A Framework for Pilot-Abstractions for Dynamic Execution

André Luckow[1], Mark Santcroos[2,1], Sharath Maddineni[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 Cyberinfrastructure and Applications require dynamical resource utilization models and not static resources. Pilot-Jobs have been one notable success – in the scope of usage and number of CI that support them and applications that use them. However, in spite of broad uptake, there does not exist a well defined, unifying theoretical framework for Pilot-Job using which different implementations can be compared, contrasted and defined. This paper is an attempt to (i) provide a minimal but complete model/framework of Pilot-Jobs, (ii) extend the basic framework from compute jobs to data, (iii) introduce TROY (tiered resource overlay) as an implementation of this framework using SAGA, i.e., consistent with its API, job-model etc., (iv) mapping DIANE – an existing and well known PJ – to P, we establish the generality of the model/framework (P*), (v) establish and validate the implementation of the TROY API by concurrently using BigJob and DIANE across multiple infrastructures.* ***Note: introduce P**

Outline

***shantenu: Need consistency in the way Pilot-Job is written: currently we have pilot-job, Pilot-Job, pilot job, and possibly more... ***andre: Would Pilot-Job and Pilot-Data be ok?

```
1. Introduction:
- Distributed CI and the need/role of Dynamic
Execution
- PJs as an effective abstraction for DE
- Brief Overview of the status of PJ
-- Many PJ out there but no consistent terminology,
framework to compare/contrast
- Basis for extension of Pilot concepts to
other dimensions

2. A Conceptual FW for Pilot-abstractions for DE

3 TROY: SAGA-based implementation of Pilot-abstractions

TROY = Pilot-Job (BigJob) + Pilot-Data (BigData)

4 Analysing other PJs using the Conceptual FW

5 Experiments/Implementation/Comparision:
- PJ interop using the TROY API using DARE
-- this is where DIANE, BigJob will be used together
using TROY API (effectively just BigJob)
-- all other experiments, measurements
and validation tests
```

The primary objectives of this work are:

1) Establish the need for dynamic execution of applications - distributed as well as high-end performance.
2) We define the basic characteristics of the dynamic apps and we understand the requirements of dynamic apps need to do in a distributed environment.
3) We understand the capability that must be provided by the infrastructure to support these application
4) We describe the pilot-job as a good prototype of an abstraction that supports dynamic execution
5) We define the characteristics that need to be supported by a pilot-job ***shantenu: Infrastructure or Application characteristics? ***andre: I think we meant application characteristics
6) There exist multiple PJ implementations out there but no way to compare and contrast. Provide a framework to aid an understanding of pilot-jobs and the ability to compare, contrast and understand different pilot-jobs. Provide both a theoretical and empirically useful approach to determining which PJ to use
7) Empirical implementation of TRoY and demonstration of concurrent/interoperation between equivalent but distinct Pilot-Job implementation. Highlight unique feature of Troy: User extensible and customisable. ***shantenu: We should talk about this in light of the reviews of the paper with Bishop

Points 1-3 can go into the beginning of §2. Points 4 & 5 should be addressed in both introduction (see one of the ***shantenu: above), as well as in the beginning of §2. Points 6, 7 are addressed in the Introduction.

I. Introduction and Overview

There exist multiple reasons why distributed applications have not been able to utilize distributed cyberinfrastructure effectively and without immense effort [**?**]. At the root of the problem is the fact that developing large-scale distributed applications is fundamentally a difficult process. The range of proposed tools, programming systems and environments is bewildering large, making integration, extensibility and interoperability difficult.

Additionally, existing development and execution models and abstractions are mostly remnants of cluster and high-performance computing – which almost by definition impose and imply a static resource utilization model. ***shantenu: Define/elaborate on static resource model.

On reflection of the properties of distributed cyberinfrastructure, and how they are provisioned and federated, it is ipso facto determined that applications that strongly and statically bind to resources are unlikely to be able to scale – either due to hindrances arising from failures, unpredictable system loads and or changing application requirements and resource availability, amongst other factors.

Viewed from the other side, there exists empirical evidence, and our own experience also 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 to efficiently utilizing distributed resources. In other words distributed cyberinfrastructure and applications demand dynamic resource assignment....

***shantenu: Need a sentence a two explaining/making the connection between pilot-jobs and dynamic resource assignment Applications using the Pilot-Job abstraction provide the strongest confirmation of this assertion [] ***shantenu: Need good citation .

Interestingly there exists many implementations of the Pilot-Job abstraction, wherein different projects and users have rolled-out their own. The fact that users have voted with their feet for Pilot-Jobs, reinforces the fact that the Pilot-Job is both an useful and correct abstraction for distributed cyberinfrastructure; 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 existing status of the usage and availability of the Pilot-Job abstraction vis-a-vis the current landscape of distributed applications and cyberinfrastructure.

***shantenu: refine erstwhile motivation Distributed cyberinfrastructure are inherently dynamic: resources can suddenly fail or new resources can become available at any time. Generally there are two types of dynamism.

