

The Design and Architecture of Microservices

TO EXPLORE THE ROLE OF DESIGN IN SOFTWARE, CONSIDER TWO OTHER FIELDS THAT ALSO DEPEND ON IT: ART AND ARCHITECTURE. Like art, much of software design can be a matter of taste. As in the art world, issues that inspire passionate debate in the field of software design resonate most loudly within its internal boundaries, and don't necessarily have much of an effect outside of those boundaries.

Design is also important in architecture, both for aesthetic and physically important reasons. As in the structure of buildings, architectural design can have serious ramifications on the reliability, robustness, and suitability for use of software. Like architects, developers are generally aware of the importance of internal structural elements in software, and study and debate the performance and business reasons for selecting one approach over another.

One would not wish to use the plans for a per-

sonal home to build a larger structure, such as a sports arena, concert hall, or high-rise building. In the same way, the choice of architectural design pattern in software must be tuned to the desired application, workload, and expected level of use.

Aesthetics and user reaction are important in all of these settings. No one would argue at this stage, in which products from all vendors tend to be beautifully designed, about the need to pay significant attention to issues of user experience and ease of use in software design. Just as we enjoy beautifully designed and functional buildings, software designs are most enjoyable when they're both useful and artfully built.

Microservice Architectures

Concepts related to microservices are discussed extensively elsewhere in this issue. They are, to some degree, old wine in new bottles. The basic approach of separating services into functions that can interact via programming interfaces has been with us for some time. Methods to implement such separation in the framework of service-oriented architectures (SOAs) are also not new.

Recent implementations of microservices in cloud settings, however, take the SOA idea to new limits that are driven by the goals of rapid, interchangeable, easily adapted, and easily scaled components. This is obviously not the only way to use clouds, but it draws well on the basic functional features of cloud computing and is a good match to the corresponding delivery framework. A continued emphasis on the use of RESTful APIs as discussed in previous "Standards Now" columns has also accelerated the pace of change and overall utility of cloud service delivery.

The resulting factorization of workloads and incrementally scalable features of microservices provide a multitude of ways by which SOA can be freed from its previously hidebound, overly formal implementation settings and be implemented in much less forbidding ways. One consequence of this evolution is the development of new architectural patterns and the corresponding emergence and use of standards.

As with art and architecture, much discretion is left to the designer in microservice delivery. You might be tempted to think that standards aren't important, or less important, in the rapidly changing

ALAN SILL

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



microservices arena, but this assumption, as I am about to show, wouldn't be correct. To a great degree, the flexibility and ease of implementation of modern approaches to microservice architecture is either compatible with or in fact greatly driven by the emergence of successful design patterns that are already in the process of becoming standards.

Microservice Delivery Using Containers

It would be equally incorrect to equate different trends in cloud design as being equivalent. The current tendency to implement different sets of software in the context of software containers, for example, is more of a coincidence than a direct consequence of microservice design. It's true that containers can be made to isolate execution environments from each other, and that they lend themselves to scalability by allowing such containers to be instantiated quickly on demand.

Other features of software containers require much more work to overcome, however, such as the need to provide well-thought-through mechanisms for network communications between them and associated complications of their use on different physical hosts or on hosts located in different datacenters. Similar problems crop up with regard to security, monitoring, and the need to minimize the operational size of containers. These issues require careful thought and attention to details that aren't directly related to the SOA aspects of microservices themselves.

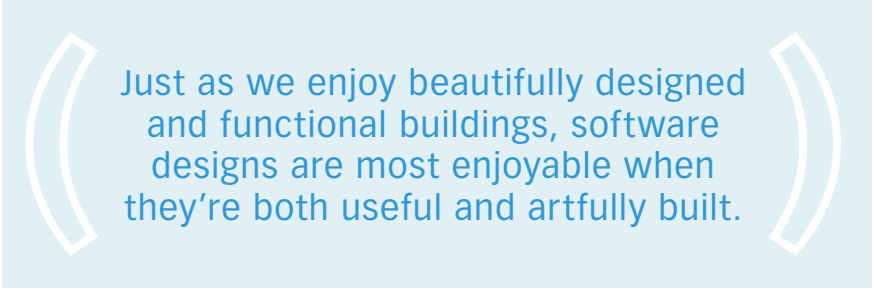
Despite these shortcomings, microservice delivery matches well in many ways to deployment in software containers. This method is, in fact, currently the most popular way to deploy them, but to deal effectively with the resulting complications just described absolutely requires the use of standards.

