From PaaS — to the Cloud Operating System

Nadav Har'El
IBM Research — Haifa
nyh@il.ibm.com

Abel Gordon
IBM Research — Haifa
abelg@il.ibm.com

Abstract

Currently, there are two competing factions of cloud computing — IaaS and PaaS. With IaaS, application developers work with a number of virtual machines, each with its own operating system. PaaS replaces this by a collection of cloud-wide services which applications can use.

In this paper we make, and justify, the analogy of PaaS being the cloud's *operating system*. We claim that as PaaS is maturing, every cloud application will use PaaS services, making the application much easier to write, more flexible and often more efficient. We explain why PaaS is better implemented as part of the cloud system, not the application, and is therefore better thought of as an operating system than a library. We predict that writing new cloud applications on IaaS (without PaaS) will soon be just as uncommon as applications that run on a computer without an operating system.

We explore the details of this analogy, explore how PaaS will need to evolve to become more effective in its new role of the cloud operating system, and propose new directions for operating system research in the cloud era.

1 Introduction

Cloud computing is becoming an increasingly important technology on the Internet. It allows providers of online services to pay exactly for the amount of computing resources they need, when they need it. Cloud computing reduces the barrier-to-entry for new online services. It allows services (*cloud applications*) to scale from a handful to millions of users and to cope with fluctuating demand without needing to plan or buy for the peak load.

Amazon's EC2 (Elastic Compute Cloud), announced in 2006, was the first cloud computing solution available to online service developers. Its model, later dubbed *IaaS* (Infrastructure-as-a-Service), was to present application developers with virtual x86 machines (VMs). The developer gets as many VMs as desired, each running a traditional operating system of his choice such as Linux or Windows, and runs on it his own application code.

x86 VMs give the application developer full control, but once the application scales to many VMs, writing the application to efficiently deploy and utilize all them is very difficult. For example, when Amazon needed a highly scalable and efficient database for its internal applications, they designed Dynamo [2], a new breed of key-value store. Since most of Amazon EC2's customers were unable to build, let alone design, such a key-value store themselves, Amazon soon started offering a scalable database service, called *SimpleDB*, to their customers. Afterwards, Amazon and the other cloud providers started to add more and more new services for their IaaS customers' use.

Some of these additional cloud services also include mechanisms to run application *code*. For example, Google's App Engine allows application developers to run code in high-level languages such as Java or Python, in response to HTTP requests. Amazon's Elastic MapReduce is a MapReduce [1] framework for scalably running user code on vast amounts of data. These code-running mechanisms mean that applications can be run on the cloud without any of the classic x86 VMs. This new approach to building cloud applications was dubbed *PaaS* (Platform-as-a-Service) — i.e., the cloud is a new platform on which to build applications.

Much of the contemporary discussion about IaaS and PaaS view these as two competing alternatives, each with pros and cons. IaaS is viewed as very expressive, but difficult to use, while PaaS is viewed as being limited in expressiveness, and easy to use.

We begin this paper by arguing (Sections 2 and 3) that PaaS can, and should, be considered the cloud's *operating system*. We explore this analogy, and conclude that soon, writing cloud applications on IaaS (without PaaS) will be similar to programming without an operating system — tedious with very little potential gains in terms

of expressiveness or performance, and therefore rarely done.

We continue in Section 4 to make predictions, and propose new research directions, on how PaaS will need to evolve to become more effective in its new role of the cloud operating system.

Several recent papers [10, 14] also recognized the usefulness of a cloud-wide operating system of the type we describe. The main contributions of this paper are: (1) Our analysis in Section 3 of the parallels between PaaS and an operating system, and (2) Our predictions and proposals in Section 4, laying down new directions for operating system research in the cloud era.

2 Why a Cloud Operating System

In a IaaS cloud the application developer needs to install a traditional operating system (such as Linux or Microsoft Windows) on each VM. But the features of these single-machine operating systems become increasing irrelevant for the cloud application developers, who do not use local devices, local filesystems, multiple local users or sometimes even processes.

Instead, cloud-wide services are becoming increasingly important. As mentioned above, development of cloud applications can be greatly simplified by using pre-built PaaS services, such as using a good existing key-value store implementation instead of developing one from scratch. An operating system simplifies programming on a classical computer platform in much the same way — applications can use simple system calls to write to files, instead of implementing a filesystem from scratch.