- **Resource dynamism:** Distributed systems are inherently dynamic – resources can become available or fail at any time. The same holds for network connections. Further, hybrid infrastructures comprised of different resources classes can vary significantly in their costs for usage, performance, availability and the guarantees for quality of service they provide.
- **Application dynamism** describes the requirement of many applications to support dynamic resource requirements. An example are applications whose execution time resource requirements cannot be determined exactly in advance (either due to changes in runtime requirements) or those that are dependent on dynamic data (e. g. sensor, in-transit or variable source/sink of data). Further, the application requirements may change, due to, for example, a failure or an application event.

Further things to consider:

- dynamic scheduling
- dynamic task placement
- autonomic behaviors: Monitoring of the system/application state and adaptations of the application and/or resources to respond to changing requirements or environment.
- Multi-level and multi-dimensional scheduling (in a distributed context). Make reference to it.

The real power of distributed systems, however, arises from adaptive algorithms and implementations that provide applications with an agile execution model, and thus the ability to use resources dynamically as opposed to a static execution model inherited from parallel and cluster computing

To achieve our objectives, we begin this work with an attempt to provide a minimal, but complete model – for the Pilot-Job abstraction, to provide a common and consistent framework to compare and contrast different Pilot-Jobs. This is, to the best of our knowledge, the first such attempt.

A natural and logical extension of the Pilot-Job model, arising from the need to treat data as a first-class schedulable entity, is a concept analogous to Pilot-Job: the Pilot-Data. Given the consistent treatment of data and compute, as potentially equal components in a framework to support dynamic resource and execution, we refer to as the P-* Model ("P-star").

In §3 we introduce TROY – A Tiered Resource Overlay framework, as an implementation of the P-* Model using the SAGA API. The Pilot-Job and Pilot-Data concepts 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 *atomic* instances of BigJob for different backends. Thus we posit that the TROY API is a general purpose API that can be used for all Pilot-Jobs. Before validating this claim, in §4 of this paper, we discuss the

mapping of DIANE to the P-* Model and outline briefly how other well known Pilot-Jobs can be understood using the *vectors* [**?**] of the P-* Model.

In §5 we validate the TROY API by demonstrating how DIANE – an existing and widely used Pilot-Job, can be given a well defined API via the TROY-BigJob API. To further substantiate the impact of TROY-BigJob API, we will demonstrate interoperability between different Pilot-Jobs – a native SAGA based Pilot-Job, referred to as bigjob/advert with DIANE. We believe this is also the first demonstration of interoperation of different Pilot-Jobs.

***shantenu: Should we to introduce Dynamic Applications explicitly in the title? Just a question, not a suggestion...

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

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 Pilot-Job abstraction has been shown to be an effective abstraction to address many requirements of scientific applications. Specifically, Pilot-Jobs support the decoupling of workload submission from resource assignment; this results in a flexible execution model, which in turn enables the distributed scale-out of applications on multiple and possibly heterogeneous resources. Most Pilot-Job implementations however, are tied to a specific infrastructure. In this paper, we describe the design and implementation of a SAGA-based Pilot-Job, which supports a wide range of application types, and is usable over a broad range of infrastructures, i.e., it is general-purpose and extensible, and as we will argue is also interoperable with Clouds.

A. Elements of the P-* Model

A. **Pilot-Job Framework:** The PJ framework provides the abstraction of a container for multiple Unit of Works that may be dynamically added to it independently from the underlying resource pool. It provides a mechanism to decouple unit of works (the compute "tasks") from being hard-coded to a specific "resource" or delay the binding.
***Note: Tasks are ultimately loaded onto specific resources using the pilot-job and late-binding. In other words ***Note: PJ provides a mechanism to decouple "task coordination" from "resource mapping".
***Note: Facilities provided include the creation of a PJ, insertion of tasks, and attachment to a CPU resource pool for late-binding task execution. ***andre: Remark from AndreM: we need to careful when talking about Pilot-Job as a framework and Pilot-Job as the placeholder job (agent).

B. **Pilot-Job:** The PJ (also referred to as pilot) is the entity that actually gets submitted and scheduled on a resource (E). Commonly, the PJ utilizes an agent to manage the set of allocated resources.

C. **Unit of Work (UW):** A unit of work is the workload that encapsulates a self-contained piece of work and will be assigned to a resource indirectly via a Pilot-Job. The PJ in turn has flexibility in determining when and how-many resources the UW will receive.

D. **Unit of Scheduling (US):** are entities that get scheduled to the resource, and to which UWs are assigned (inside the PJ framework). A sub-job runs an instance of the application kernel G.
***shantenu: I changed the US description significantly

E. **Resource:** A storage/compute resource that has a common entry point (like a queue). Commonly, a resource is accessed by a LRM. ***andre: move to impl. section

F. **Application:** is upper layer on the stack e.g. DARE-NGS. The application utilizes the BigJob API to execute instances of G.

G. **Application Kernel:** An application kernel is actual binary that gets run, e.g. `/usr/local/bin/bfast` (Alternative terms: Program, Executable, Software).

