Simple Workload & Application Portability (SWAP)

Sam Johnston
Open Cloud Initiative
Email: sam.johnston@opencloudinitiative.org

***Abstract*—Cloud computing (the delivery of computing as services rather than products) creates new challenges with respect to openness. Existing approaches such as open source licensing of software products have proven inadequate in a service-oriented world; attempts to address the limitations by closing the "service provider loophole" have failed to gain traction. Similarly, multiple providers are yet to fully adopt a single, complete cloud infrastructure management interface.**

In this paper a simple "lowest common denominator" approach to cloud portability is proposed, modeled after the success of Internet email and the Simple Mail Transfer Protocol (SMTP), and based on the ubiquitous HyperText Transfer Protocol (HTTP). The proposed approach is suitable for all workloads and supports any format specified elsewhere and identified by Internet media (MIME) type (e.g. virtual machines, databases, queues).

By addressing only the functionality absolutely required for portability, leaving the management interface out of scope, we can deliver an API which is compatible with all cloud stacks while avoiding constraining their functionality.

Index Terms - Cloud computing; IaaS; infrastructure; PaaS; platforms; SaaS; applications; software; open source; standards; interoperability; portability; HTTP; SMTP; SWAP.

I. Introduction

Simple Workload & Application Portability (SWAP) is a proposed protocol for migrating arbitrary workloads between servers (including public and private cloud, legacy infrastructure and developer workstations). It is a novel approach to the problem in that it tackles only the functionality absolutely required for workload portability, making it simple and easy to adopt, and therefore increasing the probability of any specification becoming a standard. This would break down the silos, in the same way that SMTP enabled mail systems to communicate with each other natively rather than via gateways decades ago.

The proposal leverages existing de facto standards (HTTP, TLS, JSON, etc.) rather than creating an entirely new protocol from scratch (as was done with SMTP), thereby enabling trivial implementations to deliver useful functionality in as few as 1-2 lines of code, using standard tools (curl, wget, etc.) and libraries (Python http.client, Java java.net.HttpUrlConnection, etc.). These existing standards are used as intended in that we are not creating another layer on top of the stack, rather leveraging the existing metadata channel (HTTP headers) to carry information that otherwise would have required a new envelope format (e.g. SOAP).

II. Background

Cloud computing services differ from traditional mainframe and client-server approaches in that the software itself is typically operated by a service provider rather than the end user themselves. As such, there is no software to install and run; users leverage the features of the software via user (web) and machine (API) interfaces without ever having to be in possession of it. There are many advantages to this approach, which are primarily derived from multi-tenancy; cost sharing and economies of scale significantly reduce both capital and operational costs; at "web scale" the cost per server can be reduced by an order of magnitude or more.

However the cloud service delivery model also introduces risks, particularly with respect to the security of data stored by third parties. Typically we focus on confidentiality, but availability and integrity are also critically important; users need to ensure they have uninterrupted, ideally programmatic access to data in transparent formats they can understand.

There are a number of approaches to ensuring the freedoms of both users and vendors of computing products and services, each having their own advantages and disadvantages.

A. Open Source

In a world of products rather than services, the standard approach to ensuring openness has been to use software licensing, effectively using copyright (the purpose of which is to restrict freedoms for profit) against itself (ensuring such freedoms are preserved rather than restricted). By default a software developer's works are "proprietary", in that they have rights that they can enforce through the courts to prevent users from copying their software. Rights granted by a copyright holder can include not only the ability to use the work, but also to conditionally modify and redistribute it.

Open source software licenses allow for both modification and redistribution of modified source code, and may require the latter. In either case, for copyrights to take effect the software must be conveyed between parties. This has typically been a requirement (in that one cannot execute software without being in possession of it, in the same way that one can't read a book they don't have), but where software is delivered as services rather than products this is no longer required.

1) Permissive licensing: Permissive licenses such as the MIT license [1] and 3-clause BSD licenses [2] (that allow modification but do not require distribution of modified source code) are extremely popular for web-based software. Indeed Apache, "the world's most popular web (HTTP) server since 1996" is distributed under the Apache license [3], which is generally considered permissive.