Cloud applications do not necessarily need to rely on the cloud to provide these services. For example, if an application on an IaaS cloud needs a key-value store, it can deploy a few more VMs running a key-value store implementation of its choice, e.g., the open-source Cassandra [9]. This can be described as *PaaS-over-IaaS*.

However, we argue that there are several important reasons why it is better that the cloud provider deploys the PaaS services, not each tenant:

1. For many cloud tenants, even knowing which PaaS implementation to deploy and how to set up the VMs implementing it, is beyond their expertise. Why should a person who knows how to program simple applications be expected to install operating systems and complex software packages?

2. Some PaaS services need to run on special hardware. For example, an object store service such as Amazon's S3 or Openstack's Swift needs to run on machines with persistent local disks. Moreover, it is the provider, not the tenant, who is aware of the details of the underlying hardware, and can better optimize the software for it.

3. The PaaS services can be run without an hypervisor, improving their performance.

   In other words, we disagree with the common approach of running PaaS on top of IaaS. Quite the contrary — we argue that IaaS should just be another PaaS service, allowing tenants to run legacy workloads in virtual machines.

4. When tenants deploy x86 VMs to implement a PaaS service, the cloud provider is not able to switch to non-x86 architectures or other non-standard hardware, even when such hardware might be cheaper or more efficient for running a particular service.

5. A PaaS service deployed by the cloud provider can be more effectively shared between tenants, reducing costs. This is very important for small tenants (where, for example, a full VM for its key-value store might be an overkill), but in many cases can also be useful for larger tenants. CDNs are a good example — tenant's data is replicated in many geographical locations, but the tenant doesn't need a full VM in each location.

When individual tenants deploy PaaS components as part of their application, PaaS can be likened to a library, or a library operating system [3]. However, as we just explained, it is better that the cloud already includes these PaaS services, and in that case they resemble a traditional operating system, which all applications use.

3 The Operating System Analogy

In this section we explore various aspects of our analogy of PaaS services to those of a traditional operating system. In the next section (Section 4), we will look at what insights we can get from this analogy, and how PaaS should evolve to be better at its new role of cloud operating system.

System calls: Operating systems traditionally provide applications with *system calls*, providing various services such as reading and writing disk files, doing inter-process communications, and communicating over the network.

For PaaS applications, most of these needs are met by PaaS requests, which we will call *cloud calls* in analogy with system calls. For example, cloud parallels to file access include object stores (such as Amazon's S3 and Microsoft's Azure Blob Storage), block stores (such as Amazon's EBS), and various distributed key-value stores (such as Amazon's SimpleDB and DynamoDB). Cloud

parallels to inter-process communication include, for example, message queues such as Amazon's SQS.

Cloud calls are not serviced on the local machine, but rather by some other node in the network, so the cloud call mechanism will normally involve reliably sending a network message, using REST [4] as in Amazon, or some other RPC mechanism. However, this internal mechanism will normally be hidden from the application developer — who will instead get a programmatic API, a library, for the programming language of choice. For example, Google's PaaS has APIs in Java, Python and Go. Amazon's PaaS services have REST and SOAP APIs which can be used from any language, with example code provided for several common languages. Similarly, wrapping system calls in an API is also a common practice in traditional operating systems — e.g., Unix [16] has a C API for its system calls, documented in section 2 of its programmer's manual.

The system call API is important, because it allows the operating system's implementation to change, while applications continue to run unmodified. For example, in Linux much of the implementation of the scheduler, file systems, network stack, etc., changed over the years, but it still runs Linux applications because the system calls haven't changed. Similarly, a cloud might change its underlying choice of processor architecture, hypervisor, storage implementation, etc., but the users who use the cloud operating system APIs needn't care.

Processes: Operating systems run application code encapsulated in *processes*. The OS offers scheduling services (when to run which process, on which CPU, etc.), execution services (starting a new process running certain code), isolation between different processes — and on the other hand, inter-process communication.

A cloud application also needs to run application code, including both long-lived background processes (e.g., data analysis) and short-lived execution of code in response to an event (e.g., an incoming HTTP request).

The cloud operating system will provide a notion of *cloud processes* to encapsulate these threads of execution of application code. Just like a traditional operating system makes it unnecessary for an application developer to worry about where or when to run each process, so should the cloud operating system do with cloud processes: The application developer doesn't need to worry on which machine these run, or how many of them run in parallel.