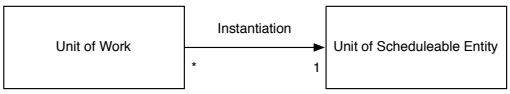

Fig. 1: Relationship between Unit of Work and Scheduleable Entity

***Note: to be moved to end Example usage mode of a pilot job:

1) The pilot is executed on a certain resource.
2) UW gets bound to a pilot-job via the central manager. ***andre: I think at this point we must clearly differentiate between pilot (aka the submitted agent) and pilot (aka the framework)
3) UW gets scheduled to a US (either one or multiple UW can be scheduled to a US). The properties of the US are dependent on the pilot job which they are running in. Scheduling has in addition to the spatial component (which US?) also a temporal component (at what time?).
4) US gets scheduled to a physical resource on which the pilot is operating.

B. Characteristics of P*-Model:

We propose a set of fundamental properties/characteristics that aid the description of P* frameworks. Further, these properties are important for the implementation of P-*.

Coordination: describes how the various components of the pilot-job framework are internally coordinated. Commonly, distributed coordination mechanisms, such as master-worker or a set of distributed software agents are utilized.

- **Decision Making:** In the **central** model information about available resources (aka pilots) are collected by a central manager. Decisions are centrally made by a manager process, which decides which UW is executed on what resource. In the **hierarchical** model the decision making process is divided up into a hierarchy of distributed agents. Each of them coordinates a coordinates a defined aspect (e. g. a certain set of resources). In the **decentral** model, control is distributed among the different components. Pilot-Jobs with decentralized decision making often utilize agents that accepts respectively pull UWs according to a set of defined criteria.
- **Push versus Pull Model:** We define the terms push and pull based upon the determinism of the binding. If the task is centrally explicitly addressed to a specific pilot, and there is thus no freedom for a pilot to select the task, we speak of a push model. In all other situations, when there is a degree of freedom for the pilots to select a task, we speak of pull. This model is applicable to different aspects of the pilot-job framework (e.g. to task binding, resource binding, resource additions/removals). ***andre: should we move push/pull to communication? It's is also currently references there already

Communication: describes the mechanisms for data exchange between the components of the framework, can be used (e. g. point-to-point, all-to-all, one-to-all, all-to-one, or group-to-group), stream (potentially unicast or multicast), publish/subscribe or shared data spaces. Shared data spaces (e. g. tuple spaces) as used by BigJob are associated with a push/pull coordination schema, i.e. the central manager pushes something to the shared data space and the agents periodically check for new data.

UW Binding defines how and at what time the assignment of a UW to a Pilot-Job is done.

- Time of binding describes the point at which the binding decision is made, e. g. in the case of early binding the decision is made by the application, while in late binding mode the decision is made by the middleware framework.

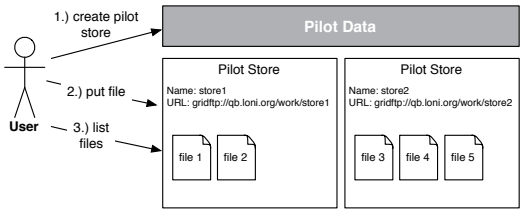

Fig. 2: Pilot Data and Store Overview

- Explicit binding refers to the ability to explicitly request a certain resource for a UW.

UW Scheduling: Task scheduling describes i) the process of mapping UW to US (pilot internal) and ii) of mapping US to physical resources (pilot external). Scheduling decisions are made on basis of a defined set of policies (e.g. resource capabilities, data/compute affinities, etc.).

- Multi-level scheduling: In a distributed environment often multiple levels of autonomous schedulers are involved.
  - The pilot job is schedules using the local RMS or some other Grid (meta)-scheduler.
  - UW are scheduled to US on application-level within the pilot-job.

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

***shantenu: note: DARE == Dynamic Application Runtime Environment: TROY + MapReduce + Other capabilities

1) Overview: Having defined the P* model, we extend this model to dynamic data using a set of abstraction that we call *Pilot-Data*. Pilot-Data is a set of abstractions for expressing data localities and affinities. Pilot-Data can be used to create groups of file clustered together using a quantifiable property, such as affinity (α) e.g., $\alpha = 1.0$ would imply that files are always stored together. ***shantenu: I've distinguished store and usage The concept of correlated access originates in Filecules [5].

Pilot-Data provides a set of basic operations on top of these file groups, whilst Pilot Store is a container that represents a logical group of physical files that share the same affinity. Pilot-Store containers can be used to express data-data affinities. The abstraction supports basic management tasks (create, delete, update, move, list).

The Pilot-Data abstraction serves the following needs:

- Reservation of physical disk space: acquisition of data storage (advanced reservation, place holder)
- Virtual destination: dynamically mapping of data to pilot stores.
- Runtime environment for α based data
- Automatic data partitioning and distribution

The Pilot-Data abstraction defines the following elements:

- **Data Unit:** The base unit of data used by the framework, e. g. a data file or chunk.
- **Pilot-Data (PD):** Allows the logical grouping of files and the expression of data-data affinities. This collection of files can be associated with certain properties. One of this property is affinity.
- **Pilot-Store (PS):** Binds a pilot-data object to a actual physical resource. A pilot-store object can function as a placeholder object that reserves the space for a pilot-data object.

A pilot-data object is a logical container and describes the properties of a group of files. A pilot-store is a placeholder reserving a certain amount of storage. By associating a pilot data object to a pilot store the data is actually moved to the physically location managed by the pilot store. Pilot-Data supports commonly occurring application patterns, such as the scatter/gather pattern commonly used by master-worker or MapReduce-applications.

