A Web-based Platform for Publication and Distributed Execution of Computing Applications

Oleg Sukhoroslov, Sergey Volkov, and Alexander Afanasiev
Centre for Distributed Computing
Institute for Information Transmission Problems of the Russian Academy of Sciences (Kharkevich Institute)
Moscow, Russia
Email: sukhoroslov@iitp.ru

***Abstract*—Researchers increasingly rely on using web-based systems for accessing and running scientific applications across distributed computing resources. However existing systems lack a number of important features, such as publication and sharing of scientific applications as online services, decoupling of applications from computing resources and providing remote programmatic access. This paper presents Everest, a web-based platform for researchers supporting publication, execution and composition of applications running across distributed computing resources. Everest addresses the described challenges by relying on modern web technologies and cloud computing models. It follows the Platform as a Service (PaaS) cloud delivery model by providing all its functionality via remote web and programming interfaces. Any application added to Everest is automatically published both as a user-facing web form and a web service. Another distinct feature of Everest is the ability to attach external computing resources by any user and flexibly use these resources for running applications. The paper provides an overview of the platform's architecture and its main components, describes recent developments, presents results of experimental evaluation of the platform and discusses remaining challenges.**

***Keywords*-distributed computing, web-based platform, web services, REST API, software as a service, platform as a service, resource integration, application composition.**

I. INTRODUCTION

The ability to effortlessly use and combine existing computational tools and computing resources is an important factor influencing research productivity in many scientific domains. However, scientific software often requires specific expertise in order to install, configure and run it that is beyond the expertise of an ordinary researcher. This also applies to configuration and use of high-performance computing resources to run such applications. The increasing complexity of problems being solved requires simultaneous use of multiple applications and distributed resources, which brings important issues of application composition and resource integration. To streamline collaboration in multidisciplinary research projects it is crucial to be able to share not only scientific expertise, but software and resources of each partner.

As such, there is a clear demand for high-level tools for publication, sharing and composition of scientific applications supporting execution of these applications across distributed computing resources. A number of systems targeting the mentioned problems have been developed in the recent decade, including grid middleware [1]–[4], scientific workflow systems [5]–[9] web service toolkits [10]–[12], web-based scientific gateways and platforms [13]–[18]. However, several challenges remain, which are not addressed well by existing approaches and systems.

The first challenge is supporting publication and sharing of scientific applications as online services. The use of service-oriented approach can enable wide-scale sharing, publication and reuse of applications, as well as automation of scientific tasks and composition of applications into new services [19]. However, the existing scientific gateways do not expose applications as services thus limiting opportunities for application reuse and interoperability. Moreover, most of such systems do not support publication of new applications by users. While the web service toolkits implement transformation of scientific applications into services, they require installation and lack an infrastructure for hosting services. The scientific workflow systems are mostly desktop tools that do not support publication of produced workflows as new services. The grid middleware implements only generic web services to access computing resources and is too low-level for the majority of researchers.

The second challenge is decoupling published applications from computing resources. The existing scientific gateways run applications on a statically configured set of resources and don't support adding new computing resources by users. This makes it impossible to implement two common use cases. The first is a user wanting to utilize local resources in addition to resources provided by the gateway to run applications. The second is a user wanting to share an application running on custom resources. While the web service toolkits support the last use case, they don't support passing custom resources to the deployed service. As such, it is not possible to share only application and enable users to run it on resources they have. The scientific workflow systems provide such ability by passing workflow descriptions between users [20]. However such approach requires each user to install and run the workflow engine on its machine. The grid

middleware supports running applications on custom subsets of resources with provided user credentials, however all required resources should be accessible via the single grid, which is not always possible.

The third challenge is to make these high-level tools available online without requiring users to install, run and update software on their machines. This requires transition from standalone desktop applications and software toolkits to online multi-user platforms. While we see this transition already in the form of web-based platforms [16], online workflow editors [12] and problem-solving environments [18], current efforts target mostly user-facing interfaces. We believe that it is of equal importance to implement open programming interfaces enabling users to integrate and use these next-generation online platforms with third-party systems. While appealing directly to a relatively small set of skilled users, such approach will benefit all users by enabling new types of applications and use cases.

In this paper we present Everest [21], [22], a web-based platform supporting publication, sharing and execution of scientific applications across distributed computing resources. Everest addresses the described challenges by relying on modern web technologies and cloud computing models. It follows the Platform as a Service (PaaS) model by providing all its functionality via remote web and programming interfaces. A single instance of the platform can be accessed by many users in order to create, run and share applications with each other without the need to install additional software on their machines. Any application added to Everest is automatically published both as a user-facing web form and a web service, addressing the first challenge. Another distinct feature of Everest is the ability to attach external computing resources by any user and flexibly bind such resources to applications, addressing the second challenge. Finally, Everest is built around the open programming interface implemented using RESTful web services and accompanied with the Python API, addressing the third challenge.

