Big Data Platforms as a Service: Challenges and Approach

James Horey, Edmon Begoli, Raghul Gunasekaran, Seung-Hwan Lim, and James Nutaro
Computational Sciences & Engineering
Oak Ridge National Laboratory
{*horeyjl, begolie, gunasekaranr, lims1, nutarojj*}*@ornl.gov*

Abstract

Infrastructure-as-a-Service has revolutionized the manner in which users commission computing infrastructure. Coupled with Big Data platforms (Hadoop, Cassandra), IaaS has democratized the ability to store and process massive datasets. For users that need to customize or create new Big Data stacks, however, readily available solutions do not yet exist. Users must first acquire the necessary cloud computing infrastructure, and manually install the prerequisite software. For complex distributed services this can be a daunting challenge. To address this issue, we argue that distributed services should be viewed as a single application consisting of virtual machines. Users should no longer be concerned about individual machines or their internal organization. To illustrate this concept, we introduce Cloud-Get, a distributed package manager that enables the simple installation of distributed services in a cloud computing environment. Cloud-Get enables users to instantiate and modify distributed services, including Big Data services, using simple commands. Cloud-Get also simplifies creating new distributed services via standardized package definitions.

1 Introduction

Cloud computing has fundamentally changed the landscape of large-scale computing. Users are now able to quickly instantiate simple virtual machines with little effort, and are able to exploit scalable platforms to serve their applications. Although most users tend to think of public cloud infrastructures (e.g., Amazon), private cloud computing has recently gained much traction from both commercial and open-source interests [18]. Tools such as OpenStack are simplifying the process of managing virtual machine resources. This growth in popularity affords the research community with an opportunity to re-visit fundamental computing abstractions. One of the core abstractions that we address in this paper is the notion of an *application* and the components that constitute the application.

In this paper, we take the position that in the context of distributed services, an application should consist of a *set* of virtual machines. Examples of such services include Big Data platforms (e.g., Hadoop [11, 10], Cassandra [16]), and scalable web platforms (e.g., Ruby on Rails [7], Django [5]). Users should interact with the service as a whole (perhaps via dedicated login node), and rarely interact with the underlying virtual machines. In this sense, a single VM is analoguous to a single process in a traditional computing environment. From the user's perspective, the VMs that are used to instantiate the service should be opaque. Even more importantly, users should no longer care about exactly what occurs within a single VM (or even how that VM operates).

To demonstrate the usefulness of this approach, we introduce *Cloud-Get*, a set of tools used to create and manage distributed Big Data platforms. Our approach is inspired by modern Linux package managers (i.e., *apt-get*, *yum*). Package managers enable users to easily search for and install applications. An application package may specify a set of dependencies necessary for the application to function properly. In theory, package managers enable users to easily install applications without worrying about specific libraries, compilation flags, etc. Likewise, Cloud-Get enables users to search for distributed services that reside on a cloud computing environment (either private or public), and to instantiate these services using a set of simple commands. In turn, a Cloud-Get may instantiate multiple, heterogeneous virtual machines (depending on the package description), and also fetch other Cloud-Get dependencies. For example, the HBase package (a key-value store for Hadoop) [3] may require the Hadoop package, ZooKeeper [13] package, etc. Unlike traditional package managers, however, Cloud-Get does not "install" applications. Instead, instantiating a service is more analoguous to *forking* a process. Users

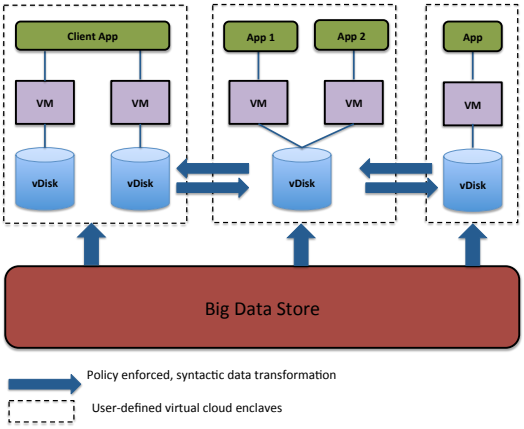

Figure 1: Cloud-Get simplifies the construction of Big Data services that exhibit different organizations.

can instantiate a package multiple times (perhaps with different parameters), and use a unique handle to manage each instance.

Another key element of Cloud-Get is the ability to communicate and transfer data across services. This is analoguous to the notion of a *named pipe*. Cloud-Get provides a set of tools for users to create *data packages* that specify how data should be transferred from one service to another set of services. These packages are designed to be declarative in nature and emphasize the *what* (i.e., data) over the *how* (i.e., protocol) Like the service packages, data packages can be uploaded to a shared repository so that users can search for and reuse existing packages. Since Cloud-Get operates over an IaaS environment, actual data transfer between services can be heavily optimized.