While the application-level abstraction enables application developers to model data affinities, dependencies etc., the runtime framework will be responsible for managing data-compute co-allocations, data-transfers, the dynamic expansion of storage pools etc.

***andre: insert Millau figure

2) Extension of P-Model:* The P* model can in many parts be applied to Pilot-Data: The communication and coordination element e. g. describes important characteristics of a Pilot-Data implementation. The elements defined in section II-B can be extended by the following elements:

- **Data Characteristics:**
  - Static data refers to data that is infrequently changed and does not need to be moved.
  - Dynamic data refers to different spatial and temporal properties of data:
    - Data that is generated or changing at runtime (temporal).
    - Data that is in place or needs to be moved (spatial).
  - Streaming data
- **Data Access Patterns:** While the P* is primarily concerned with capturing aspects of distributed coordination, this elements is extended to include patterns of data access.
  - Co-Access: a collection of data that is always accessed together
  - One-to-One: two data objects that are always accessed together (a special case of co-access)
  - One-to-All:
  - All-to-All:
  - Scatter:
  - Gather:
- **Affinities:**
  - Data-Data affinity arises when different datasets/elements are required to be at the same compute element.
  - Compute-Compute may arise, for example, either if C1 and C2 are related by a data-element D, which could be the output of C1 and is the required input for C2 (Case-I), or if both C1 and C2 need to operate on (replicas of) the same data-element (Case-II)
  - Data-Compute
- **Unit of Scheduling** describes the core unit of data that is used by the framework.
  - Chunk-based
  - File-based
- **Data management:** Managing input and output files can be critical in particular with the current increase of data volumes. Pilot-Jobs can support different kinds of data management, e. g. file stage-in and out. Data and UW must be efficiently scheduled obeying data-data and data-compute affinities.
- **Data Unit Binding and Scheduling:**
  - Affinity-based scheduling
  - Data-transfer scheduling

Pilot-Job and Pilot-Data encapsulate cross-cutting properties across data and computation. Both Pilot-Job and Pilot-Data can be used to express affinities between data and compute elements.

D. Implementation Considerations

To implement the P* model, additional consideration must be taken e. g. the exposed end-user abstraction and usage model, the type of communication (API or service) etc. In the following we will use the developed framework to develop the TROY-based implementation of the P* model.

The PJ framework must define the usage model, i. e. how are resources allocated, UW specified and assigned, as well as the way how the framework can be accessed, e. g. via a service or API. Pilot-Jobs can be run on different types of homogeneous and heterogeneous resources. Generally, the Local Resource Manager (LRM) is the gateway to local resources. This can be e. g. a PBS/-Torque or WMS service. HPC resources are specifically designed for high-end parallel jobs, while HTC resources are particularly suited for independent ensemble of tasks. Different applications require a mix of HTC and HPC, e. g. when running an ensemble of MPI jobs. Different workload characteristics (UW heterogeneities, parallelism, UW dependencies, etc.) can be supported by a Pilot-

Job, e. g. via special information collector and scheduler capabilities.

Further, non-functional properties must be considered:

- **Fault tolerance:** Large, distributed Grids are highly dynamic and inherently prone to failures and thus unreliable. To deal with failures, systems can deploy strategies, such as automatic resubmission etc.
- **Security**
  - Authorization, authentication, accounting (AAA) describes the authentication and authorization mechanisms supported by the PJ (e. g. GSI, VOMS, MyProxy, etc.).
  - Single vs. multiple user: A pilot-job that runs under the identity of a single user and is only able to accept jobs from this user is referred to as private PJ. A PJ that is able to accept jobs from different users is referred to as public PJ.
- **Resource abstraction** describes the framework that is used for accessing distributed resources (e. g. SAGA, GANGA, JGlobus etc.).

III. TROY: A SAGA-based Implementation of the P-* Model ***shantenu: AL, MS, SM

The API we have designed is similar to SAGA in appearance and philosophy: it re-uses many of the well defined (and standardized) semantics and we attempted to keep it simple in the amount of details it exposes.

***shantenu: Need to define and motivate TROY better, i.e. TROY: BigJob + BigData etc.

TROY is a SAGA-based implementation of the P* model defined in section II. As shown in Figure 3 the framework is located on top of the SAGA API and runtime environment, which is utilized for job management, data transfer and distributed coordination.

Table I gives an overview of the different SAGA-based Troy implementations and their characteristics. BigJob currently supports different backends: SAGA BigJob can be used in conjunction with different SAGA adaptors, e. g. the Globus, PBS, Amazon Web Service and local adaptor. In addition to the SAGA-based reference implementation, there are various other implementations of the BigJob API, e. g. for Amazon Web Services, Azure and Condor resources.

