When to Use Standards-Based APIs (Part 2)

ALAN SILL

Texas Tech University,
alan.sill@standards-now.org

THIS IS THE SECOND OF TWO COLUMNS THAT EXPLORE THE ROLE OF STANDARDS IN APPLICATION PROGRAMMING INTERFACE (API) DESIGN. In the first part, I looked at the special role of APIs in cloud computing, why and where they make sense in software design, and in particular, when to use (and when not to use) standards-based APIs in your work.

Here, I'll look at standards that apply to real-world infrastructure using examples from datacenter software and hardware control, summarizing some important standards that are actively used in these areas. Along the way, I'll introduce other types of interfaces beyond APIs, and describe the contexts of these interfaces as well as the relationships among them.

Once again, I'll try to make the discussion practical by including samples of tools and software that can be used in a variety of settings, and by including familiar and easily replicated use case situations. Part of my intent in this column is to introduce the context needed to discuss clouds that span multiple providers or physical datacenter settings, since these will be the topics of upcoming columns.

Interfaces in Context: Beyond APIs

To understand the role of APIs in physical infrastructure control, it's useful to supply a bit of context to see where they fit into the overall picture. The first step is to add some details regarding different types of interfaces.

Calls to underlying layers of computing systems are sometimes made through precompiled interface components that are built into the system's control software. When these occur through library functions, operating system feature invocations or machine code, they represent lower-level operations that are properly referred to as application binary interface (ABI) calls, to be distinguished from API invocations that traditionally take place at the source code level.

The distinction between these types of interfaces is sometimes obscured in cloud systems by the fact that much of the interaction between cloud software components takes place using remote methods. In the past, such parameter-passing interfaces have been referred to as remote procedure calls (RPCs), and although that term has recently fallen out of favor in cloud settings, it remains an accurate description.

In clouds, the term "API" has come to be applied broadly to interfaces that accept exchanges of parameters between software components that are often on different machines entirely, and don't depend on the details of the underlying nature of the interface. This change in the meaning of the terminology has some negative consequences. For example, broad classes of conceptually different styles of interfaces are now referred to as APIs, even if they have little to do with programming applications and are more often simply parameter passing interfaces that are addressable through URIs.

One difficulty in terminology is that the parameter-passing portions of cloud applications are often the only parts that are made visible by service providers for use by outside entities. In Web service settings, these can be referred to as Web APIs, but no equivalent term has yet evolved to describe this aspect of API use in clouds and to differentiate it from the direct operations that are possible when using the interface layer directly at the source code level.

For a while, it was popular to refer to assemblages of multiple Web and cloud services that call each other or other systems to achieve different steps as "mashups," but that term also fell out of favor, since such patterns became the rule and not the exception. For this reason, APIs that operate as URI-based remote calls are often thought of as the only type available in cloud settings. For the moment, I'll refer to these as "cloud RPCs," understanding that this term is shorthand for the URI-based remote invocation of cloud-based services.

Most such RPCs in modern cloud software are carried out through Representational State Transfer (REST) or Simple Object Access Protocol (SOAP) methods over HTTP and HTTPS connections, usually passing parameters as text, JavaScript Object Notation (JSON), or Extensible Markup Language (XML) data. The standards basis of each particular cloud RPC invocation is thus just the degree to which it uses the standards or design architecture of each of these methods.

In principle, however, nothing limits the application of remote methods to these protocols and data types. In some cases, other choices could produce better results, depending on considerations such as latency, transport protocol, or the application's architectural design.