It follows that the community of developers releasing web-based software are not as concerned about derivative works as others, which further limits the suitability of software licensing to enforce freedoms in a world of services.

978-1-4799-3088-3/14/$31.00 ©2014 IEEE

2) Copyleft licensing: Copyleft licenses such as the GNU General Public License (GPL) [4], that allow modification but require distribution of modified source code with derivatives, have been wildly successful in many scenarios (Linux), and also allow for dual-licensing business models (MySQL).

Indirect use of even modified copyleft software internally without distribution (for example, custom Linux distributions deployed internally by large cloud providers) is generally considered acceptable, while allowing others to avail of modified server software directly is not, giving rise to the term "service provider loophole". Examples of this include Ruby (licensed under the weak copyleft Ruby license) versus Ruby on Rails (licensed under the permissive MIT license), or Linux running MySQL (both licensed under the GPL).

3) Service provider loophole: While proponents of permissive licenses generally accept that software can be delivered as services, authors of copyleft software typically see this as a limitation of the license. Referred to as the "service provider loophole", it essentially means others can use and modify the software, but as they deliver it as services they never convey it to users as a product. As such, they can benefit from the authors' work without having to contribute their enhancements to the community, as is intended by copyleft licenses.

Attempts to close the "service provider loophole" through license wording have had limited success. For example, the Affero General Public License (AGPL) demands that source code be advertised and made available to users of the software accessing it via web services. Due to the resulting complexity and uncertainty, few users are willing to accept the risk, and some (notably, Google) have reportedly banned its use. Furthermore, the license triggers can often be trivially bypassed using intermediaries [5].

B. Open Cloud

There has been ongoing discussion of the meaning of "open" in the context of cloud computing, and the term "open cloud" has been used widely despite not having been well defined. Some refer to the modular nature of their software stacks while others advertise the availability of APIs, and many more simply conflate the term with "open source" (despite its lack of effectiveness for services, even if an elegant solution for products).

1) Open Cloud Initiative: The Open Cloud Initiative (OCI), a California non-profit, was founded in 2009 and modeled after the Open Source Initiative (OSI). Led by a board of industry representatives, it conducted a community consultation process to define a set of Open Cloud Principles (OCP) by consensus, in the same way that the OSI produced the Open Source Definition (OSD). Products and services that are found to comply with the definition are licensed to use OCI branding, in the same way that software licensed under an OSI approved license is allowed to use OSI logos. The OCI is complementary to the OSI, providing similar functions for cloud computing products and services, rather than solely products. The core of the initial principles is as follows:

Open Cloud must meet the following requirements:

- Open Formats: All user data and metadata must be represented in Open Standard formats.
- Open Interfaces: All functionality must be exposed by way of Open Standard interfaces.

In other words, users must be able to access their data programmatically in formats they understand (e.g. transparent, unencrypted) in order for their freedoms to be preserved. Similarly vendors are not required to release the source code of their platforms and are free to compete on performance, functionality, operational excellence, etc. [6]

C. Management interfaces

As an alternative to open source licensing of the underlying software stack, the other approach commonly proposed for ensuring portability between services is to implement a common and complete management interface. The theory is that if you can manage multiple cloud services from a single tool then you should also be able to migrate workloads between them. This is not necessarily the case, unless the management interface happens to have portability primitives.

In practice little attention is given to the actual programmatic accessibility of workloads in transparent formats. In other words, while this approach may improve interoperability and manageability, it does not necessarily solve for portability. Furthermore, standardising the management interface is unnecessary; the vendors have adequate, implementation-specific solutions for this problem already that are generally incompatible with each other (similarly the management interface for Microsoft Exchange is completely different from Lotus Notes, yet they can interoperate over SMTP).

The Open Grid Forum (OGF)'s Open Cloud Computing Interface (OCCI) is one such example of this approach, aspiring to be a management interface that any cloud infrastructure provider can adopt [7]. The interface allows users to manipulate virtual machines (e.g. start, stop, restart), but this is unnecessary in order to be able to migrate them from one cloud to another.

