On Service-Oriented Architectures for Mobile and Internet Applications

Sathiamoorthy Manoharan
Department of Computer Science
University of Auckland
New Zealand

***Abstract*–Service-oriented architectures have been around for long now, but the surge in the Smartphone and tablet market and the wide availability of fast mobile networks now cast new light on service-oriented architectures. The diversity of mobile platforms demand application abstraction. Such abstraction can be made possible by adapting a service-oriented architecture where the bulk of the business logic is hosted as a service. A service-oriented architecture may appear to be unsuitable when mobile networks are slow or unreliable. However, most modern mobile networks are reliable and reasonably fast, and thus applications employing a service-oriented architecture do not necessarily reduce user experience when compared to native, device-hosted applications. This paper reviews some of the technological advantages and challenges arising from the use of a service-oriented architecture for mobile and Internet applications.**

***Keywords*-Service-oriented architecture (SOA), Software architecture, Application layer, Application security.**

I. Introduction

The surge in the Smartphone and tablet market and the wide availability of fast mobile networks now cast new light on service-oriented architectures.

The diversity of mobile platforms poses interesting challenges to application developers. To reach all potential users of the application, the developers need to make the application available for a number of platforms where the operating systems [1], [2], development environments and languages [3], [4], [5] may substantially differ. This may therefore require the developers to ‘replicate’ code for these different platforms. However, code replication, in whatever form, does not follow good software engineering principles. Depending on the type of the application, a possible alternative that mitigates code ‘replication’ is to employ a service-oriented architecture [6]. In this case, the bulk of the business logic is centralized in a service and is published on a centrally located server (typically in the Cloud). Thin clients are then developed for the chosen number of platforms; these clients consume what the service offers and typically share little code.

See Figure 1 that shows a broad overview of service-oriented architecture. The clients can either be native or browser-based, depending on the application requirements.

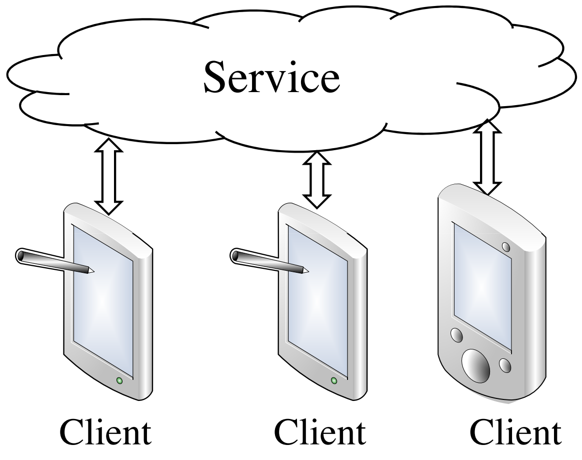

Figure 1. An overview of service-oriented architecture

Two immediately recognizable challenges with such a service-oriented approach is availability and performance. The thin client may not be available for use if there is no or limited connectivity to the service. The amount and frequency of data transfer between the client and the service may lead to performance bottlenecks. Not only that the application logic needs to take into account its algorithmic efficiency, but also it needs to take into account its data transfer efficiency.

This paper reviews some of the technological advantages and challenges arising from the use of service-oriented architectures for mobile and Internet applications. Note that this is a review paper drawing from the practice and experience of constructing services and clients. It does not propose new research results.

The rest of the paper is organized as follows. Section II reviews the fundamental aspects of services and service-oriented architectures. Section III discusses performance of various segments of service-oriented architectures. Section IV explores how service-oriented architectures can be secured using standard Internet authentication mechanisms. Section V presents a case study illustrating some of the concepts discussed in this paper. The final section concludes the review with a summary.

II. Services and Service-oriented Architectures

The World Wide Web Consortium (W3C) defines service-oriented architecture as a "set of components which can be invoked, and whose interface descriptions can be published and discovered" [7]. Sprott and Wilkes extend this to say that "services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface" [8]. The service interface defines the operations provided by the service and the message types and formats to be used with these operations.

SOAP [9] has traditionally been the message format used by service operations. The SOAP envelope has a header and body and the body wraps the object requested of or sent to the service. SOAP is based on XML, and is verbose. The verbosity can result in slower transfer speeds as well as slower parsing of messages. In addition, being a text format, it requires binary objects to be encoded in text, typically using base-64 [10] which expands the binary object by a factor of $\frac{4}{3}$.

SOAP-based Web Services consume SOAP messages and produce SOAP message. Since SOAP messages are verbose (and thus large), the messages are generally sent to the service using HTTP POST. A downside of using HTTP POST is that POST responses are not easy to cache.

SOAP-based Web Services are therefore not quite efficient. SOAP messages are heavy-weight and the responses from SOAP-based Web Services are difficult to cache.

Representational state transfer (REST) is an HTTP-based light-weight protocol that provides loose-coupling between the service and the client consuming the service [11], [12]. REST does not have a fixed message format, and thus a REST-based service (commonly known as a *RESTful service*) can deliver any message (i.e., any MIME type). For instance, binary objects such as a PNG image need not be encoded into a textual format in a RESTful service, thus there is no message bloat required by REST. Besides, objects can be delivered to the service using HTTP GET (i.e., as part of the URL). For example, the following URL describes a service operation that searches for the term "soa" and returns the top 10 results:

http://search.site.org/?term=soa&results=10

This URL passes to the service at *serach.site.org* two objects: a string named *term* with value *soa*, and an integer named *results* with value *10*.