A. What is Unique about a SAGA-based Implementation of the P-* Model?

- Consistent with based API and job/file/data model
- thus programmable (eg. affinity-based PJ) and extensible

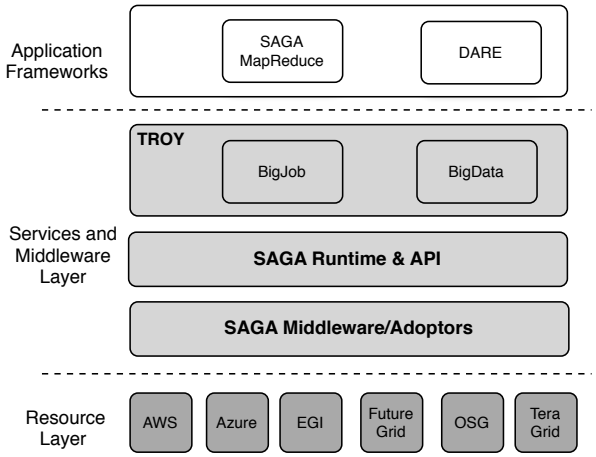

Fig. 3: TROY Overview

- ....
- different usage modes: a stand-alone PJ, API access to other PJ, concurrent usage with other PJ (as will be demonstrated)

***shantenu: MUST provide SAGA URL for updated BigJob API and documentation [3]

B. BigJob: A SAGA-based Pilot-Job

***shantenu: Alternative title: "BigJob: TROY Pilot-Job" ?

BigJob is the SAGA-based Pilot-Job implementation of the TROY framework [2]. It supports a wide range of application types, and is usable over a broad range of infrastructures, i. e. it is general-purpose and extensible. As shown in Figure 5 the framework consists of two layers: the atomic and the dynamic BigJob:

- **Atomic BigJob:** An atomic BigJob is confined to a single cluster resource and represented by a single master process (i. e. a single BigJob Manager or DIANE RunMaster).
- **Dynamic BigJob:** A dynamic BigJob consists of several pilot-jobs distributed across 1 or more resources. Dynamic BigJobs are malleable, i. e. resources can added or be removed.

***shantenu: It is CRITICAL to explain why we need to expose the details of multiple atomic BigJobs to the end-user? Remember part of the whole idea of the exercise is, (i) theory: to provide a framework for understanding any differences, (ii) practice: make all these differences go away from the end user! ***andre: Since we were not sure about the term "atomic", we could also use base bigjob, or core bigjob

The BigJob framework provides several types of atomic BigJob for various resource types. The dynamic BigJob is discussed in section III-B.1.

	Task Binding	Resource Types	Coordination	Communication
BigJob-SAGA				
Globus/PBS adaptor	late binding (at sub-job submission)	HTC/HPC	central decision making	SAGA Advert (pull/push)
Cloud adaptor (EC2)	late binding (at sub-job submission)	HTC/HPC	central decision making	SAGA Advert (pull/push)
BigJob-Condor	late binding (after sub-job submission by Condor)	HTC	central decision making	Condor-internal
BigJob-Cloud	late binding (at sub-job submission)	HTC/HPC	central decision making	local python-based queue / SAGA Job (SSH adaptor)
BigJob-Azure	late binding (at sub-job submission)	HTC	central decision making	Azure Storage (push/pull)
BigJob-Diane	late binding	HTC	decentral decision making	CORBA (pull from master)

TABLE I: SAGA-based Troy Implementations: Characteristics According to Defined Vectors

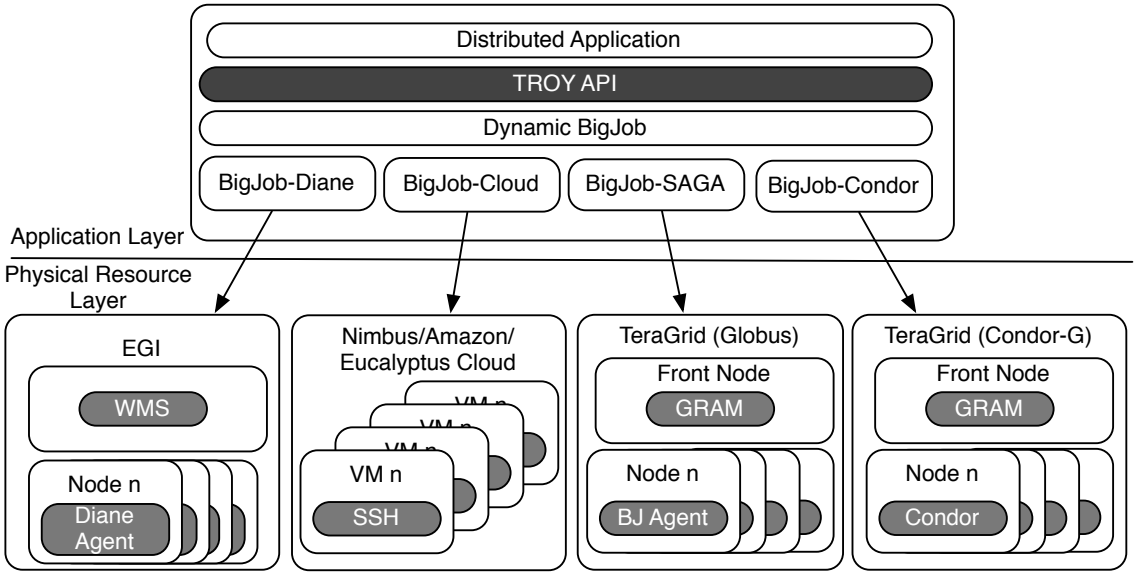

Fig. 4: SAGA-based TROY Implementation - BigJob ***shantenu: fix to include EGI/DIANE