2 Motivation

Cloud-Get is motivated by the observation that while IaaS provides a simple means to create individual virtual machines, there is no analaguous method to instantiate complex, distributed services. For example, a user that wishes to deploy a distributed Big Data platform will need to install various packages on individual VMs (after creating the VMs). The installation process may differ across machines depending on the role of that machine. Once installed, the platform will most likely require some configuration. For example, to exchange *ssh* keys or provide topology information. In addition, different Big Data services will have different organizations. While some systems may prefer local storage associated with several virtual machines, others services may prefer shared storage across machines (Figure 1).

The notion of providing distributed services in a cloud computing environment is not a new idea. Both established commercial entities (e.g., Amazon [2], Microsoft [8]) and recent start-ups (e.g., Heroku [6]) provide users with a simple interface to instantiate particular Big Data tools. Other commercial vendors (e.g., Cloudera) provide packages for Big Data appliances to instantiate in private clusters. These Platforms-as-a-Service (PaaS) are often instantiated over IaaS to ensure scalability, reliability, etc. These platforms, however, are created in a service specific manner. Internal tools used by these companies are most likely specific to the platform they offer. We seek to provide basic abstractions and tools that can be used to create *new* services over IaaS platforms.

Cloud-Get can be viewed as a way to *create* new PaaS and SaaS (Software-as-a-Service) platforms with an emphasis on Big Data services. We envision multiple use cases for this technology. These use cases are motivated by challenges that we've encountered in standing up Big Data systems.

- **Data dissemination** Organizations that need to disseminate data to users usually do so by providing links over the web [1]. However, as datasets become larger, the time and cost of transferring data becomes prohibitive [12]. Also, many datasets are sensitive in nature [15]; controlling access to the data is difficult to accomplish in current settings.

- **Complex architectures** Big Data platforms require a complex set of storage, caching, and analysis components. Instead of treating these pieces as separate things to install, Cloud-Get enables users to package these components together into a single service.

- **Analytic services** By packaging various storage and analysis components together, users can create *analytic* services. Using data-get (the Cloud-Get communication component), users will be able to interact with complex analytic services without considering the underlying software and hardware.

3 Architecture

Users interact with Cloud-Get via RESTful API and a set of command-line tools. Cloud-Get is designed to operate within a cloud computing environment and itself consists of multiple virtual machine components. Figure 2 illustrates the basic architectural components. Users connect to the Cloud-Get management node. This node implements the Cloud-Get API and interacts with the other components (including the cloud computing layer). The management node also implements components to start

[1] http://www.data.gov/

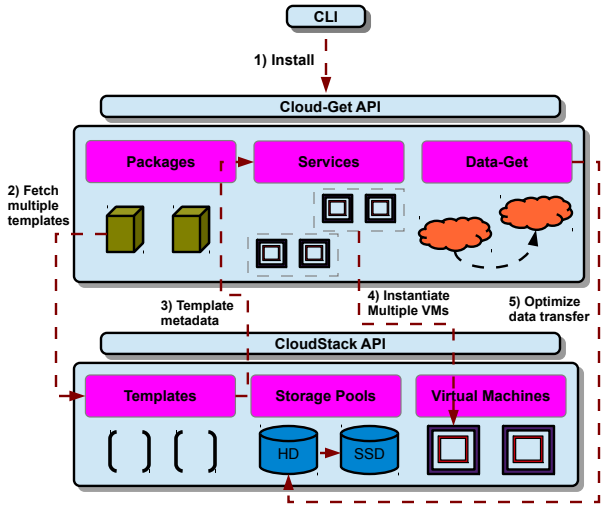

Figure 2: Users interact with the Cloud-Get API via command-line tools. Cloud-Get, in turn, interacts with an underlying cloud management layer.

new services, keep track of running services, etc. Cloud-Get also includes nodes to manage the service and data package repositories. These repositories store *packages* that describe the distributed services. These packages do not actually store the virtual machine templates; instead the repositories interact with the underlying cloud computing layer to manage the images.

3.1 Service Packages

A service package (a.k.a. *cpack*) defines a Cloud-Get application. In the most simple form, a cpack file includes a description of the various VM resources that constitute the service. These include a unique identifier (e.g., *head node*, *slave node*), a template identifier (to identify the VM template), and a set of hardware requirements (i.e., CPU, memory, etc.). The cpack file also includes information regarding external dependencies. Finally, each VM type can specify what data it requires after it starts (i.e., the IP address of a seed node). This data can come from another VM or from an external dependency.