This column has covered the emergence of such standards in this area many times, starting with the appc application container specification originally developed by CoreOS, and the runC container engine originally developed by Docker. Much community work has gone into integrating the approaches of these two specification sets and extending them to newer, broader use cases.

Two current relevant projects of the Linux Foun-

dation are the Open Container Initiative (www.opencontainers.org) and the Cloud Native Computing Foundation (CNCF, <https://cncf.io>). Popular image formats include ACI, the container image format defined in the appc specification, and OCI, the Open Containers Image Format specification. Much of the work going on within the CNCF is aimed higher up in the software stack and addresses the large-scale behaviors of distributed systems of microservices.

Although work is still in progress on various aspects of each of these standards within these organizations and on their relationship to each other, it's encouraging to see efforts of this sort emerge naturally from ongoing community interests.



Just as we enjoy beautifully designed and functional buildings, software designs are most enjoyable when they're both useful and artfully built.

Data Formats and APIs

To make microservice architectures work in practice, one must get information into and out of these services and find ways to make the information exchange and control-passing features take place at component boundaries. Programmers must therefore address design topics dealing with data exchange and messaging, and must implement these services with suitable orchestration and control.

Standards exist that provide the basis for such data exchange. The most popular data formats in cloud computing are JavaScript Object Notation (JSON) and XML. JSON is documented in two standards: Ecma International's ECMA-404¹ and IETF's RFC 7159.² XML is a somewhat older but still popular text-based format for data exchange supported by several W3C standards. Although it isn't as human-readable as JSON, each format has particular strengths and weaknesses and both are still very much in use.

For Internet of Things (IoT) and sensor-oriented settings, as discussed in the previous issue on manufacturing, the Sensor Network Object Notation (SNON, www.snon.org) is a representation based on JSON that includes some predefined fields that are especially useful in dealing with sensor data. In addition, the Data Distribution Service (DDS, <http://www.omg.org/spec/DDS>) and DDS Data Local Reconstruction Layer (DDS-DLRL) specifications were developed by the Object Management Group specifically to handle data interchange tasks related to IoT systems.

General data standards are available to deal with the wide variety of formats for datasets without having to be locked into a particular format.

Unlike the other protocols I've mentioned, DDS can handle content-aware network routing, data prioritization by transport priorities, and both unicast and multicast communications within the methods defined by the standard set itself.

Additionally, general data standards are available to deal with the wide variety of formats for datasets without having to be locked into a particular format. For example, working with the US National Center for Supercomputing Applications (NCSA, <http://www.ncsa.illinois.edu>) and IBM, the Open Grid Forum has developed a language for describing the structure of data formats without needing to rewrite them. The resulting Data Format Description Language (DFDL, www.ogf.org/dfdl) is a flexible, general specification set suited to a wide variety of data input, output, and format transcription problems and is supported by both commercial and open source software implementation tools.

Many approaches currently used in microservices create custom APIs for access to specific data. This approach is compatible with, though typically implemented without, reference to external data for-

mat standards. As a result, current cloud microservice designs are burdened with a huge variety and multiplicity of API definitions.

In previous columns, I've referred to the API directory maintained, for example, by the website ProgrammableWeb.com, which at the time of this writing maintains a directory of more than 15,000 APIs (www.programmableweb.com/apis/directory). This situation requires APIs to be designed to work either in small subsets of the application arena in which either the API is stable, or to be built to a common self-describing or standardized pattern.

Examples of effective API standards are the RESTful API Markup Language (RAML, <http://raml.org>) and Swagger, which has evolved into the Open API Initiative (<https://openapis.org>), as discussed in previous columns.

Messaging Standards

The next step after understanding data formats and APIs for data exchange is to move in the direction of messaging and application control. HTTP and its secure variant HTTPS are the most familiar messaging standards, and are specified in a range of IETF documents summarized at the working group website (<http://www.ietf.org/specs>).