Existing PaaS clouds suggest how cloud processes would be used. Google App Engine "Runtime Environment" runs cloud processes (application-specified code) in response to HTTP requests; There can be many of these requests running in parallel, on many different machines, without the application knowing or caring. Another Google App Engine feature, "Push Queues", allows the application to queue processes for execution — these processes will eventually be run, at desired rate and order, on potentially many different machines. The system decides where to run each process, and ensures it really does run to completion (retrying it if necessary in case of crashes). The Heroku PaaS solution revolves around cloud processes, which they call "dynos".

Programs and execution: An operating system also handles program execution, i.e., starting a new process running a given piece of application code.

The IaaS implementation of cloud processes (namely, heavy long-lived VMs complete with a traditional operating system) is too heavy for many use cases. Much lighter methods of safely running compiled user code exist, for example OS-level virtualization [8], system-call interposition [6], and limited instruction sets [18].

Even better, to allow the cloud provider to experiment with different processor architectures and not be limited to the venerable x86 architecture, the cloud operating system might encourage, or even mandate, that application code be written in a high-level language. We call such code a *cloud executable*. Google App Engine, for example, can only execute code in Java, Python or Go. It has a further optimization, where loading a certain application code on a machine where it recently ran can reuse an already loaded process (known as a Servlet in the case of Java) instead of starting a new one.

The idea of an operating system which can only run user code written in high-level languages is not new to the cloud. The Singularity operating system [7] can only run CLR [11] bytecode, and while Android can run native code, it strongly encourages user code to be written in Java compiled to Dalvik bytecode. However, we argue that cloud executables needn't be bytecode, and can be the original source code — with the cloud operating system deciding what to do with this source code, e.g., interpreting it or compiling it into bytecode or native code.

Tenant isolation: One of the primary goals of operating systems is to share resources, such as compute power, storage capacity and network bandwidth, between different users and applications. The original paper on UNIX was titled "The UNIX Time-Sharing System" [16] to emphasize this fact.

Similarly, PaaS emphasizes multi-tenancy, as multiple applications and application owners (*tenants*) access the same services, but are isolated from each: Processes running for one tenant cannot access the processes, or the data, of a different tenant, and application performance is isolated in the sense that one application cannot negatively impact the performance of a other applications.

External network: Another function of traditional operating systems is to relay inbound network data to the particular processes which own the relevant connection.

PaaS clouds have a similar feature, where each end-user's connection is relayed to some cloud process on some machine. The cloud tracks these connections, and does load balancing and scaling as necessary.

File system: Operating systems offer *file systems* to provide applications with convenient and secure (tenant-isolated) access to persistent storage.

Similarly, PaaS provides storage services for cloud applications. The API of these services does not have to resemble the familiar file API of traditional operating system. For example, object store services (such as Amazon's S3 and Microsoft's Azure Blob Storage) allow storing and retrieving whole files, but not modification to existing files, as this is enough for many cloud applications and easier to make more scalable and reliable than the traditional filesystem API. Various cloud database services (such as Amazon's SimpleDB and DynamoDB) are also filesystem replacements, and there are additional storage APIs such as the Google File System [5] with different use cases in mind (in this case, temporary storage for MapReduce).

Inter-process communication: Another common function of operating systems is *inter-process communication* (IPC), allowing different processes, which are normally isolated, to efficiently communicate when required.

On the PaaS cloud, processes come and go, and run on thousands of different machines, so PaaS usually encourages asynchronous and anonymous, but reliable, IPC: one process leaves instructions for other, unknown, processes. One common example of cloud IPC are message queues, such as Amazon's Simple Queue Service (SQS). MapReduce [1] uses temporary files stored in a Google File System [5] as a form of IPC between cooperating processes working on the same task.

We argue that PaaS IPC services should be the only intra-cloud communication mechanism available to cloud executables. These would not be aware of the low-level network layers such as IP, just like UNIX executables are not aware of the internal mechanisms used to implement pipes. We believe that the recent trend of network virtualization [12] is an artifact of IaaS, and not necessary for PaaS.

Daemons: A complete operating system typically includes, in addition to its kernel, various *daemons*, or server processes, providing additional services such as an HTTP server or a mail server. PaaS clouds also provide similar daemons. For example, Google App Engine "Runtime Environment" runs application code in response to HTTP requests, much like a traditional HTTP daemon. Amazon's Simple Email Service (SES) is an email sending service, much like a traditional mail delivery daemons.