Examples of alternate transport protocols include the Advanced Message Queuing Protocol (AMQP) developed by the Organization for Advancement of Structured Information Systems (OASIS),[1] which was adopted in May 2014 as an international standard through the Joint Technical Committee on Information Technology of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC); and the Extensible Messaging and Presence Protocol (XMPP), which has evolved through the RFC process to become an Internet Engineering Task Force (IETF) proposed standard.[2–4]

Both AMQP and XMPP are used in message-oriented middleware (MOM) systems and have features that make them preferable to HTTP in asynchronous message-passing applications. XMPP has also led to creation of the XMPP Standards Foundation (http://xmpp.org), which describes itself as an independent, nonprofit standards development organization whose primary mission is to define open protocols for presence, instant messaging, and real-time communication and collaboration on top of the IETF standards.

API Design Ecosystem and Standards

The fullest meaning of a cloud API includes the software components that allow control of cloud functionality by specific methods that apply to each component, and can be invoked either remotely through URI-based methods or directly on the infrastructure controlled by the software. Using these components, a programmer should be able to customize the underlying system's behavior to the functional needs of the task at hand—in other words, to build a cloud application.

Ideally, it would also be possible to convey and describe the ecosystem associated with a cloud application that influences its use in common terms. Some thoughts on the difficulty involved in approaching this topic in simple terms are often gathered on the "API Evangelist" website, for example, in a recent blog article by Kin Lane.[5]

Lane describes the conceptual simplification needed to adequately describe the API lifecycle using a subway map analogy. Subway maps began more than 80 years ago to employ highly stylized depictions of transit routes that focused on how these routes were experienced by riders, instead of perfect accuracy in geographical detail and content. By this analogy, it should be possible to develop representations for interacting with cloud applications that focus on developer and user experience, rather than on perfect accuracy in depicting the underlying software and hardware layers.

I've described standards that are applicable to this task in previous columns in this series. In the next column, I plan to present some important cloud standards that have taken hold in infrastructure control at the software level, and to extend the previous discussion to cover considerations such as those described above in hybrid and private cloud settings. Future issues and the corresponding topics to be covered in this column will also focus on cloud migration, networking, and a number of specific industry environments, giving us a chance to describe the standards, protocols, and procedures that are applicable to such settings.

Examples from Hardware Control

To illustrate the important aspects of API design concepts, I've included some examples using a simplified setting based on physical device power monitoring and control. This setting is worth exploring on its own, and will set the stage for further discussion in later columns covering hybrid and private clouds, failover, migration, and the role of developing standards in those settings. Beyond its intrinsic interest, this use case is also relevant to the increasingly important topic of the Internet of Things (IoT) and to infrastructure automation.

A simple example that can serve as a starting point is that of device power control in a home or small office automation setting. This topic can also serve as a stepping stone into a discussion of hybrid cloud infrastructure control, which will be covered in the next issue.

Example 1: PC Power Control

Suppose you manage a collection of devices that host your local household or small office on-premise computing and storage services, and that this collection has grown to where it's no longer practical or comfortable to locate them all within easy physical reach of your normal working environment. You might have placed them in a well-ventilated storage space, or otherwise isolated them from your normal desktop setting to reduce noise or visual clutter. You still need to keep track of their status, monitor their power state and temperature, and control them physically.

Stage 1 of this local collection might be a home or small-business office that still depends on commodity PCs that you've gathered together into a small collection or cluster. You'd like to control them physically (power on, power off, or remotely force them to reboot), but would like to do so via an API that can be tied in programmatically with other processing services, such as a monitoring system that detects and responds to certain conditions. An example might be to deal with a computer or storage device in a nonresponsive state, or to apply a security patch or other intervention that requires rebooting one of the system components to complete. Server-class machines have dedicated hardware with APIs that provide such features, as discussed in the next stage of this example, but let's stick to the commodity-PC case for now.

Being cloud oriented, you might be attracted to use the metal-as-a-service (MaaS) features of the Ubuntu Linux distribution (https://maas.ubuntu.com) to help you with this integration. An obvious sticking point is that MaaS doesn't have built-in features to control your particular hardware, because your hardware explicitly doesn't have API-based hardware power control capabilities of any kind.