The IETF specifications underlying TCP form the basis of a large fraction of Internet traffic. TCP continues to receive detailed attention from the community due to its importance in a wide variety of settings. The most important TCP specifications and their relationships with one another are summarized in RFC 7414.³ A number of other application, transport Internet, and link layer protocols are also useful.⁴

The User Datagram Protocol (UDP) is useful for Internet communications that can be intermittent or don't have to be completely received at all times.⁵ UDP can be used to carry out IP communications in situations in which handshaking and verification of receipt of the individual message packets aren't necessary. The Stream Control Transmission Protocol (SCTP) provides an alternative to TCP and UDP applicable to streaming use cases.⁶

Another example of a manufacturing-relevant specialized transfer standard is the Constrained Ap-

plication Protocol (CoAP).⁷ According to its description, “CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. CoAP is designed to easily interface with HTTP for integration with the Web while meeting specialized requirements such as multicast support, very low overhead, and simplicity for constrained environments.”

The Extensible Messaging and Presence Protocol (XMPP) is an XML-based communications standard designed for message-oriented middleware communications. The core specifications for XMPP are RFCs 6120,⁸ 6121,⁹ and 7622¹⁰ and include a WebSocket binding defined in RFC 7395.¹¹ Several extensions beyond the base specifications are supported by the dedicated XMPP organization (see <http://xmpp.org/extensions>). Beyond its applications to human-oriented communications, XMPP is also used in smart electrical grid applications and a variety of industrial settings. Several extensions oriented toward use in IoT settings were published in late 2015.

Methods to handle publish/subscribe messaging can have advantages compared to the protocols when used for machine-to-machine communications at high speeds. The Message Queuing Telemetry Transport (MQTT, <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html>), recently standardized by the Organization for Advanced Structured Information Systems (OASIS), is another example of such a method.

The Advanced Message Queuing Protocol (AMQP) is another popular middleware messaging standard set. It can be applied using either publish/subscribe or point-to-point communication patterns. OASIS published AMQP as a set of standards in 2014 (www.oasis-open.org/standards#amqp1.0) and adopted it as a joint International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) later that year.¹² AMQP has a layered architecture, and the specification set is organized into different parts to reflect that architecture.

Networking Considerations

Networking provides the core feature that ties all cloud services to each other. I discussed this topic extensively in the May/June 2016 issue of this maga-

zine,⁴ stating there that “the underpinnings of the cloud consist of the ways in which otherwise disconnected, highly scaled, and rapidly changing collections of service components can be instantiated and hooked together swiftly and flexibly to form the basis of a cloud service.”

The need for performance is especially important in the implementation of microservice architectures. This consideration obviously provides the practical limit to the degree to which individual service components can be scaled down in terms of information exchange and functionality. Issues related to security, connectivity between microservice components, and scalability also have considerations that are affected by the choice of networking architecture and protocols.

A US National Institute of Standards and Technology draft publication covers this topic, with an emphasis on security considerations.¹³ Although the comment period has closed on this particular draft, the topic’s general nature makes it seem to me that the issues discussed in this document will be revisited several times in the near future as the general area of microservice delivery matures.

Special considerations that relate to networking are also pushing some microservice frameworks in directions that lead away from human readability of the interchanged data and even of the on-the-wire protocols used in the API calls and messaging. Examples that I’ve discussed in previous issues continue to mature, including the recent release by Google of its gRPC framework at version 1.0 (<https://github.com/grpc/grpc/releases/tag/v1.0.0>) after an extended period of development, with multiple language bindings now available.

The design approach underpinning gRPC makes extensive use of protocol buffers (<https://developers.google.com/protocol-buffers/docs/reference/overview>), a design construct intended to serialize structured data in a simpler manner than in XML but without some of JSON’s limitations, and is designed to be compatible with HTTP/2. My personal belief is that these developments illustrate the emergence of new design trends for cloud services that favor speed and responsiveness over human readability of the exchanged data and API calls, and that might lead to radical revisions of some of the fundamental assumptions that govern microservices in the future.