Open Database Connectivity (ODBC) is a similar solution for relational databases, though it has been largely supplanted with native/proprietary interfaces to expose additional functionality, even if performance was comparable to native drivers [8]. A user does not need to be able to manage the database in order to avail of the data within it, and conversely a database administrator need not necessarily have access to the data in order to administer it. These are two separate use cases for which there are completely different user and machine interfaces.

1) Connectors: As it is not always practical (or indeed possible) to modify cloud stack(s) to support a common API, an alternative is to implement a "connector" that supports consumption of multiple cloud provider APIs while exposing a single interface to the developer. In addition to the performance and functionality limitations described below, connectors require the introduction of third-party code, possibly in another programming language (though the impact of this is limited for network rather than library interfaces). Examples of this include jclouds and DeltaCloud, which have both enjoyed some success, and which provide a sound transitional solution in the absence of a standard for portability, and in the longer term for management applications.

2) Impedence Mismatch: The primary drawback with the common management interface approach is that there is almost always an "impedance mismatch" between the implementation and the API itself. This is similar to object-relational interfaces, which exhibit a number of different types of "impedance mismatch", for which there are only a limited number of complex workarounds [9]. Typically proprietary interfaces such as the AWS APIs are designed specifically for the underlying platform and adapted to it as it evolves (thus they tend to be easier to use, and perform better).

The two main ways this "impedance mismatch" manifests itself are in the hiding of functionality in the underlying platform due to "limitations" in the API, and in the failure and/or poor performance of the API due to "limitations" in the underlying platform. For example, if a provider wishes to expose some new functionality to transform workloads, but the custodian of the API specification is unwilling or unable to incorporate it and does not allow for safe extensibility (for example, via shared registries rather than proprietary markup), then the functionality is either hidden or has to be exposed via a separate API. Similarly, if the API exposes some functionality that is not supported by the underlying platform (for example, some platforms don't support "stopping" workloads) then the API must either translate the call (in this case "stop" typically becomes "destroy", which has the unintended consequence of data loss), or return an error.

In less extreme cases it may still be possible to perform the function, albeit with degradation in performance or efficiency. For example, it may be possible to return a detailed list of running workloads on some platforms in a single call, while others may need to return a "thin" list and retrieve the details for each entry in turn, requiring O(n+1) underlying API calls.

3) Proprietary management interfaces: Many proponents of this approach advocate that competing cloud stacks, such as OpenStack, adopt proprietary interfaces, such as the Amazon Web Services APIs [10]. There are however a number of drawbacks; in the specific case of Amazon:

- They are the sole custodian of the API and define the direction according to their rapidly changing business needs.
- They claim copyright over the API documentation and it "may not be reproduced, duplicated, copied, sold, resold, visited, or otherwise exploited for any commercial purpose".
- Their virtual machine images (AMIs) are encrypted such that only their owner and Amazon can read them. While import/export functionality is provided for users' own images, this is limited and not currently supported for shared or public images.
- Their server-side implementation of the API is accessible as a service only, not as binary or source code.
- Their client-side implementation of the API is downloadable in binary form only and the Amazon Software License specifies that it "may be used or intended for use with the web services, computing platforms or applications provided by Amazon".

Amazon is well within its rights to protect its intellectual property in this fashion, though according to industry analyst Gartner, they have thus far "adopted a laissez-faire attitude" [11]. Indeed Eucalyptus has successfully adopted this approach (albeit for a subset of the AWS APIs), reportedly executing an agreement with Amazon to license the APIs for their own use.

D. Internet E-mail

Original electronic mail ("email") services were similar to today's cloud computing services, in that they were generally confined to a single system. In the simplest sense, each user had a file-based repository of email messages to which other users could append. In the 1970s the local SNGMSG utility was updated to transfer messages between hosts and the "user@host" addressing format adopted. This became the "killer app" of ARPANET, and then the Internet. Most other mail networks had their own servers, protocols and addressing (PROFS, SNADS, X.400, MHS, MCI, MSMail, etc.) and used "gateways" to reach "foreign mail systems", which are similar to (and have many of the same drawbacks as) the management interface connectors described above [12].