Everest was introduced in a position paper [22] that outlined the proposed approach and described an early prototype. In this paper we revisit the platform's architecture, describe recent developments, share results of experimental evaluation of the platform and discuss remaining challenges.

The major contributions of this paper:

- To the best of our knowledge, Everest is the first to introduce a complete PaaS solution enabling users with minimal skills to publish and share scientific applications as services (Section 2).
- Everest implements decoupling of published applications from computing resources by enabling users to attach external resources and introducing dynamic resource binding (Section 3).
- Everest implements programmatic access to the platform's functionality enabling development of client applications and integration with other systems (Section 4).
- We demonstrate the productivity and performance of current Everest implementation by using synthetic and real applications (Section 5).

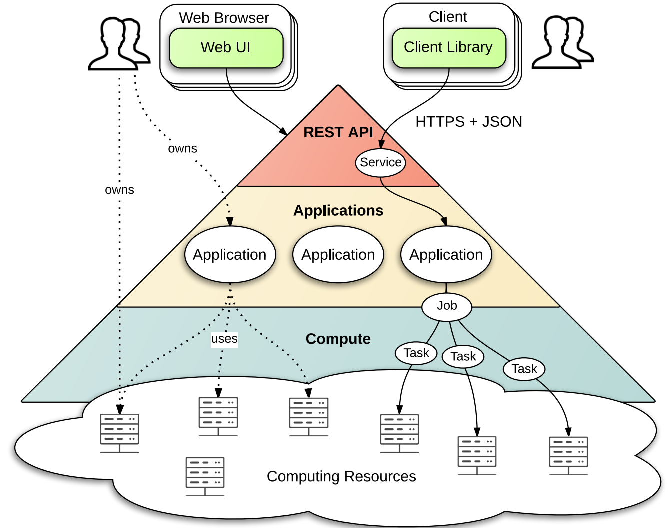

Figure 1. High-level architecture of Everest.

II. Everest Overview

A. Architecture

A high-level architecture of Everest is presented in Figure 1. The server-side part of the platform implemented in the Scala programming language is composed of three main layers: REST API, Applications layer and Compute layer. The client-side part includes the web user interface (Web UI) and client libraries.

Everest is designed around an open programming interface, the so called *REST API*, implemented as a RESTful web service [23]. The API provides access to the platform's functionality, including operations for accessing and manipulating users, applications, jobs and resources. It serves as a single entry point for all platform clients, including the Web UI and client libraries. API requests and responses are served over HTTP with TLS, using JSON as a data interchange format.

Applications layer corresponds to a hosting environment for applications created by users. Applications are the core entities in Everest that represent reusable computational units that follow the common abstract model. An application has a number of *inputs* that constitute a valid request to the application and a number of *outputs* that constitute a result of computation corresponding to some request. It is convenient to think of an application as a "black box" with some input and output ports or as a "function" with some arguments and return values. It is assumed that applications usually process

each request independently from other requests in a stateless fashion.

The described model is generic and applicable to many scientific applications. This model also makes it possible to define a uniform web service interface for accessing such applications [12]. This interface is implemented in Everest as a a part of the REST API. Each application is automatically exposed by Everest as a RESTful web service enabling remote access to the application both via the Web UI and client libraries. An application owner can manage the list of users that are allowed to access the application.

Everest doesn't provide its own computing infrastructure to run applications, nor is it tied to some fixed external infrastructure like grid. Instead Everest enables users to attach to it any external computing resources and to run applications on arbitrary sets of these resources.

Compute layer manages execution of applications on remote computing resources. When an application is invoked via the REST API it generates a *job* consisting of one or more computational *tasks*. Compute layer manages execution of these jobs on remote resources and performs all routine actions related to staging of task input files, submitting a task, monitoring a task state and downloading task results. Compute layer also monitors the state of resources attached to the platform and uses this information during job scheduling.

Web UI provides a convenient graphical interface for interaction with the platform. It is implemented as a JavaScript application that can run in any modern web browser. The Web UI is built directly on top of the REST API. This ensures that users have access to the full range of platform's capabilities when they use the REST API. As a user interface, the Web UI implement additional features, such as wizards and client-side validations.

Client libraries simplify programmatic access to Everest via the REST API and enable users to easily write programs that access applications, combine them in computational pipelines or integrate Everest with third-party tools. At the moment, a client library for Python programming language is implemented.

The technical details regarding the implementation of Everest are out of scope of this paper and can be found in [22].

B. Application Implementation

From the user's viewpoint running an application basically means sending it a request containing input values and waiting for a result containing corresponding output values. For each request Everest performs the following actions as depicted in Figure 2:

1) Authenticate and authorize the client.
2) Parse and validate input values.
3) Create a new job which can be used to track the status of the request and to collect the results.