Aspects that need to be addressed:

- Big-Job Agent: capacity (physical size) is a property of an agent. cardinality: how many sub-jobs can be managed by an agent?
- Sub-Job Agent: Agent assignment should be separated from resource assignment. Agent has the freedom to assign tasks to sub-job in any way it want. Agent can do local decisions.
- Would it make sense to use the "internal" versus "external" coordination concept to distinguish sub-job versus big-job agent

1) Dynamic BigJob: A traditional BigJob is always confined to a single resource. 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 Dynamic BigJob provides an abstraction for multiple big-jobs. These big-jobs can be distributed across multiple resources and types of infrastructures. Each big-job manages it's own resources. An extensible scheduler is used for dispatching sub-jobs to the different big-jobs that are managed by a dynamic big-job.

Dynamic BigJob provides the ability to dynamically add and remove resources to a big-job. The API consists of two parts, the resource management and the resource introspection part:

- `add_resource()`: New resources are added by starting a new big-job.There are various flavors of this method:
  - `add_resource(resource_dictionary)`: Start another big-job on the resource defined in the `resource_dictionary`.

  - `add_resource(affinity, number_cores)`: Add another big-job to the specified affinity group.
- `remove_resource(bigjob)`: Removes the big-job from the resources.

Higher-level wrappers that encapsulate e. g. the specific resource descriptions can be implement. Further, to implement this dynamic resource capabilities it is necessary to provide different dynamic resource introspection in the dynamic big-job layer:

- `get_resources()`: returns a list of managed big-job objects. Each big-job object can be queried for it's allocated resources (number nodes, number cores).

***shantenu: The rationale behind dynamic BJ is that for the same application scenario different BJ with different properties/characteristics may be required. Thus dynamic BJ maybe comprised of either homogeneous or heterogeneous atomic BJs

In particular for data-intensive applications data locality is an important concern. Different types of affinity, e. g. data-data or data-compute, exists. Dynamic BigJob provides support for data-compute affinities. Each resource (i. e. each big-job) can be assigned to a certain affinity. The affinity-aware scheduler then ensures that sub-jobs that demand a certain affinity are only executed on resources that fulfill this constraint.

C. BigJob for Cloud Computing

***shantenu: Once again – why do differences in execution details between grids and clouds result in the need for different atomic BJs needs to be explained. Must emphasis that the API remains the same.

At the execution level, clouds differ from Clusters/-Grids in at least a couple of different ways. In cloud environments, user-level jobs are not typically exposed to a scheduling system; a user-level job consists of requesting the instantiation of a virtual machine (VM). Virtual machines are either assigned to the user or not (this is an important attribute that provides the illusion of infinite resources). The assignment of job to a VM must be done by the user (or a middleware layer as BigJob). In contrast, user-level jobs on grids and clusters are exposed to a scheduling system and are assigned to execute at a later stage. Also a description of a grid job typically contains an explicit description of the workload; in contrast for Clouds a user level job usually contains the container (description of the resource requested), but does not necessarily include the workload. In other words, the physical resources are not provisioned to the workload but are provisioned to the container. Interestingly, at this level of formulation, pilot-jobs attempt to provide a similar model of resource provisioning as clouds natively offer.

1) BigJob and SAGA AWS Adaptor: BigJob provides support for various cloud computing environment. The SAGA BigJob implementation can be used in conjunction with the AWS adaptor for SAGA to run on EC2-based cloud infrastructures, such as FutureGrid. However, there are some limitations mainly caused by the restrictions of SAGA/AWS adaptor for the SAGA Job package. The SAGA job service object e. g. does not provide a mean to specify a set of resources. Using the AWS adaptor it is only possible to utilize a single VM instance, which must be configured prior to the run in a configuration file. If multiple VMs are required, the dynamic BigJob implementation must be used. In this case however, it is still not possible to run MPI jobs across multiple VMs. ***sharath: why is it not possible to run MPI jobs across Multiple VM's? ***andre: MPI jobs are (unless you do something outside of BJ) constraint to run on resources managed by a single BJ agent. The agent must generate a nodefile from this list of resource it is managing. The agent is not aware of resources managed by another BJ)

2) BigJob Cloud & BigJob Azure: To address this limitation, BigJob-Cloud [7] was developed. BigJob-Cloud provides an implementation of the BigJob API, which is completely independent from the SAGA (and thus, the SAGA AWS adaptor). It directly utilizes the Amazon tools to access cloud resources. It can manage cluster of VM; for this purpose BigJob provides a rich interface for describing cloud resources. For this purpose a Python dictionary is used (see section **??**). The VMs can be managed centrally by the BigJob manager: All VMs have a public IP and there is no need to interface with a local resource manager (SAGA BigJob e. g. evaluates the `$PBS_NODEFILE` to obtain a list of resources). Thus, it is not necessary to deploy an agent on the VM - all necessary metadata can be obtained from the AWS backend. Job are spawned via SSH.