With the Standardizing [of] Network Mail Headers in RFC561 [13] in 1973 and the introduction of the Simple Mail Transfer Protocol (SMTP) in RFC821 [14] in 1982 (now RFC 2821 [15]), we had both the open format and interface we needed for a common Internet email framework (per the Open Cloud Principles above). Existing mail systems quickly adopted the new standard, lost relevance, or both, and SMTP is now a first class citizen in most modern mail systems, rather than a gateway or connector.

It is important to note that the standards do not even attempt to describe how the mail should be stored (separate files, flat files, binary databases, etc.), how the system should be administered, how attachments should be formatted, etc., just how mail should be transferred between systems. As such SMTP can be considered the "lowest common denominator" between otherwise independent email systems, and provided this interface is present they can evolve independently of it. For example, the management tools for Sendmail, Microsoft Exchange and Gmail are completely different and yet they all seamlessly transfer messages over SMTP.

III. Proposed Solution

In the same way that SMTP provided the minimum functionality required for portability of mail messages between entirely different email systems, it is proposed that a similar approach can and should be adopted for portability between cloud computing services. Doing so would allow cloud software stacks to evolve independently, locking down only the minimal functionality required for seamless portability.

By avoiding the subject of management (e.g. starting and stopping virtual machines) and decoupling the interface from the formats transferred over it (i.e. by supporting multiple data types defined elsewhere and designated by Internet media types), we can greatly simplify the specification. This significantly reduces the costs of implementation and maintenance of the interface. Complexity is also reduced by relying on mature underlying layers for services such as encryption, thereby also improving security.

This is similar to the way in which SMTP defines the transport but allows attachments of arbitrary types to travel over it. The format of the attachments themselves are specified by Internet media (MIME) types, and the protocol simply treats the payload as opaque. HTTP works in the same way, which is in part why it was selected.

Such an interface is suitable for all types of cloud services (e.g. databases, runtimes, queues, archives, applications, user data, etc.) rather than solely cloud infrastructure (i.e. virtual machines), as is the case for most existing proposals. Furthermore, separating the format from the protocol rather than binding the two together with interdependencies (like DMTF's CIMI interface and OVF format) provides for significantly more flexibility and room for future evolution.

A. HyperText Transfer Protocol (HTTP)

It is proposed that the protocol be built upon the Hypertext Transfer Protocol (HTTP) [16] rather than a new protocol (like SMTP), as it is widely deployed, very mature, broadly implemented and provides services including addressing, authentication, caching, compression, encryption, gateways, integrity verification, load balancing and proxies. It is further proposed that HTTP be used as intended by its authors, leaving the message body free for the payload and leveraging the message headers for metadata, including attributes, categories and links.

The far more complex alternative is to define a new envelope format, like the Simple Object Access Protocol (SOAP) HTTP bindings, and introduce as a dependency an existing document format such as the eXtensible Markup Language (XML) or JavaScript Object Notation (JSON). A side effect of this design decision is that modern programming and scripting languages include support for HTTP (and therefore the protocol's essential functionality) "out of the box"; retrieving a workload simply requires a "HTTP GET".

In contrast, existing protocols tend to describe new formats which are passed in HTTP messages, in addition to the workload formats themselves. This adds significant complexity and therefore implementation cost (thereby raising the barriers to entry and impacting the probability of widespread adoption).

B. Basic interactions

The proposed process for manipulation of workload/s is for the client to establish a secure HTTP connection to the server (preferably over SSL/TLS) and, for basic "CRUD" functionality:

1) **Create**: Upload the desired resource/s, specifying Universal Resource Locator/s (URL/s) in HTTP PUT request/s.
2) **Retrieve**: Retrieve a list of available resources, one URL per line, from a user-specified or default URL (optional), or retrieve the desired resource/s, specifying Universal Resource Locator/s (URL/s) in HTTP GET request/s.
3) **Update**: As for create.
4) **Delete**: Delete the resource/s, specifying Universal Resource Locator/s (URL/s) in HTTP DELETE request/s.