Liam Young, an OpenStack software engineer at Canonical, has written a post explaining how to build a small relay control setup based on the Raspberry Pi Model B (www.raspberrypi.org/products/model-b/) to achieve this task.[6] Young's approach uses several cloud-relevant features that illustrate the level of integration that's possible with current techniques. Beyond the use of low-cost hardware components, his solution includes a small Python module that adds a template to the MaaS power management interfaces, implemented in the context of an OpenStack instance established using a custom software bundle described in his article. The Python module operates in the context of Gunicorn (http://gunicorn.org), a Web Server Gateway Interface (WSGI) compliant HTTP server,[7] to provide the REST interface.

This is an example of using an API by direct source code integration, as opposed to just using the cloud RPC features of the interface. This approach lets you conveniently add physical control for specific hardware to the MaaS power control library, although at present you do need to edit and recompile the relevant MaaS software at the source code level.

Young mentions in his write-up that work is in progress within the community to establish better-defined software patterns that can make these modules more pluggable, thus making it easier for programmers to write MaaS power interface control and monitoring routines. If you've been reading this column regularly, this situation and this kind of evolution will sound familiar. They constitute steps of the sort I've described previously in the direction of developing a common standard.

Of course, the subject of home automation doesn't stop with this example. Rapid evolution is taking place in the development of standards and techniques for interfacing a wide variety of physical infrastructure for local and remote control. For our purposes here in illustrating API usage and features, this example is sufficient, and I'll leave a more thorough exploration of IoT APIs and techniques for a future column. To understand some of the broader context of this problem, let's look now at variations of these methods in the context of datacenters.

Example 2: Datacenter-Oriented Control Interfaces

A more sophisticated physical infrastructure control system might involve the use of existing interfaces built into server-class datacenter hardware. Such interfaces are often called "out of band" management systems, since their functions take place separately from the stream of normal ("in-band") data communications for services provided by the machines, and often occur on different networks. Once again, we can look at the standards basis for datacenter hardware control in this context.

An important class of such interfaces is the baseboard management controller (BMC). Although widely used and provided by many vendors, implementations of BMC methods have never been completely uniform among hardware manufacturers and suppliers, at least in part due to the wide variety of