BigJob-Azure [8] utilizes a similar approach as BigJob-Cloud. It utilizes the Azure REST interface to startup VM Worker Roles. However, since Azure does not support SSH access it is necessary to utilize an agent-based approach. For communication between the agent and the manager the Azure Storage is used.

***mark: If BigJob is the atomic unit, it should not differ per backend ***andre: That's mainly a restriction of the job package which does not really map to AWS. There is no common way in the job package to specify the # of resources that suppose to be used. Thus, this limitation

D. BigData

BigData is the SAGA-based implementation of the Pilot Data abstraction. Figure 5 gives an overview of the architecture. The system consists of two components: the

Pilot Data manager and the agents deployed on a specific physical storage resource. The 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 and 2) it schedules data movements and data replications taking into account the application requirements defined via affinities. Data placement and locality is the key for a optimal performance of data-intensive applications. The architecture is designed to function as a **user-level overlay** on top of existing storage resources.

***shantenu: Need to explain/describe architecture of BigData? using the terminology of Section II and P*-Model

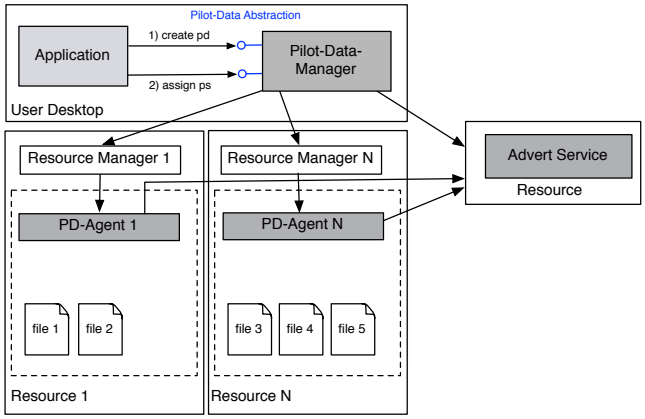

Fig. 5: Pilot Data Architecture

Each pilot store is managed by an agent. The agent can be started manually or using the Pilot Data API. A possible implementation option would be the integration of the PD and BigJob agent, which is particularly useful for managing data-/compute-affinities.

A core part of the data manager is the data scheduler. The scheduler aims for a optimum of data and compute locality for an applications.

- Move data to compute
- Move computer to data
- Streaming of data
- Data prefetching
- Replication

1) Data Movement: The Pilot Data implementation is based on SAGA and thus, is infrastructure independent. It supports all underlying SAGA adaptors (SSHFS, GridFTP) and future adaptors such as Globus Online.

Work on optimizing file transfers: Kosar[2011]

Work on reliable file transfer: RFT, Globus Online

2) Related Work: iRods

Stork

BitDew

Random Notes

- Focus on Desktop Grid
- Java-based implementation (ie difficult to interface with Python-based PS/SAGA)
- highly distributed: stable and volatile nodes
- pull model, i.e. a node pulls for new data

Mapping to BitDew:

- Pilot Store in its current implementation covers Bitdew Data Catalog and Repository
- For data management and placement the Active Data API and the Bitdew data scheduler could be used
- Transfer Management is done via SAGA File API

How to evolve pilot data/store?

- Active management of data (e.g. replication, automatic affinity management) requires an active component:
  - Manager/Agent model as in BigJob?
  - Who runs active components? Started as part of batch job or separate install/start?

Questions:

- How should we store data in order to effectively cope with non-uniform demand for data?
- How many copies of popular data objects do we need?
- Where should we store them for effective load balancing?

E. TODO/Future Work

The current framework provides building blocks for expressing data localities and operation on file groups (similar to filecules).

Limitations:

- No active agent that monitors state of files
- No placement policy support or autonomic behavior
- Infrastructures generally expose insufficient locality/-topology information
- Compute – Data Affinity: Dynamic BigJob with affinity only provides a very coarse-grained affinity
- No policy for what's happening if data is not available in right location:
  - Run anyways – affinity is just an hint
- When to move pilot stores? Move or copy?
- Move data to compute or visa versa?
- Data Replication: Identification of the same file: logical filename -¿ physical files. Manage replication process (consistency!)

IV. Understanding Other Pilot-Jobs

*****shantenu: MS - to focus on DIANE**

***shantenu: Depending upon where the TROY API is

discussed we can have two ways forward. If TROY API is discussed in §3, then we go for Mode I, where Mode I: The aim of this section is to show: (i) that our P* Model can be used to explain/understand DIANE, (ii) Show that the TROY API can be used to marshall Diane stand-alone, (iii) Using TROY API, both BigJob and Diane can be used standalone

***shantenu: If TROY API is discussed in §5, then we go for Mode-II, where Mode II: The aim of §4 is to show: (i) that our P* Model can be used to explain/understand DIANE. Then in §5, after having discussed TROY API we, (ii) Show that the TROY API can be used to marshall Diane stand-alone, (iii) Using TROY API, both BigJob and Diane can be used stand alone

***shantenu: I think there was agreement to go with Mode II

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: (i) that our P* Model can be used to explain/understand DIANE as well as other Pilot-Job frameworks.