The service package can also include a set of event handlers. Events are generated by the Cloud-Get management tool and are used to control the operation of the distributed service. Event handlers are divided into *package events* and *node events*. Package events operate over the entire service, while node events operate over specific nodes. Within a package event handler, package authors can specify the order in which VM types (including dependencies) are started. Any data that a VM type requires are also explicitly listed in the handler. Package events include:

- **start/stop**: Defines what happens when the service is started or stopped. The handler defines which VM types to start along with any ordering or data restrictions.
- **storage increase**: Defines what happens when the user requests to increase the total storage capacity.
- **node increase**: Defines what happens when the user requests to increase the number of working VMs.

Each VM node must also be able to respond to events (either originating from the package event handler, or from an external client). Node specific events include:

- **configuration update**: Indicates a configuration value has changed (e.g., IP addresses, etc).
- **machine update**: Indicates that the underlying VM has changed (e.g., CPU, memory, disk).
- **custom update**: Package authors can also define their own events specific to their service.

We provide a simple Python library that package authors can include in their VM templates that provides basic event handlers. However, users are free to implement their own event handlers.

3.2 Data Packages

A data package (a.k.a. *dpack*) defines how data is transferred between services. To that end, a dpack is similar in concept to a *named pipe*. However, unlike most pipe semantics, a data package can define what data to retrieve from a particular source and can optionally define expected communication characteristics (i.e., the number of bytes, timeliness constraints). Since the source may consist of an arbitrary service, the dpack does not restrict the user in defining the data extraction method. For example, if the data source consists of a Hadoop service, the data transfer may consist of the output of a MapReduce job. Likewise, the data sink must be able to understand the output of the source. Once a dpack is attached a sink and source, the dpack can operate in either *continuous* or *single-shot* mode. In single-shot mode, the data transfer takes place once. However, in continous mode, the data transfer occurs regularly (defined by the user). By attaching a set of data packages to services, the use can define an explicit data flow through the system.

Note that the use of data packages does not preclude any service from communicating with other services via other means. Instead, data packages offer a simple way to copy over bulk data and chain services together. In addition to this, the use of data packages enables potential optimizations. For example, data co-located on shared disks can be locally copied. For read-mostly workloads,

the system can employ copy-on-write semantics leading to further bandwidth improvements.

3.3 Usage

We anticipate two types of users: package maintainers and clients. Package maintainers create and maintain the Cloud-Get packages. Creating a package consists of creating the appropriate virtual machine templates, populating the VM with software and management scripts, and creating a package description. This process is largely performed on the cloud computing platform. Clients interact with existing packages via command-line tools or scripts. Clients may also create data packages to define how data flows between services. An example of the Cloud-Get management API is shown in Table 1.

4 Implementation and Challenges

We currently have a prototype implementation of Cloud-Get running on top of the CloudStack platform [2]. CloudStack provides Infrastructure-as-a-Service. The various Cloud-Get services (the service manager, package repository, and data manager) are instantiated as a set of virtual machines managed by CloudStack. Cloud-Get functionality is not specific to CloudStack, and we anticipate offering additional backends in the future (e.g., OpenStack[3], Amazon EC2, etc.). In our current implementation, we do not provide any tools to simplify the creation of cloud-get packages, although this is something we hope to address in the future. Specifically we hope to support operations such as *merge* to combine multiple, existing cloud-get packages.

Like many IaaS platforms, CloudStack includes many opportunities for low-level control of virtual resources. For example, CloudStack enables VMs to allocate storage from a variety of *storage pools*, including shared (e.g., iSCSI) and local (i.e., attached directly to a physical host). Although the disk image is treated the same by the VM, different storage pools have different tradeoffs. For example, a single VM guest may have lower latency access to local storage. However, as the number of co-located VMs are increased, a more capable, shared disk may result in higher aggregate bandwidth. Also, different storage pools may have different limitations. For example, CloudStack does not normally allow users to allocate virtual disks over local storage (since these disks can only be attached to VMs residing on a specific host). Optimally allocating storage across a set of virtual machines is an ongoing research challenge.

Although CloudStack offers a relatively comprehensive API, it does hide certain hypervisor options. For example, some hypervisors enable users to specify the write buffer cache strategy (i.e., none, write-through, write-back). We have found that write-back consistently offers the best performance for large, sequential writes and can perform even better than writes executed in the host (data not shown). Write-through consistently exhibits the worst performance (nearly a 30x difference). Although it is tempting to simply choose write-back, write-back has the highest likelihood of losing data since the data is cached twice (once on the VM and the host). Exposing these options in a standard manner would greatly simplify the implementation of Cloud-Get.