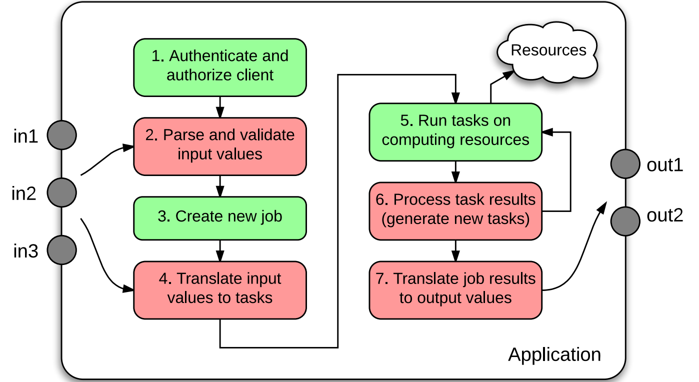

Figure 2. Structure of Everest application.

4) Translate input values to one or more tasks that represent units of computation.
5) Execute job tasks on computing resources bound to the application.
6) Process task results, possibly generating new tasks for execution.
7) Translate job results to output values returned to the client.

Steps 1, 3 and 5 above (colored green in Figure 2) can be implemented in a similar fashion for all applications. However other steps (colored red) are application dependent. In order to simplify creation of applications Everest provides generic implementations of these steps, the so called *application skeletons*, that can be configured for a specific application purpose. This "declarative" approach makes it possible to avoid programming while adding applications to Everest.

In order to add an application to Everest a user should produce an *application description* that consists of two parts:

- *Public information* that is used by clients in order to discover application and interact with it, including specification of inputs and outputs (this information is also used by Everest to implement Step 2 above).
- *Internal configuration* that is used by Everest in order to process requests to the application and generate results, including configuration of the application skeleton, application files and resource binding.

C. Command Application Skeleton

Currently Everest implements a single application skeleton for command-line applications which is suitable for porting to Everest existing applications. This skeleton generates a job consisting of a single task running the ported application.

The internal configuration for the command application skeleton includes:

- A string template for mapping input values to a task command (used in Step 4),

- Input mappings that define how input values map to task input files (used in Step 4),
- Output mappings that define how task output files map to output values (used in Step 7).

D. Parameter Sweep Application

In addition to using application skeletons, it is possible to implement generic applications on top of Everest. At the moment this can be done only by platform developers, however the produced application can be accessed by all users. We have implemented one such application called *Parameter Sweep* that can be used to run arbitrary parameter sweep experiments.

The experiment is described by means of the so called *plan file* similar to the one introduced in Nimrod/G system [24]. This is a plain text file that contains parameter definitions and other directives that together define rules for generation of parameter sweep tasks and processing of their results. Besides a plan file a user can also provide an archive with executables, scripts and input files referred in the plan file.

The Parameter Sweep application parses submitted plan file and uses it in order to generate job tasks (Step 4) and convert task results into an output archive (Step 7). The output of the application is an archive that contains the results of the individual tasks taking into account filtering rules specified in the plan file.

III. Integration with Computing Resources

Everest doesn't provide a computing infrastructure and instead relies on external resources to run application tasks. This choice was motivated by the fact that many researchers already have access to several computing resources ranging from servers and clusters to grids and clouds. Therefore it is more practical to utilize the potential of these resources within Everest instead of building own infrastructure.

A resource can be attached to the platform by any user. As with applications, a resource owner can manage the list of users that are allowed to use the resource to run applications. Everest allows a user to run an application on an arbitrary set of attached resources provided the user has access to them.

A. Compute Agent

Currently the preferred method for attaching a resource to Everest is based on using a developed program called *agent*. The agent runs on the resource and acts as a mediator between it and Everest. This approach has a number of advantages in comparison to plain SSH access such as supporting resources without inbound connectivity (behind a firewall or NAT) and ability to control actions performed by Everest on the resource. However, such approach requires the manual deployment of the agent on each resource. To mitigate this disadvantage the developed agent has minimal software requirements and is easy to deploy by an unprivileged user.

The agent supports integration with various types of resources via adapter mechanism. At the moment the following adapters are implemented:

- *local* - running tasks on a local server,
- *docker* - running tasks on a local server inside Docker containers,
- *torque* - running tasks on a TORQUE cluster,
- *slurm* - running tasks on a SLURM cluster.

In the two latter cases an agent is running on a submission host of the cluster.

The communication between an agent and the platform is implemented through the WebSocket protocol [25]. Upon startup an agent initiates connection with the platform to establish a bidirectional communication channel. This channel is used only for task control and status messages exchanged between Everest and an agent. Task data transfer is performed by an agent via the HTTP protocol. Authentication of an agent is performed by passing a secret token issued by Everest.

In order to attach a resource a user should register the resource in Everest and run the agent on the resource with configuration including the obtained token. After the agent is connected to Everest it starts to send information about the resource state that is displayed in the Web UI.

B. Integration with European Grid Infrastructure

Besides standalone servers and clusters supported via the described agent, Everest also implements integration with the European Grid Infrastructure (EGI). A user can attach as a new resource a specific virtual organization within EGI by uploading a valid proxy certificate. This certificate is used by Everest to submit jobs to EGI on behalf of the user. The interaction with the grid is implemented via the EMI User Interface (UI) which provides a standard set of commands for accessing EGI.