A. Diane

DIANE [9] is a task coordination framework which implements the Master/Worker pattern. Table II compares the BigJob-SAGA and Diane.

***mark: Whats VO in this context? ***andre: Suppose to describes the capability to dynamically add/remove resource from/to a BigJob

***andre: Update: atomic pilot, multiple pilot model

B. Condor-G

Condor-G pioneered the Pilot-Job concept [6]. Using Condor-G a complete Condor pool can be initiated using the GRAM service; subsequently jobs can be submitted to this pool using the standard Condor tools and APIs.

C. SWIFT

SWIFT [11] relies on its scripting language to describe 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 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 way to manage this resource pool programmatically. By default a 1:1 mapping for UW and jobs is used. However, SWIFT supports the clustering of UWs into a single large job as well as pilot-jobs for which the term Coaster [1] is used. Coaster relies on a master-worker coordination model; communication is implemented using GSI-secured TCP sockets.

***shantenu: It should probably be Coasters – which is their notion of a pilot-job. Just to keep life interesting, they call it head-job and not pilot-job!

`http://www.ci.uchicago.edu/swift/guides/release-0.92/userguide/coasters.php`

***andre: SWIFT eval: no standard resource abstraction (SAGA), proprietary language (not Python), TODO: check how coasters work! 1 coaster == 1 Condor-G job?

D. Other Pilot-Jobs and Conclusion

Table III compares the three Pilot-Job frameworks discussed in this section.

In addition to the three Pilot-Job framework discussed in this section, various other frameworks exist.

- MyCluster
- Falkon [10] is a Pilot-Job framework that emphasizes the performance of its task dispatcher.
- Nimrod/G
- DIRAC [4] is another pilot-job framework used by the LHCb community.
- Topos
- Panda

***shantenu: Can we add some structure to these *other* PJ.. this will be ambitious and time-consuming, but if we can, that'll be (i) a great service to the community, (ii) a strong intellectual addition to the paper by virtue of validation of the P*-model

V. Implementations and Experiments ***shantenu: AL/MS, SM

In this section we discuss the TROY API. Further, (i) we show that the TROY API can be used to marshall DIANE stand-alone as well as (ii) that the TROY API can be used stand alone with both BigJob and DIANE concurrently.

Term	BigJob-SAGA	Diane
Central Coordinator	BigJob/Dynamic BigJob Manager	RunMaster
Resource Agent / Pilot	BigJob Agent	Worker Agent
Number Agents ***shantenu: Whats an "Agent"? ***andre: In this case the agent is the actual pilot job? Should we call it that way?	1 BJ Agent/n BJ Agents	1 Worker Agent per core
Capacity of Agent	n cores	1 core
Scheduleable Unit	Sub-Job	Task
Communication	SAGA-Advert	CORBA
Support for task execution by generations	yes	yes
Task-resource Binding	Late (At sub-job submission or later for dynamic BJ)	Late
MPI/Multinode Applications	yes	no (yes with custom implementation of ApplicationWorker)
Advanced Scheduling	no/ManyJob Scheduler	yes (ITaskScheduler)
Dynamic Resources	no/yes	yes (AgentFactories)
Agent Submission	API	Ganga Submission Script
Application Interfaces	Big-Job/Sub-job Management	Big-Job/Sub-job Management Master/Worker API (`ITaskScheduler`, `IApplicationManager`, `IApplicationWorker`)
Fault Tolerance	Error Propagation	Configurable Re-Tries

TABLE II: BigJob-SAGA versus DIANE: Terms and Features

	SAGA BigJob	DIANE	Condor Glide-In	SWIFT
External Coordination	Local API for UW execution management		Master/Worker	SWIFT language
Internal Coordination	Master/Worker (push)	Master/Worker (pull/-push)	Master/Worker	Master/Worker
External Communication	local	Central Manager exposes local & remote CORBA service		
Internal Communication	Advert Service	CORBA		GSI-enabled TCP
UW Binding & Scheduling				
Security	Middleware dependent (GSI, Advert DB Login)	GSI	-	GSI
Resource Abstraction	SAGA	Ganga/SAGA	Globus	Resource Provider API/Globus CoG Kit
Fault Tolerance				

TABLE III: Pilot-Job Comparison

VI. Conclusion and Future Work

implement a more decentral distributed control mechanism in TROY

Acknowledgements

Mark Santcroos 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.

References

[1] Coasters. `ttp://wiki.cogkit.org/wiki/Coasters`.

[2] SAGA BigJob. `http://faust.cct.lsu.edu/trac/bigjob`.

[3] TROY API. `https://svn.cct.lsu.edu/repos/saga-projects/applications/bigjob/branches/bigjob_overhaul/api/base.py`.

[4] 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.

[5] Shyamala Doraimani and Adriana Iamnitchi. File grouping for scientific data management: lessons from experimenting with real traces. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 153–164, New York, NY, USA, 2008. ACM.

[6] 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.

[7] 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.

[8] Andre Luckow and Shantenu Jha. Abstractions for loosely-coupled and ensemble-based simulations on azure. *Cloud Computing Technology and Science, IEEE International Conference on*, 0:550–556, 2010.

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

[10] 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.

[11] 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.