These basic commands provide for most scenarios, including but not limited to:

- Deploying workloads to a new provider
- Removing workloads from an existing provider
- Migrating workloads between providers
- Backing up data from an existing provider

C. Advanced interactions

More advanced functionality is envisaged for more complex use cases, such as mobile clients remotely manipulating workloads over low bandwidth links:

- **Copy**: Instruct a server to duplicate the source URL to the target URL using the HTTP/WebDAV "COPY" verb.
- **Move**: As for "Copy", only using the HTTP/WebDAV "MOVE" verb and deleting the source URL.

D. Metadata

Standard HTTP message headers (e.g. Accept, Authorization, Content-Length, Content-MD5, Range, etc.) can be used for out-of-band transfer of metadata about the workload.

Custom headers can also be used to assign descriptions and categories to the workload as well as to link to related resources (e.g. snapshots, icons):

- **Attribute**: Defines an associated key/value pair (e.g. copyright is GPL, memory is 16GB).
- **Category**: Defines membership in specified taxonomy/s, based on the Atom publishing protocol (e.g. type is virtual machine, operating system is Linux).
- **Link**: Defines an associated resource (e.g. disk image is root.img, screenshot is console.png).

E. Authentication

It is suggested that HTTP Basic authentication be used (over SSL/TLS for security), with the username and password fields set to the corresponding credentials depending on the cloud provider. For example, Amazon and GoGrid use ID and secret keys while VMware uses traditional usernames and passwords.

For advanced authentication mechanisms (for example, x.509 certificates) it may be necessary to encode the credential or pass a URL pointer to it.

F. Responses

Standard HTTP response codes are used to indicate success or failure (e.g. 200 OK, 302 Found, 404 Not Found). Additional information may be encoded in the HTTP body.

G. Formats

The format of the resource depends on the type of workload or application. In the same way that Internet Email treats attachments as opaque objects identified by MIME types defined elsewhere (and can therefore carry documents, images, videos and other arbitrary/proprietary data formats), SWAP considers workloads to be opaque and transfers them irrespective of their type.

Formats must be defined elsewhere and specified by Internet media (MIME) type. In addition to virtual machines such as VMware disk images (application/vmdk) and Microsoft VHDs (application/vhd), obvious candidates for transfer over SWAP include Java web archives (application/java-archive), MySQL database dumps (application/sql), web application root archives (application/zip) and filesystem backps (application/x-tgz). There are no doubt many others yet to be defined, such as saved queue states, load balancer configurations, etc.

H. Translation

In the simplest case, both source and target understand a common format. To maximise compatibility, multiple/many formats should be supported. Where there is not a common format, it is suggested that a translation gateway be used, communicating over the same protocol.

IV. Implications

There are various implications of the availability and adoption of such a standard for providers and consumers of cloud computing products and services. Providers can achieve and advertise portability as a key feature with minimal effort, and consumers enjoy the freedom to move between providers depending on their changing requirements (price, performance, features, etc.). Current approaches to the problem of portability are difficult (and therefore expensive) to implement, suffering from limited adoption and therefore having limited value.

A. Providers

A simple protocol is more likely to be adopted by cloud product and service providers, particularly if a reference implementation in one or more popular programming languages is made available. Ideally this would be integrated with a number of cloud software stacks (e.g. OpenStack), however in order to encourage rapid adoption (and in cases where a provider does not or can not adopt the standard) it is also suggested that gateways be developed and deployed.

Such gateways would be clients of the native cloud interface, consuming it and exposing the simple interface to the outside world. In many cases they could be implemented as workloads (e.g. virtual machines) and for the highest levels of performance and security, deployed into the target cloud itself (a kind of "trojan horse"). For example, such a gateway could host the Amazon EC2 import/export client, passing through credentials from the user and returning the requested workload/s directly from EC2. It could either be run as a public service (billed by usage or funded by third-party/s or the cloud provider itself) or published as a public workload others can instantiate for their private use.