C. Binding Resources to Applications

In order to run an application it should be bound to at least one available resource. Everest implements flexible binding of resources to applications supporting different use cases presented in Figure 3 and discussed below.

Static resource binding means that an application owner configures a static set of resources that should be used by Everest to run the application. In this case the owner implicitly allows application users to run the application on these resources.

This approach is used by many scientific web services that are tied to a fixed computing infrastructure. Static binding is convenient for application users since they don't deal with resources directly. Such approach is also desirable if an application has specific hardware requirements or commercial value. However, in this case the application

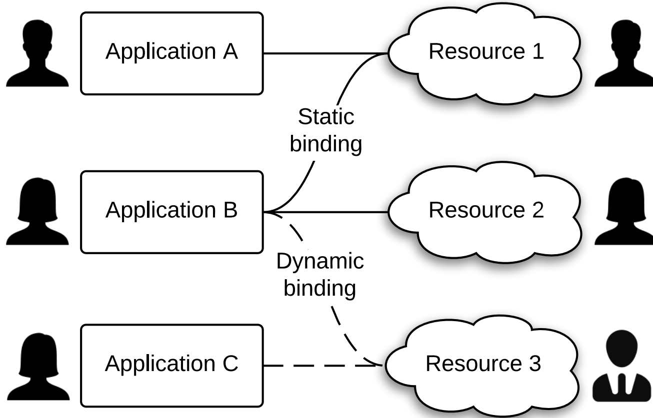

Figure 3. Resource binding models.

owner has to supply computing resources in addition to the application itself. This is a serious barrier for individual scientists wanting to easily share their applications in the form of services with as many colleagues as possible.

Dynamic resource binding means that an application user specifies resources for running her job. This approach eliminates the mentioned barrier by enabling users to run the application on their resources. In this case the application owner need not to supply any computing resources. For the application users this implies the need to have access to some resources. Such resources can be attached to Everest by a user herself or by a resource administrator.

The implementation of dynamic binding is usually problematic for self-hosted applications since there is a lack of trust between the application owner and users for direct delegation of resource credentials. Since Everest both hosts applications and interacts with resources it can play a role of a trusted third party ensuring that an application owner can not obtain a direct access to a user's resource.

Everest implements both static and dynamic resource binding models. It is possible to use both models in an application by providing default static resources and enabling users to dynamically override it with their resources.

Flexible resource binding opens new possibilities, but also brings some challenges discussed below.

D. Application Scheduling

For both static and dynamic models Everest supports binding of multiple resources to an application. In this case the platform should perform automatic scheduling of the application tasks across these resources. From this point of view Everest can be seen as a multitenant metascheduling service since different users can have different sets of resources. The scheduling component of Everest is currently under an active development and we plan to describe it and the underlying approach in the following papers.

E. Enabling Secure and Portable Applications

Dynamic resource binding implies running the application code on an arbitrary resource supplied by the user. This approach has several challenges including the protection of resources from malicious code and making the application portable across heterogeneous resources.

The problem of malicious code is less severe in scientific computing in comparison to consumer software. However by making it easy to publish applications and run them on supplied resources Everest can increase such security risks. Here we propose to follow the common practices by warning users about running untrusted applications on their resources, identification of application owners, digitally signing application files and scanning them with antivirus software. The application owners could also publish the code and files used by the application for examination by users.

Scientific computing resources are heterogeneous both in terms of hardware and software. This makes it hard to take an application that runs on one resource and get it to run on another arbitrary resource without resorting to manual compilation, installing and configuration of the application and its dependencies. We have prototyped two solutions to this problem in Everest based on *application virtualization*.

The first solution relies on Linux containers [26] and Docker [27] to run the application in a disposable container using the configuration provided by the application developer. This approach provides both application isolation and ability to reproduce any custom software environment. Unfortunately, this solution requires that Docker is installed on a resource which is rather uncommon today.

The second solution implements the generation of a portable application package in Everest by using CARE [28]. This is a lightweight virtualization tool that rely on system call interposition to monitor the execution of the application and generate a package including executables, libraries, and data files accessed by the application.

We plan to further address these issues and describe the proposed solutions in detail in the following papers.

IV. Programmatic Access

Running Everest applications via the Web UI is easy and convenient, but it has some limitations. For example, if a user wants to run an application many times with different inputs, it is inconvenient to submit many jobs manually via the web form. In the other case, if a user wants to produce some result by using multiple applications, she has to manually copy data between several jobs. Finally, the Web UI is not suitable if one wants to run an Everest application from his program or some other external application.