Accounting: Operating systems often track the amount of resources consumed by each user, such as CPU time (in application code and in the kernel), disk space and accesses, and so on. The PaaS cloud similarly tracks resources used by different tenants, and this information is later used for billing the tenant.

4 Predictions and Recommendations

In this section, we consider how existing PaaS solutions need to evolve, in order to be more effective in their new role of cloud operating systems. We also suggest new directions which we believe operating system research should take in the cloud era.

Standardization: Today, each cloud provider presents to application developers a very different set of PaaS services. This makes it exceedingly difficult to write portable PaaS-based cloud applications, leading many developers to prefer developing for IaaS.

Because we believe that as PaaS will win over IaaS (for reasons we explained in this paper), we predict an inevitable move towards standardization of PaaS services and PaaS APIs. Such standardization may come in several forms: (1) The appearance of open-source PaaS implementations, which many providers will use; (2) Providers will agree on a standard PaaS API, doing for the cloud operating system what Posix did for UNIX; and (3) The appearance of compatibility or abstraction layers, allowing writing cloud applications that will work on many different PaaS implementations.

A cloud kernel: Most operating systems try to keep a clear separation between their *kernel*, and a set of user-space utilities and daemons which provide additional useful services, but are written on top of the kernel's APIs (namely, the system calls), just as any other application. For example, an HTTP daemon is just a user-space program calling system calls like socket(), bind(), accept(), read() and write(), and any user might run an alternative HTTP daemon, without modifying the kernel.

In most existing PaaS implementations, this distinction is lost. They provide a hodgepodge of different services, with no clear distinction of what constitutes the basic set of services (the cloud OS kernel) on which the other services, as well as the user applications, are built.

Cloud MapReduce [10] is a good demonstration that given a few basic services in Amazon's cloud (S3 object store, SQS queue service, and threads on EC2 VMs), a MapReduce feature can be easily built (in only 3,000 lines of code). This means that MapReduce can be left out of the cloud operating system kernel, and instead may be run by individual tenants over the more fundamental services of the cloud operating system.

Similarly, both Amazon and Google have mail-sending PaaS services, but these could be built using

other simpler components such as Push Queues which provide the required scalability and reliability — but not the exact policy and the details of the SMTP protocol.

We believe that more research is necessary to determine a good set of fundamental PaaS services which would constitute the kernel of tomorrow's cloud OSs.

Expressive power: One of the oft-mentioned downsides of PaaS, compared to IaaS, is its lack of expressive power: PaaS is great when it provides the services you need, but when it doesn't, you're stuck.

Limiting applications' expressiveness—making important things not doable without being part of the operating system—is the hallmark of a *bad* operating system. A good operating system provides a rich set of broad services, on which any application can be built. The expressiveness of modern operating systems is what caused bare-metal (no OS) programming to all but disappear, but the expressive power of PaaS still needs to improve for it to be a better cloud operating system.

As an example, Google App Engine allows a cloud application to run some code on HTTP requests. But what if the application wants to serve a newly invented protocol, not HTTP? The cloud OS should have more general cloud calls for listening for incoming requests, and processing them in a scalable way. The HTTP-specific daemon would be written using these cloud calls, and application developers can opt to use a different one.

Low latency: To behave like system calls, *cloud calls* must be always available, and have as low latency as possible. Existing PaaS services such as Amazon's Dynamo [2] have indeed been designed for high-availability and fault tolerance. The Dynamo design also emphasizes predictable low latency, ensuring low latency not just at the average case, but also in the 99.9th percentile (when 200ms latencies are reported). While these numbers are low enough for many applications, with more research into hardware and software, they could be made much lower. In their vision paper, Rumble et al. [17] explain that *It's time for low latency*, and that $5 - 10\mu$s remote procedure calls across the data center are desirable, and achievable with some improvements to existing network and OS technology, and by relying on RAM instead of magnetic disks [13]. Other papers explored different options for low-latency persistent storage, such as flash and phase-change memory.

Heterogeneous hardware: Cloud performance can be further improved by co-designing the hardware and the software, deploying each cloud OS service on machines especially suited for this service. While IaaS clouds were limited to a few architectural choices for which their customers could build VMs, when the cloud provider builds both the hardware and the PaaS implementation, it can make much larger deviations off the beaten path, and experiment with different types of processors offering better performance or using less energy, or with unusual hardware configurations — with applications unaffected.

