 (Jha). We thank Ole Weidner for useful feedback.

References

[1] http://www.loni.org.
[2] EGI. http://www.egi.eu/.
[3] SAGA BigJob. http://faust.cct.lsu.edu/trac/bigjob.
[4] TROY API. https://svn.cct.lsu.edu/repos/saga-projects/applications/bigjob/branches/bigjob_overhaul/.
[5] Coasters. http://wiki.cogkit.org/wiki/Coasters, 2009.
[6] Topos - a token pool server for pilot jobs. https://grid.sara.nl/wiki/index.php/Using_the_Grid/ToPoS, 2011.
[7] Frederic Brochu, Ulrik Egede, J. Elmsheuser, K. Harrison, R. W. L. Jones, H. C. Lee, Dietrich Liko, A. Maier, Jakub T. Moscicki, A. Muraru, Glen N. Patrick, K. Pajchel, W. Reece, B. H. Samset, M. W. Slater, A. Soroko, C. L. Tan, and Daniel C. Vanderster. Ganga: a tool for computational-task management and easy access to grid resources. *CoRR*, abs/0902.2685, 2009.

[8] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. *High-Performance Computing in the Asia-Pacific Region, International Conference on*, 1:283, 2000.

[9] Adrian Casajus, Ricardo Graciani, Stuart Paterson, Andrei Tsaregorodtsev, and the Lhcb Dirac Team. Dirac pilot framework and the dirac workload management system. *Journal of Physics: Conference Series*, 219(6):062049, 2010.

[10] Po-Hsiang Chiu and Maxim Potekhin. Pilot factory – a condor-based system for scalable pilot job generation in the panda wms framework. *Journal of Physics: Conference Series*, 219(6):062041, 2010.

[11] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, July 2002.

[12] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Andre Merzky, John Shalf, and Christopher Smith. A Simple API for Grid Applications (SAGA), 2008.

[13] N. Homer, B. Merriman, and S. F. Nelson. BFAST : An alignment tool for large scale genome resequencing. *PLoS One*, 4(11):e7767, 2009.

[14] Joohyun Kim, Sharath Maddineni, and Shantenu Jha. Building gateways for life-science applications using the distributed adaptive runtime environment (dare) framework. In *Proceedings of TeraGrid'11 Extreme Discovery*, 2011.

[15] 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.

[16] Jakub T Moscicki. Diane - distributed analysis environment for grid-enabled simulation and analysis of physics data. 2003.

[17] Ioan Raicu, Yong Zhao, Catalin Dumitrescu, Ian Foster, and Mike Wilde. Falkon: A Fast and Light-Weight TasK ExecutiON Framework. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[18] SAGA, 2008.

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

[20] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, In Press, Accepted Manuscript:–, 2011.