To support all these cases, from automation of repetitive tasks to application composition and integration with third-party tools, Everest implements an open programming interface in the form of the REST API. It can be used to access Everest applications from any programming language that

```
import everest

session = everest.Session(
  'https://everest...', token = '6bc...')
appA = everest.App('52b...', session)
appB = everest.App('34a...', session)
appC = everest.App('341...', session)
appD = everest.App('48d...', session)

jobA = appA.run({'a': '...'})
jobB = appB.run({'b': jobA.output('out1')})
jobC = appC.run({'c': jobA.output('out2')})
jobD = appD.run({
  'd1': jobB.output('out'),
  'd2': jobC.output('out')
})

print(jobD.result())
session.close()
```

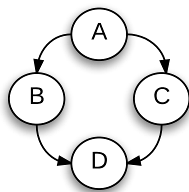

Figure 4. Example of a program using the Python API.

can speak HTTP protocol and parse JSON format. However the REST API is generally too low level for most users, so it is convenient to have ready-to-use client libraries built on top of it. For this purpose a client library for the Python programming language called Python API was implemented.

Figure 4 contains an example of a program using the Python API. It implements a simple diamond-shaped workflow (depicted in the bottom right corner of the figure) that consists of running four different applications - *A*, *B*, *C* and *D*.

At the beginning the program imports *everest* module which implements the Python API and creates a new session by using a *client token*. Each client accessing Everest should present such token with its request in order to authenticate itself.

In order to access the applications the program creates a new *App* object for each application by passing the application ID and the session. The program initiates requests to applications by invoking the *run()* method of the application object. Inputs are passed as a Python dictionary with keys and values corresponding to the input names and values respectively. The *run()* method returns a *Job* object that can be used to check the job state and obtain the result.

Note that the *run()* method doesn't block the program until the job is done and its' results are available. Instead the Python API allows the program to continue its execution after the job is created by performing job submission and monitoring in the background thread.

In the presented example all jobs except *jobA* cannot be submitted to Everest immediately after the *run()* call because they depend on results of other jobs. The program refers to output values of a possibly incomplete job by using the *output()* method of the job and specifying the output name. This method also doesn't block the program until the output value is available. Instead the Python API waits in the background thread until the job is completed, reads the output values and submits dependent jobs as soon as all their inputs are ready. For example, *jobB* and *jobC* will be automatically submitted to Everest after *jobA* is completed.

The nonblocking semantics of the Python API, similar to the dataflow programming paradigm [29], has a number of advantages. It makes it simple to describe computational pipelines without requiring a user to implement the boilerplate code dealing with waiting for jobs and passing data between them. This approach also implicitly supports parallel execution of independent jobs such as *jobB* and *jobC* in the presented example.

After all jobs are created (while possibly not submitted) the program waits for a final result by calling the *result()* method on *jobD*. This method blocks the program until the final job is completed and returns the job result. The result is returned as a Python dictionary with keys and values corresponding to output names and values respectively. Finally the program closes the session by invoking the *close()* function. This terminates all background activities and ensures that the program exits normally.

V. EXPERIMENTAL EVALUATION

We performed experimental evaluation of Everest with all server-side platform components deployed on a single machine. The server running Ubuntu 12.04 has two quad-core Intel(R) Xeon(R) E5620 CPUs operating at 2.4 GHz and 24GB of RAM. A minimal tuning of the system and the Nginx web proxy server was done in order to support a high number of concurrent connections.

A. Applications

In our experiments we have used three applications deployed in Everest.

Sleep is a command application which runs the standard *sleep* command with a specified duration. While it has no practical use, it is convenient for performing various tests, e.g., submitting tasks with a fixed run time or emulating execution of a large number of tasks on a resource without overloading it.

Vina is a command application with runs Autodock Vina [30], an open-source program for molecular docking. It represents a typical scientific application Everest is targeted at. Figure 5 shows a web form for running the Vina application generated by the Web UI based on the application description. This application has nine inputs that correspond to the command line options of Autodock Vina and two

AutoDock Vina

About Parameters Submit Job

Receptor /api/files/5456599a330000760038c6b6/protein.pdbqt + Add file...
rigid part of the receptor (PDBQT)

Ligand /api/files/5456599c330000190b38c6b7/ligand.pdbqt + Add file...
ligand (PDBQT)

Center X 11
X coordinate of the center

Center Y 90.5
Y coordinate of the center

Center Z 57.5
Z coordinate of the center

Size X 22
size in the X dimension (Angstroms)

Size Y 24
size in the Y dimension (Angstroms)

Size Z 28
size in the Z dimension (Angstroms)

Exhaustiveness 8
exhaustiveness of the global search (roughly proportional to time)

Resources The application doesn't have default resources.
Please select at least one resource below to run your job. × test

Request JSON

▶ Submit

Figure 5. A web form for running the Autodock Vina application generated by Everest.

outputs that correspond to the output models and log files. In the command pattern we have set the number of CPUs to use by Vina to 1 since Everest currently doesn't support multicore tasks. The Vina executable has been added to the application files in order to enable execution on resources without pre-installed Vina. As shown in Figure 5 we haven't specified default resources for running the application and enabled dynamic resource binding.