We are also optimizing the co-scheduling of virtual machines. CloudStack and other IaaS platforms enable the use of pluggable virtual machine schedulers. Often, these default schedulers assume independence between virtual machines, though they consider performance interference among virtual machines [17]. However, with Cloud-Get, sets of virtual machines are designed to work together: their execution might be correlated to each other; and they may communicate heavily internally with more sporadic communication with external services. Co-locating guests on a single host will often result in better network performance between those guests (since communication never leaves the host). Co-locating guests, however, may also lead to poor disk performance with wide variance [9, 19]. Preliminary results on our CloudStack infrastructure indicate a drop from 133 MB/s (single VM performing sequential read of a 1GB file) to 109 MB/s when 11 VMs are simultaneously performing sequential reads of the same file. In addition, both sequential and random access of large files without buffer caching can lead to linear slowdown with large read variance (up to 64%) [4] Managing these communication and disk I/O tradeoffs to ensure the highest performance remains a critical challenge.

5 Discussion and Conclusions

Cloud-Get is a system to simplify the installation and management of distributed services. Cloud-Get leverages cloud computing platforms to dynamically instantiate sets of virtual machines that work together. In addition, Cloud-Get features powerful data transfer capabilities between services to enable users to create explicit data flows. Unlike other management software (Puppet [14], Chef [4] Cloud-Get only interacts with virtual machines. Users are not expected to interact directly with software packages or libraries. Such an approach simplifies packaging and management, since all library dependencies, language frameworks, etc. can be encapsulated

[2] http://cloudstack.org

[3] http://openstack.org/

[4] We experimented with a node with a 12-core CPU, 48GB RAM, and two 2TB 7200 RPM SATA drive. Each VM had one vCPU and 4GB RAM.

start(service, nodes, capacity)	Starts a new service and returns a unique instance handle
stop(instance)	Stops the instance
modify_nodes (instance, nodes)	Increases the number of nodes associated with the instance
modify_storage(instance, capacity)	Increases the storage associated with the instance

Table 1: Cloud-Get service management API.

within a single virtual machine. At Oak Ridge National Laboratory, we are using a prototype version of Cloud-Get to manage our data analysis infrastructure consisting of Hadoop and a variety of distributed storage tools.

We believe that the cheap provisioning of virtual machines, enabled by cloud computing, enables us to re-visit core computational abstractions. Virtual machines should serve the same purpose as a *process* for distributed applications. Much like how an application might use multiple processes to improve performance, we envision distributed applications automatically spawning VMs to improve scalability. In this view, distributed services consisting of virtual machines would consitute a core component of a cloud-computing operating system. Our approach shares common goals with prior proposals [20], although our approach is coarser-grained. We believe that working towards finer-grained control over the underlying VM resources will constitute an important step towards our shared vision [1]. To that end, we expect Cloud-Get to continue playing an important role in realizing this vision.

6 Acknowledgements

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. This research has been made possible by a Lab Directed Research and Development grant at Oak Ridge National Laboratory.

References

[1]

[2] Amazon web services. `http://aws.amazon.com`.

[3] Apache hbase. `http://hbase.apache.org`.

[4] Chef. `http://wiki.opscode.com/display/chef/Home`.

[5] Django. `http://www.djangoproject.com`.

[6] Heroku. `http://www.heroku.com`.

[7] Ruby on rails. `http://rubyonrails.org`.

[8] Windows azure. `http://www.windowsazure.com`.

[9] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Communications of the ACM 53*, 4 (2010).

[10] BORTHAKUR, D. The hadoop distributed filesystem: Architecture and design. `http://hadoop.apache.org`.

[11] DEAN, J., AND GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM 51*, 1 (Jan. 2008), 107–113.

[12] DEELMAN, E., AND CHERVENAK, A. Data management challenges of data-intensive scientific workflows. In *IEEE International Symposium on Cluster Computing and the Grid* (2008).

[13] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIX Association.

[14] KANIES, L. Puppet: Next-generation configuration management. *The USENIX Association Newsletter 31*, 1 (2006).

[15] KIFER, D., AND MACHANAVAJJHALA, A. No free lunch in data privacy. In *ACM SIGMOD/PODS Conference* (2011).

[16] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (Apr. 2010), 35–40.

[17] LIM, S.-H., HUH, J.-S., KIM, Y., AND DAS, C. R. Migration, assignment, and scheduling of jobs in virtualized environment. In *3rd USENIX Workshop on Hot Topics in Cloud Computing* (2011).

[18] MICROSOFT. Microsoft private cloud. Tech. rep., 2012.

[19] WACHS, M., XU, L., KANEVSKY, A., AND GANGER, G. R. Exertion-based billing for cloud storage access. In *Proceedings of 3rd USENIX Workshop on Hot Topics in Cloud Computing* (2011).

[20] ZAHARIA, M., HINDMAN, B., KONWINSKI, A., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. The datacenter needs on operating system. HotCloud'11.