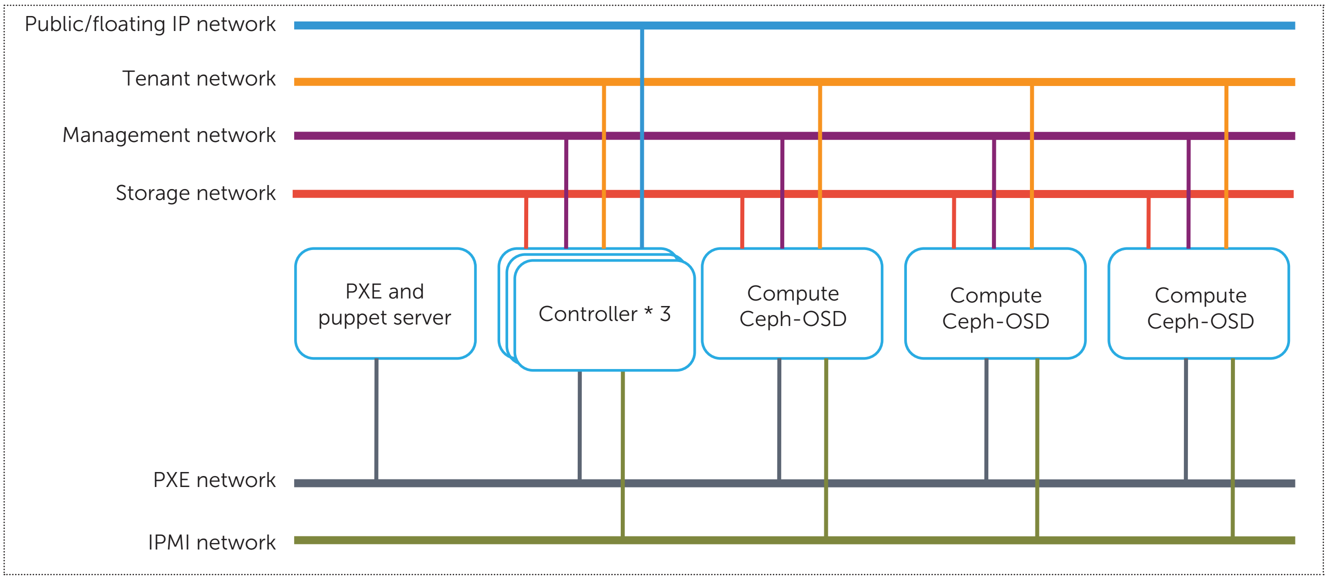

FIGURE 1. Example of current trends for use of multiple networks in cloud infrastructure for monitoring and control of physical hardware.[8]

physical components used in constructing server hardware, and their design has a long history.

Current methods used for out-of-band control in datacenters usually involve variations on the Intelligent Platform Management Interface (IPMI) toolset, which was derived in a somewhat ad hoc way beginning in the late 1990s by a loose industry consortium after earlier development of somewhat incompatible versions. IPMI is a multiprotocol interface layer that accepts both local and remote API and ABI calls for the purpose of controlling and monitoring the status of hardware components in distributed systems. Current and historical versions of the toolset are hosted at the IPMI site (www.intel.com/content/www/us/en/servers/ipmi/ipmi-technical-resources.html) along with associated supporting specifications.

IPMI isn't the only level of infrastructure control needed in distributed cloud settings, nor is it the only function that merits its own physical or virtual network. Large-scale clouds often implement multiple control and monitoring networks, devoting specific ones to provide, for example, separate functionality for external connectivity; internal tenant communication; cloud software management, storage, machine, and boot image traffic; and hardware physical monitoring and control. Figure 1 shows an example network topology for a modern OpenStack deployment.

One reason for such multiple network architectures in large-scale cloud settings is the need to differentiate between failures of the hardware or software at the individual node level and failures in connectivity among the systems. A particular node might be operating fine but shows up as unavailable because of a temporary overload or loss of communication in one or more of the networks connected to it.

Sophisticated methods are evolving to allow decisions to be made regarding whether to reinitiate or physically power cycle cloud hardware components and migrate their existing workloads, based on multiple simultaneous automated approaches to monitoring their state. Many of these methods use the standards I've mentioned, as well as others covered in previous columns, such as the Open Cloud Computing Interface (OCCI).[8,9]

Example 3: Future Standards-Based Infrastructure Control

Despite evolving considerably over the last several years from its disparate multivendor roots into a fairly well-accepted industry standard, the IPMI specification exhibits some limitations that prevent it from providing an ideal match to all of the needs of modern datacenter infrastructure deployments.

Some of these problems stem from the specification's underlying design, and some are due to evolution in the preferred methods for interacting with APIs in datacenter settings. Although it was designed initially with scaling in mind, IPMI's scaling behavior for very large numbers of machines isn't robust at the levels encountered in modern datacenters, and it suffers from some internal modeling limitations that prevent it from being used with RESTful API tools, data exchange methods, and RPC patterns currently popular in cloud infrastructure support settings.

To address these problems and extend the interface options for datacenter device control, the Distributed Management Task Force (DMTF) has recently introduced a new standard, designated Redfish, through a consortium of participants that includes many of the earlier developers of IPMI.[10] Additionally, the Open Compute Foundation has introduced a set of open hardware management specifications (www.opencompute.org/wiki/Hardware_Management/SpecsAndDesigns) that includes a proposed cloud server multinode system specification that could be the basis for further work.[11]

Although most public cloud deployments are insulated from this level of infrastructure control, that's not always the case, and such detailed control can in fact be important in a variety of private and hybrid cloud settings. I'll cover this topic and say more about the applicable standards in my next column.