The whole process of porting Autodock Vina to Everest, including test runs, took less than an hour. We have observed similar results while porting to Everest other well-known applications such as POV-Ray, PyMOL and R environment. This demonstrates high productivity of Everest for publication and sharing of scientific applications.

Parameter Sweep is a generic application for running parameter sweep experiments described in Section 2.

B. Performance Tests

To estimate the throughput and scalability of the REST API we have performed repeated job submissions during one minute with varying number of concurrent clients. Job submission was chosen as the most frequent and critical operation among those provided by the API.

Table I
RESULTS OF RAW JOB SUBMISSION TESTS FOR THE SLEEP APPLICATION

Clients	Mean RPS	Mean RT (ms)	95th percentile RT (ms)
1	95	10	16
10	395	24	32
100	1108	88	200
1000	1865	509	967
2000	1819	1051	2154

Table II
RESULTS OF RAW JOB SUBMISSION TESTS FOR THE VINA APPLICATION

Clients	Mean RPS	Mean RT (ms)	95th percentile RT (ms)
1	11	19	37
10	25	76	101
100	59	521	1317
1000	54	3204	9176

The results of raw job submission tests for the Sleep application, including obtained throughput (requests per seconds, RPS) and latency (response time, RT), are presented in Table 1. The platform running on a single server is capable of serving up to a thousand of concurrent clients with acceptable latencies. Note however that these values correspond to the upper bounds of performance, since the job request has minimal size and the job doesn't have input files.

To estimate the effect of input file uploads we have performed the same tests for the Vina application. In this case the job submission is preceded with uploads of two input files (216 KBytes in total) performed sequentially. The results of these tests are presented in Table 2. A large number of concurrent uploads put a stress on disk I/O resulting in decreased throughput and increased latency of job submission calls. This is a potential scalability bottleneck that we plan to address in the future.

To perform the end-to-end testing we have simulated the complete client interaction, starting with job submission, proceeding with periodic polling of the job state and completing with job deletion. This test can be used to estimate the latency introduced to the job processing time, i.e., system overhead. We used the Sleep application with varying job duration as a workload. A single server with the agent was set up to simultaneously run 100 tasks. The job polling period in a client was set to 5 seconds. In the case of 100 concurrent jobs when all jobs can be processed by the resource immediately, the maximum observed overhead is around 10 seconds. In the case of 1000 jobs when jobs are processed in multiple waves, the maximum overhead has increased to 40 seconds due to additional scheduling delays. While such overhead is negligible for typical long-running jobs, it could be improved in the future to better

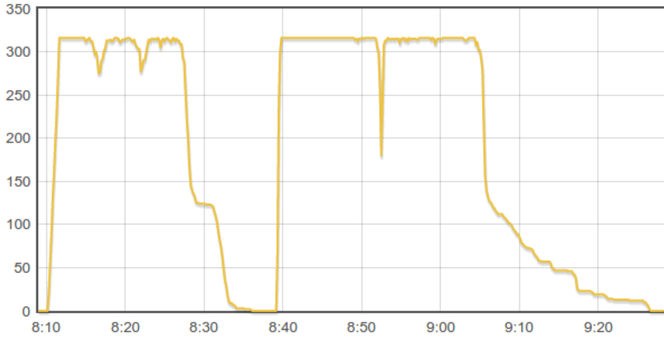

Figure 6. The number of running tasks over the course of two experiments.

accommodate short jobs.

To estimate the effect of many resources attached to Everest the same full job cycle tests were performed with 100 agents running on different servers across several locations. Each agent was configured to simultaneously run 10 tasks. The maximum total overhead observed in tests with 1000 concurrent jobs is 23 seconds. This confirms that Everest can handle relatively large number of attached resources without significant impact on performance.

As a final note, during all of the described experiments we didn't observe any failed REST API requests that demonstrates the robustness of the Everest implementation.

C. Real Application Runs

We have performed several real application runs on the ad-hoc computing infrastructure consisting of three servers and three clusters attached to Everest via agents (316 cores in total). To avoid the noise we disabled all other job submissions on these resources during the experiments. Figure 6 displays the number of running tasks over the course of two such experiments. The first experiment consisted of running 1000 Vina jobs with the ramp-up period of 30 seconds. In the second experiment we used Parameter Sweep application to run a single job consisting of 670 tasks that represented a real application from the geophysics domain.

As it can be seen from the Figure 6, Everest managed to fully utilize the attached resources. The periodic drops of utilization are due to task monitoring and scheduling delays that occur between the task waves. This effect is less significant in the second experiment where tasks have longer run times. However, due to the same reason it suffers from a long tail effect in the end of the experiment. We plan to address these issues in future by optimizing the scheduler component of Everest.

VI. Related Work

A large number of systems have been developed for simplifying publication, execution and composition of scientific applications in distributed computing environments. The presented Everest platform shares many similarities with these systems and relies on established approaches. In this section we perform analysis of prior work and comparison of Everest with related projects to identify major innovations of our platform.