With a combination of native support being officially added to cloud products and services, delivered as third-party extensions and/or implemented as gateways (with or without the blessing of the host), gaining critical mass should not be difficult; certainly far easier than the alternatives. Requiring cloud providers to release their source code (and thereby surrender any trade secrets contained therein), or adopt existing open source projects, has proven largely ineffective. Even where the underlying stack itself is open source, the operational infrastructure and management interface/s tend to be proprietary, thus limiting portability. Furthermore, despite the availability of various common management interface specifications, none have emerged as a standard.

B. Consumers

The consumers of cloud computing products and services potentially have the most to gain from the availability of a standard dedicated to portability; they have the freedom to move between providers to satisfy dynamic requirements, and the security of knowing that they can migrate in the case of problems with the provider. Typically failing providers only give limited time (if any) to migrate away from their servers.

While it is anticipated that most services will, for the foreseeable future, continue to differentiate on performance, features, location, etc., it is also anticipated that certain workloads will become commodities, differentiated only on price. This not only increases competition, thereby reducing costs, but also enables new business models such as wholesale cloud computing exchanges like the Deutsche Börse Cloud Exchange (DBCE) and others.

V. Implementation

An experimental implementation has been developed in the Go language, including both client and server. It was deployed on a number of commercial cloud providers (Amazon Web Services in North Virginia and Ireland, and Digital Ocean in Amsterdam), as well as VMware clients and servers in a lab. For rapid prototyping, JSON was used to describe the workloads, which has pros (reduced operations) and cons (complexity).

A decision will have to be made about whether to use native HTTP, JSON, or both. An alternative would be to use a simple URL list format for enumeration (one URL per line), with HTTP headers for interrogation of each workload, but this has the limitation of requiring O(n+1) queries, and rules out the option of simple implementations using regularly updated static files. It is likely that a hybrid approach will be adopted, with a single JSON manifest including the headers that would also be available with each resource via HTTP.

The implementation needs to be integrated with its host, inspecting the filesystem and/or interrogating the API and/or datastore in order to enumerate and describe each workload. There is also often a requirement to translate from one format to another and/or between different types of the same format (e.g. VMware's thin vs thick provisioning). Furthermore, import and export operations (where supported) may be implemented as batch operations, which need to be triggered and monitored for successful completion through the API. In other words, for cloud services like AWS it will likely be necessary

to request an image and return for it later, or submit an image and wait for it to be imported.

That said, a minimum viable product has proven to be very simple and extremely good performance can be guaranteed by implementing the interface using static files. There are many challenges to be overcome (for example, pass-through authentication when instructing one cloud to migrate a workload to another from a slow connection), but the core will be easily implemented for most providers.

With the current implementation a server simply needs to execute 'swap server', which creates a HTTP listener. Clients can then use 'swap parse <url>' to retrieve and parse the manifest, which results in a list workloads which can be transferred. They can then use the binary to do the transfer, or use a separate tool such as a http client (curl, wget) or proprietary protocol (S3). It is anticipated that this source code will be released as an open source reference implementation along with the protocol specification, and additional functionality will be developed over time.

VI. Conclusion

In the 5-10 years since the introduction of cloud computing, the few solutions proposed to address the problem of portability between service providers have achieved limited adoption. Similarly, attempts to adapt software licenses from product to service delivery model have been largely rejected by developers and users.

At the time of writing, cloud computing products and services were comparable to the electronic mail silos of the seventies. It was only with the introduction of the Simple Mail Transfer Protocol (SMTP) in the eighties that Internet email achieved widespread deployment in the nineties, and is ubiquitous today. It is anticipated that by adapting the "least common denominator" approach used for Internet email to cloud computing products and services, a specification has the best chance of becoming a standard via widespread deployment. This is due to the low cost of implementation, and the ability of third-parties to implement gateway for uncooperative cloud providers, which would typically run as workloads within the cloud itself (a kind of trojan horse).

A high level design for such a protocol based on HTTP has been proposed, using existing mature standards for authentication, compression, encryption and other supporting services. For flexibility and simplicity, the workload format specification has been separated from the interface, with existing formats being used and specified by Internet media (MIME) type. An experimental implementation has also been written in the Go language, delivering both client- and server-side components. This has been deployed onto a number of commercial public and private clouds. Lessons learned from the implementation process have been incorporated and others will be addressed in future revisions.