Examples abound in existing PaaS services, e.g., Rackspace's Swift object store which runs on machines with many local disks [15], and Amazon Glacier, a write-once-read-rarely storage service working on special hardware to lower maintenance costs of unread data. We believe that much more research is required in this area, on how to co-design optimal software and hardware for different kinds of services.

Application execution: While IaaS started with heavy machine-virtualization (long-running VMs complete with their own operating systems), PaaS will continue to develop lighter mechanisms for running application code, based on new techniques of OS-level virtualization and application-level virtualization. We also expect to see research on ways of effectively and reliably utilizing small short-lived processes on the cloud, such as Google's ideas of Push Queues and MapReduce.

We believe that most of the code to be run will likely be written in high-level languages, not compiled code for a specific processor, but because of the proliferation of different languages, and the need to write common code and libraries for all of them, we believe the future lies with virtual machines similar to Microsoft's Common Language Runtime [11], capable of running (and mixing) code compiled from many different high-level languages, or other approaches allowing mixing source code from different languages.

Security and isolation: As mentioned above, the cloud operating system must provide secure isolation between different tenants or applications. While IaaS offers well-understood security guarantees of hypervisors, we feel that the security risks of PaaS services shared by different tenants are much less understood and researched, and we hope to see more research on models and techniques for hardening PaaS services against data leaks and privilege escalation attacks caused by software bugs.

5 Summary

We began this paper by making and justifying the claim that PaaS is becoming the cloud *operating system*, and using it makes cloud applications easier to write, more efficient and more scalable. Just like most traditional software is written on an operating system, so will most cloud applications be written over PaaS.

We continued to identify areas in which state-of-the-art PaaS solutions still need to be improved in order to better fulfill their new role of cloud operating system, and made predictions on which directions PaaS research and development will take over the next few years.

References

[1] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *6th Symposium on Operating Systems Design and Implementation (OSDI)* (2004).

[2] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM symposium on operating systems principles (SOSP)* (2007), pp. 205–220.

[3] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE JR., J. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP)* (1995), pp. 251 –266.

[4] FIELDING, R. Representational state transfer (REST). *Architectural Styles and the Design of Network-based Software Architectures* (2000), 120.

[5] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The Google file system. In *Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003).

[6] GUO, P. J., AND ENGLER, D. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the 2011 USENIX Annual Technical Conference* (June 2011).

[7] HUNT, G., AND LARUS, J. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review 41*, 2 (2007), 37–49.

[8] KAMP, P., AND WATSON, R. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Network Engineering Conference (SANE)* (2000).

[9] LAKSHMAN, A., AND MALIK, P. Cassandra a decentralized structured storage system. *Operating systems review 44*, 2 (2010), 35.

[10] LIU, H., AND ORBAN, D. Cloud MapReduce: a mapreduce implementation on top of a cloud operating system. In *Proceedings of 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (2011).

[11] MEIJER, E., AND GOUGH, J. Technical overview of the common language runtime. `http://research.microsoft.com/en-us/um/people/emeijer/papers/clr.pdf`.

[12] MUDIGONDA, J., YALAGANDULA, P., MOGUL, J., STIEKES, B., AND POUFFARY, Y. Netlord: a scalable multi-tenant network architecture for virtualized datacenters. *SIGCOMM-Computer Communication Review 41*, 4 (2011), 62.

[13] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ONGARO, D., PARULKAR, G., ET AL. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *Communications of the ACM 54*, 7 (2011), 121–130.

[14] PIANESE, F., BOSCH, P., DUMINUCO, A., JANSSENS, N., STATHOPOULOS, T., AND STEINER, M. Toward a cloud operating system. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium Workshops* (2010).

[15] RACKSPACE. OpenStack Swift deployment guide. `http://docs.openstack.org/developer/swift/deployment_guide.html`.

[16] RITCHIE, D., AND THOMPSON, K. The UNIX time-sharing system. In *Proceedings of the Fourth Symposium on Operating System Principles (SOSP)* (1973).

[17] RUMBLE, S., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. It's time for low latency. In *USENIX Workshop on Hot Topics in Operating Systems (HOTOS)* (2011).

[18] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of 2009 IEEE Symposium on Security and Privacy* (May 2009).