While comparing Everest with related projects it is essential to distinguish between *standalone* and *online* systems. The former are installed and run on user machines, while the latter run on dedicated servers and are accessed via remote interfaces. Online systems such as Everest and related Software-as-a-Service (SaaS) model are becoming mainstream by enabling instant user access and streamlining software updates.

The majority of related online systems are web-based, i.e. focused on implementing remote access to scientific tools and computing resources via convenient web user interfaces. The examples of such systems include grid portals, science gateways and scientific hubs [13]–[15]. While being successful among unskilled users, such systems lack high-level tools for application composition. This drawback was targeted by the second generation of online platforms [16]–[18] that enabled users to compose applications and publish produced workflows as new applications. However, while relying internally on service-oriented approach, these systems do not actually expose applications as web services or provide programming interfaces. In fact, users can work with applications only via system's user interfaces. This limits opportunities for application reuse, composition and integration with third-party tools.

Everest takes a different *API-centric* approach by building the platform around an open programming interface. All applications in Everest are exposed via the REST API thus enabling remote access to these applications from other systems. This makes it possible, for example, to use third-party workflow systems to compose Everest-hosted applications or mix them with other applications.

Programmatic access via web service based APIs is common among modern web applications and cloud computing services. The proliferation of Web APIs [31] has spawned development of mashups [32] that combine data, presentation and functionality from multiple services. Web service composition tools, such as Yahoo! Pipes, provided convenient interfaces for building mashups and making them available to everyone as new services. These lessons are largely ignored in existing web-based scientific environments. One of notable exceptions is the Galaxy platform [16] implementing the REST API that can be used to programmatically access the platform, e.g. run workflows. However, Galaxy doesn't expose individual applications (tools) as services.

A number of service-oriented toolkits have been developed for transformation of scientific applications into web services [10]–[12]. Everest is based on our prior work on MathCloud platform [12], which included all core tools for building a service-oriented environment such as service

container, service catalogue and workflow system. Everest shares many similarities with these toolkits, such as declarative approach for application description and automatic generation of web forms. While existing toolkits represent *standalone* platforms, Everest is *online* platform supporting service development and hosting using PaaS model. This approach has similar benefits as SaaS model discussed above.

Scientific workflow systems [5]–[9] implement high-level tools for composition of applications into computational pipelines. The majority of existing systems are standalone, running both workflow editor and engine on a user's machine. Some systems, such as Taverna [5], support remote execution of workflows on a central server, which is convenient for long-running jobs. Triana [8] introduced support for publication of workflows as web services, while MathCloud [12] featured web-based workflow editor. Recent online platforms, such as Galaxy, implement their own tools for workflow editing, execution and sharing. Everest doesn't implement a workflow editor, however, as was discussed above, it can be integrated with any existing workflow system that supports calling web services. The Python API introduced in Section 4 can also be used to define and execute computational pipelines with Everest.

All discussed systems support integration with distributed computing resources and infrastructures needed to run applications. The range of supported resource types (servers, clusters, grids, clouds) differ from system to system. While standalone systems doesn't impose any restrictions on used resources, users of existing online systems are often limited to a fixed set of resources configured by the administrator. Frequently this shared infrastructure cannot meet increasing demand without implementing limits on resource usage, resulting in delays that users may find unacceptable. A common approach to this problem is to enable users to run a personal installation of the system configured to use local or cloud resources [33]. However, this approach requires additional efforts and expenses from users in order to self-host and upgrade the system.

Everest follows a different approach by enabling any user to attach to the platform external computing resources. This approach helps both to avoid the shared infrastructure bottleneck and efficiently utilize local resources available to individual users. Dynamic resource binding decouples applications from resources enabling users to publish and share applications without having to provide resources. Potentially Everest can be used as a central marketplace for both application and resource providers. To the best of our knowledge, Everest is the first to implement such approach among web-based scientific environments.

VII. Conclusion

We have presented Everest, a new web-based platform for publication, execution and composition of applications running across distributed computing resources. Everest is a complete PaaS solution enabling users with minimal skills to publish and share scientific applications as services. It implements decoupling of published applications from computing resources by enabling users to attach external resources and introducing dynamic resource binding. Everest supports programmatic access to the platform's functionality enabling development of client applications and integration with other systems. We have demonstrated the productivity and performance of current Everest implementation that is capable of serving up to a thousand of concurrent users while running on a single server. The platform is available online to all interested users [21].

Future work will address the remaining challenges and gaps in platform's functionality, such as implementation of advanced scheduling across multiple resources, integration with other types of computing resources and supporting secure and portable applications. We plan to extend the types of applications that users can publish on Everest by providing native support for parallel, many-task and data-intensive applications, as well as supporting publication of composite applications. Last but not least, we also plan to address issues related to transforming Everest into a public computing platform, such as supporting horizontal scalability and providing QoS guarantees.

Acknowledgment

The reported work was partially supported by the Russian Foundation for Basic Research, research project No. 14-07-00309 a.