An approach for encouraging adoption of the specification using reference implementations (product integration to reach multiple cloud providers, and gateways to simplify the integration and enable third-parties to do it) has also been proposed. It remains to write the specification itself, ideally having reached community consensus on the design decisions proposed herein.

Acknowledgment

The design decisions proposed in this document have been refined (mostly reduced) following extensive discussions with many members of the cloud computing community. The initial idea to use HTTP for cloud management was floated in the initial OGF Open Cloud Computing Interface (OCCI) drafts some years ago and became the basis for that specification; co-chairs Thijs Metsch, Andrew Edmonds and Alexis Richardson, as well as working group members also actively contributed. The IETF hosted a clouds Bar BoF at IETF-78 in Maastricht in 2010, led by Bhumip Khasnabish, who provided a forum for further development of the high-level concepts and access to industry experts. Thanks also to Vint Cerf and Google for sponsoring my attendance at this event as an employee, and to Equinix for enabling me to continue my research. Mark Nottingham, HTTPbis chair, and other working group members have also given useful feedback. In 2010, OSCON hosted a Cloud Summit in Portland, Oregon, where Simon Wardley invited Benjamin Black and I to debate the value of cloud standards. In the process the true value of such standards became clear to me; the creation of tradable commodities which enable marketplaces. Fellow Open Cloud Initiative board members have helped reach community consensus on the meaning of "open cloud" and advocate its adoption. Finally, thanks to Yehia Elkhatib, Gordon Blair and Raouf Boutaba for coordinating the CrossCloud workshop at IEEE INFOCOM 2014, Gareth Tyson for shepherding the paper, and to the anonymous reviewers for providing valuable feedback. To these and anyone else who has contributed to or reviewed this (my first) published paper, I am sincerely grateful.

References

[1] Massachusetts Institute of Technology (MIT), The MIT License, http://opensource.org/licenses/MIT

[2] W. Hoskins, The BSD License, http://opensource.org/licenses/BSD-3-Clause

[3] Apache Software Foundation, Apache License Version 2.0, 2004 https://www.apache.org/licenses/LICENSE-2.0.html

[4] Free Software Foundation, "GNU General Public License (GPL)", June 2007, https://www.gnu.org/copyleft/gpl.html

[5] I. Hammouda, T. Mikkonen, V. Oksanen, and A. Jaaksi, "Open Source Legality Patterns: Architectural Design Decisions Motivated by Legal Concerns", ICSE FLOSS '09, IEEE-CS, 2009, pp. 54-57.

[6] Open Cloud Principles, http://www.opencloudinitiative.org/

[7] Open Cloud Computing Interface (OCCI) specification, http://occi-wg.org/about/specification/

[8] K. North, "Comparative performance tests of ODBC drivers and proprietary database application programming interfaces", unpublished.

[9] C. Ireland, D. Bowers, M. Newton, and K. Waugh, "A classification of object-relational impedance mismatch.", DBKDA'09, IEEE, 2009.

[10] Amazon Web Services APIs, https://aws.amazon.com/documentation

[11] L. Leong, "The Amazon-Eucalyptus partnership", Gartner, April 2012.

[12] C. Partridge, "The Technical Development of Internet Email" unpublished.

[13] A. Bhushan et al, "Standardizing Network Mail Headers", RFC 561, IETF, 2001, http://tools.ietf.org/html/rfc561

[14] J. Postel, "Simple Mail Transfer Protocol", RFC 821, IETF, 1982, https://tools.ietf.org/html/rfc821

[15] J. Klensin et al, "Simple Mail Transfer Protocol", RFC 2821, IETF, 2001, http://tools.ietf.org/html/rfc2821

[16] R.Fielding, J. Gettys, J. Mogul, H. Frystyk, L.Masinter, P.Leach, T. Berners-Lee, "Hypertext Transfer Protocol - HTTP/1.1", RFC 2616, IETF, 1999, https://tools.ietf.org/html/rfc2616