THIS DISCUSSION COMPLETES THE TWO-PART INTRODUCTION TO THE CONDITIONS AND PATTERNS THAT CAUSE STANDARDS TO EVOLVE IN A VARIETY OF CLOUD SETTINGS, AND THAT LEAD TO THEIR USEFULNESS. By selecting and presenting specific examples in each of these columns that start from familiar, almost everyday activities, I've tried to trace a path from these simple starting points to much more complicated cloud settings. Along the way, I've covered the role of APIs in cloud software, and distinguished between APIs and other types of interfaces and how these interfaces, behind the scenes, are part of the overall setting of cloud software.

Please respond with your opinions on any of these topics or on those I've explored in previous columns. Please also include any news you think the community should know about the general areas of cloud standards, compliance, or related topics. I'm also happy to review ideas for article submissions or for proposed guest columns on these topics, and can be reached for this purpose at alan.sill@standards-now.org.

References

1. *Advanced Message Queuing Protocol (AMQP) Version 1.0*, OASIS, Oct. 2012; www.oasis-open.org/standards#amqpv1.0.
2. P. Saint-Andre, *Extensible Messaging and Presence Protocol (XMPP): Core*, IETF RFC 6120, Mar. 2011; www.rfc-editor.org/info/rfc6120.
3. P. Saint-Andre, *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*, IETF RFC 6121, Mar. 2011; www.rfc-editor.org/info/rfc6121.
4. P. Saint-Andre, *Extensible Messaging and Presence Protocol (XMPP): Address Format*, IETF RFC 6122, Mar. 2011; www.rfc-editor.org/info/rfc6122.
5. K. Lane, "The API Life Cycle (My Talk From @Defrag and @APIStrat)," blog, 29 Nov. 2015, http://apievangelist.com/2015/11/29/the-api-lifecycle-my-talk-from-defrag-and-apistrat.
6. L. Young, "PC Power Control with a Raspberry Pi and MAAS," blog, 23 Sept. 2015, http://insights.ubuntu.com/2015/09/23/pc-power-control-with-a-raspberry-pi-and-maas.
7. P.J. Eby, *PEP 3333—Python Web Server Gateway Interface v1.0.1*, Python, Sept. 2010; www.python.org/dev/peps/pep-3333.
8. Z.S. Zhou et al., "Distributed Health Checking for Compute Node High Availability," OpenStack Summit, Tokyo, Japan, 2015; http://schd.ws/hosted_files/openstacksummitoctober2015tokyo/d5/summit_ha.pdf.
9. A. Ciuffoletti, "Automated Deployment of a Microservice-Based Monitoring Infrastructure," *Procedia Computer Science*, vol. 68, 2015, pp. 163–172; doi:10.1016/j.procs.2015.09.232.
10. *Redfish Scalable Platforms Management API Specification*, Distributed Management Task Force document DSP0266, Aug. 2015; www.dmtf.org/sites/default/files/standards/documents/DSP0266_1.0.0.pdf.
11. OCP Hardware Management Project, *Cloud Server Multi Node System Specification V0.7.5*, Open Compute Foundation; Aug. 2015; www.opencompute.org/wiki/Hardware_Management/SpecsAndDesigns.

ALAN SILL *is interim senior director at the High Performance Computing Center at Texas Tech University, where he is also site director for the US National Science Foundation Cloud and Autonomic Computing Center and adjunct professor of physics. Sill has a PhD in particle physics from American University and is an active member of IEEE, the Distributed Management Task Force, and TeleManagement Forum, and he serves as President for the Open Grid Forum. He is a member of several cloud standards working groups, and national and international standards roadmap committees, and he remains active in particle physics and advanced computing research. For further details, visit http://cac.ttu.edu or contact him at alan.sill@standards-now.org.*