References

[1] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[2] D. W. Erwin and D. F. Snelling, "Unicore: A grid computing environment," in *Euro-Par 2001 Parallel Processing*. Springer, 2001, pp. 825–834.

[3] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," in *Network and parallel computing*. Springer, 2005, pp. 2–13.

[4] E. Laure, A. Edlund, F. Pacini, P. Buncic, M. Barroso, A. Di Meglio, F. Prelz, A. Frohner, O. Mulmo, A. Krenek *et al.*, "Programming the grid with glite," Tech. Rep., 2006.

[5] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin *et al.*, "Taverna: lessons in creating a workflow environment for the life sciences," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, 2006.

[6] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*. IEEE, 2004, pp. 423–424.

[7] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good *et al.*, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[8] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, "Programming scientific and distributed workflow with triana services," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1021–1037, 2006.

[9] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, *Workflows for e-Science: scientific workflows for grids*. Springer Publishing Company, Incorporated, 2014.

[10] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyanszky, S. Winter, and P. Kacsuk, "GEMLCA: Running legacy code applications as grid services," *Journal of Grid Computing*, vol. 3, no. 1-2, pp. 75–90, 2005.

[11] S. Krishnan, L. Clementi, J. Ren, P. Papadopoulos, and W. Li, "Design and evaluation of opal2: A toolkit for scientific software as a service," in *Services-I, 2009 World Conference on*. IEEE, 2009, pp. 709–716.

[12] A. Afanasiev, O. Sukhoroslov, and V. Voloshinov, "MathCloud: Publication and reuse of scientific applications as restful web services," in *Parallel Computing Technologies*. Springer, 2013, pp. 394–408.

[13] P. Kacsuk, "P-grade portal family for grid infrastructures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 3, pp. 235–245, 2011.

[14] M. A. Miller, W. Pfeiffer, and T. Schwartz, "Creating the cipres science gateway for inference of large phylogenetic trees," in *Gateway Computing Environments Workshop (GCE), 2010*. IEEE, 2010, pp. 1–8.

[15] M. McLennan and R. Kennell, "Hubzero: a platform for dissemination and collaboration in computational science and engineering," *Computing in Science & Engineering*, vol. 12, no. 2, pp. 48–53, 2010.

[16] E. Afgan, J. Goecks, D. Baker, N. Coraor, A. Nekrutenko, and J. Taylor, "Galaxy: A gateway to tools in e-science," in *Guide to e-Science*. Springer, 2011, pp. 145–177.

[17] G. Radchenko and E. Hudyakova, "A service-oriented approach of integration of computer-aided engineering systems in distributed computing environments," in *UNICORE Summit 2012*, 2012, pp. 57–66.

[18] K. V Knyazkov, S. V Kovalchuk, T. N Tchurov, S. V Maryin, and A. V Boukhanovsky, "Clavire: e-science infrastructure for data-driven computing," *Journal of Computational Science*, vol. 3, no. 6, pp. 504–510, 2012.

[19] I. Foster, "Service-oriented science," *Science*, vol. 308, no. 5723, pp. 814–817, 2005.

[20] C. A. Goble, J. Bhagat, S. Aleksejevs, D. Cruickshank, D. Michaelides, D. Newman, M. Borkum, S. Bechhofer, M. Roos, P. Li *et al.*, "myexperiment: a repository and social network for the sharing of bioinformatics workflows," *Nucleic acids research*, vol. 38, no. suppl 2, pp. W677–W682, 2010.

[21] Everest. [Online]. Available: http://everest.distcomp.org/

[22] O. Sukhoroslov and A. Afanasiev, "Everest: A cloud platform for computational web services," in *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014)*. SCITEPRESS, 2014, pp. 411–416.

[23] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

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

[25] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455 (Proposed Standard), Internet Engineering Task Force, Dec. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6455.txt

[26] LXC - Linux Containers. [Online]. Available: https://linuxcontainers.org/

[27] Docker. [Online]. Available: https://www.docker.com/

[28] Y. Janin, C. Vincent, and R. Duraffort, "CARE, the comprehensive archiver for reproducible execution," in *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, ser. TRUST '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:7. [Online]. Available: http://doi.acm.org/10.1145/2618137.2618138

[29] W. M. Johnston, J. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[30] O. Trott and A. J. Olson, "Autodock vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading," *Journal of computational chemistry*, vol. 31, no. 2, pp. 455–461, 2010.

[31] ProgrammableWeb - Mashups, APIs, and the Web as Platform. [Online]. Available: http://www.programmableweb.com/

[32] J. Yu, B. Benatallah, F. Casati, and F. Daniel, "Understanding mashup development," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 44–52, 2008.

[33] E. Afgan, D. Baker, N. Coraor, H. Goto, I. M. Paul, K. D. Makova, A. Nekrutenko, and J. Taylor, "Harnessing cloud computing with galaxy cloud," *Nature biotechnology*, vol. 29, no. 11, pp. 972–974, 2011.