THE DISCUSSION HERE HAS FOCUSED ON THE DESIGN AND ARCHITECTURE OF MICROSERVICES.

I've covered considerations related to packaging and delivery of microservices in containers, data exchange, and data formats, messaging and networking, focusing on some up-to-date topics on standards related to these areas.

My next column will address topics related to microservices orchestration, including relevant standards such as Topology and Orchestration Specification for Cloud Applications (Tosca) and Cloud Application Management for Platforms (CAMP); microservices control, including the Open Cloud Computing Interface (OCCI) and Cloud Infrastructure Management Interface (CIMI) standard sets; and serverless microservices, such as Amazon Lambda and related concepts. I'll also take another look at the SOA basis for microservice architectures to tie both of these columns together.

As always, this discussion only represents my own viewpoint. I'd like to hear your opinions and experience in this area. I'm sure other readers of the magazine would also appreciate additional information on this topic.

Please respond with your input on this or previous columns. Please include news you think the community should know in the general areas of cloud standards, compliance, or related topics. I'm happy to review ideas for potential submissions to the magazine or for proposed guest columns. I can be reached for this purpose at alan.sill@standards-now.org.

References

1. Ecma International, *The JSON Data Interchange Format*, Ecma-404, 1st ed. 2013; www.ecma-international.org/publications/standards/Ecma-404.htm.
2. T. Bray, ed., *The JavaScript Object Notation (JSON) Data Interchange Format*, IEEE RFC 7159, 2014; <https://www.rfc-editor.org/info/rfc7159>.
3. M. Duke, et al., *A Roadmap for Transmission Control Protocol (TCP) Specification Documents*, IETF RFC 7414, 2015; www.rfc-editor.org/info/rfc7414.
4. A. Sill, "Standards Underlying Cloud Networking," *IEEE Cloud Computing*, vol. 3, no. 3, 2016, pp. 76–80.
5. J. Postel, *User Datagram Protocol*, IETF RFC 768, 1980; www.rfc-editor.org/info/rfc768.
6. R. Stewart, ed., *Stream Control Transmission Pro-*

ocol, IETF RFC 4960, 2007; www.rfc-editor.org/info/rfc4960.

7. Z. Shelby, K. Hartke, and C. Bormann, *The Constrained Application Protocol (CoAP)*, IETF RFC 7252, 2014; www.rfc-editor.org/info/rfc7252.
8. P. Saint-Andre, *Extensible Messaging and Presence Protocol (XMPP): Core*, IETF RFC 6120, 2011; www.rfc-editor.org/info/rfc6120.
9. P. Saint-Andre, *Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence*, IETF RFC 6121, 2011; www.rfc-editor.org/info/rfc6121.
10. P. Saint-Andre, *Extensible Messaging and Presence Protocol (XMPP): Address Format*, IETF RFC 7622, 2015; www.rfc-editor.org/info/rfc7622.
11. L. Stout, ed., *An Extensible Messaging and Presence Protocol (XMPP) Subprotocol for WebSockets*, IETF RFC 7395, 2014; www.rfc-editor.org/info/rfc7395.
12. *Information technology—Advanced Message Queuing Protocol (AMQP)*, Int'l Organization for Standardization/Int'l Electrotechnical Commission, ISO/IEC 19464, v.1.0, 2014; www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=64955.
13. A. Karmel, R. Chadromouli, and M. Iorga, "NIST Definition of Microservices, Application Containers and System Virtual Machines," Nat'l Inst. of Standards and Technology (NIST) Special Publication 800-180, 2016; http://csrc.nist.gov/publications/drafts/800-180/sp800-180_draft.pdf.

ALAN SILL directs the US National Science Foundation's Cloud and Autonomic Computing industry/university cooperative research center. He's interim senior director of the High Performance Computing Center and adjunct professor of physics at Texas Tech University, and visiting professor of distributed computing at the University of Derby. Sill has a PhD in particle physics from American University. He's an active member of IEEE, the Distributed Management Task Force, and the TeleManagement Forum, and he serves as president for the Open Grid Forum. He's 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. Contact him at alan.sill@standards-now.org.