HTTP GET has the advantage of letting its response cacheable. For example, the response from the above URL may be cached in the client (e.g., browser) or any other intermediaries (e.g., proxy) for a period of time.

Modern Web Services are based on REST [13]. The service operations of a RESTful Web Service are defined by URLs and are therefore a lot simpler than the equivalents in a SOAP-based service. The messages in a RESTful service are more efficient than SOAP messages. Besides, service responses can often be cached for re-use, thus saving on server resources, network bandwidth, and latency.

Overall, a RESTful Web Service is more efficient than a SOAP-based Web Service. Service-oriented architectures are therefore best-implemented using RESTful services [14].

III. Performance Considerations in Service-oriented Architectures

As we have already seen in the previous section, RESTful services outperform SOAP-based services. Thus, it is desirable to use RESTful services unless there is a strong reason not to (e.g., compatibility with legacy systems).

Given the inefficiencies of SOAP-based services, we will discuss performance considerations only for RESTful services.

One of the most important performance consideration in RESTful services is to reduce latencies involved in data transfer. Principles of latency reduction can be summarized by the three "R"s:

- Reuse
- Repetition avoidance
- Redundancy removal

In the context of service-oriented architecture, reuse is achieved by exploiting temporal locality. Items that have been used in the past are likely to be used again, and thus it pays to keep those items cached for later reuse. Effective caching can result in large savings in latencies, for there is little data transfer involved when the item is found locally in the cache.

The transport protocol HTTP supports caching extensively [15], [16]. All service responses should be considered carefully to permit caching at the service so as to optimize repeat-requests. Such caching reduces load on the service as well as reducing latency for the service consumer. Personalized responses (e.g., bank balances) will need to be cached on a per-user basis at the service; if this is not possible, then such responses must not be cached. Caching at the client end (i.e., service consumer end) should always be allowed.

When a response is cacheable, a suitable expiry time stating how long a cache could keep the response fresh should be indicated by the service. The service is in the best position to determine such expiry times. An item that is past its expiry date may not always be stale. In this case, it can still be used from the cache so long as it is re-validated to be fresh by the service. To help such re-validation, the service should consider setting a validator (e.g., HTTP *ETag* [16]).

Repetition avoidance and redundancy removal are typically achieved through compression. HTTP supports compressing the payload in responses. RESTful services thus can benefit from this. However, payload in a request is not typically compressed (by a client): payloads are allowed in HTTP POST but not in HTTP GET. If POST has to be used to transfer data, then it is important to consider compressing the data prior to the transfer; and this has to be done by the service consumer manually and the service should be designed to accept compressed payloads.

HTTP headers are never compressed. The service and the consumer have to be mindful of this and ensure the headers and header values are optimally used. For instance, unnecessary headers and header values should not be exchanged.

Similarly, it is beneficial to keep service URLs terse. Given that the URLs are seldom manually processed, terseness will not be an issue. Smaller URLs are quicker to send. Besides, there are limits on URL length imposed by HTTP entities (such as browsers). For example, the sample service URL we saw in the previous section

http://search.site.org/?term=soa&results=10

can be shortened to

http://search.site.org/?t=soa&r=10

Such shortening will allow room for larger data input to the service still using HTTP GET.

Output of a service operation is typically a custom object. Both XML and JSON (JavaScript Object Notation) can be used to serialize a custom object so that the object can be transferred to the service consumer. In a RESTful service, the XML response does not include a SOAP envelope and therefore the response is more compact. However, XML is still verbose. The other widely used alternative is JSON and JSON-serialized objects are generally more compact than XML-serialized objects. BSON is binary-encoded JSON and thus is even more compact than JSON.

With compression, the size difference between JSON-serialized objects and XML-serialized objects may not be apparent. However, generating and parsing large serialized forms require additional resources, so usually it is beneficial to use a compact serialized form.

When choosing the serialization format for custom objects in a service-oriented architecture, we must consider both

1) data transfer time, and
2) serialization and de-serialization times.

IV. Security Considerations in RESTful Service-oriented Architectures

In many systems, the service operations should only be made available only to those who have access rights.

One of the simplest form of limiting service consumers is based on their IP addresses. The service can be configured to either allow or deny accesses from service consumers originating at a given range of IP addresses.

The other simple form of limiting access is based on username and password. A consumer in this case will first need to authenticate herself by supplying the pre-registered username and password combination. Once authenticated, the service may decide either to authorize the consumer or not based on the access rights setup for the user.

HTTP servers support two standard authentication mechanisms: basic and digest access authentication [17].

With basic authentication, the username and password are supplied in clear in the HTTP headers. Basic authentication is thus prone to man-in-the-middle attacks and is not secure. However, used with HTTPS which encrypts all the transactions including the authentication headers, basic authentication is secure enough.

Digest authentication never transfers the password across the transport channels [17]. It only sends the digest of the password and a number of other entities (such the user name and realm) using a challenge posed by the server. Therefore, digest authentication is stronger than basic authentication, and is a candidate to consider if using HTTPS is not possible (e.g., because of setup costs or speed).

Another aspect to consider is the security of data during transmission. Sensitive data should not be available to eavesdrop. Use of HTTPS is the widely-interoperable and simplest means of protecting data from eavesdroppers.

V. Case Study

The case study involves revamping a University website to using a service-oriented architecture so that both native mobile and web applications can be created providing a richer user experience than what is currently available with the site through a standard browser.

Most of our websites are not easily viewable on smart devices with a small screen (e.g., Smartphones). As an example, accessing the Computer Science website at the University of Auckland from a Smartphone shows the difficulties of reading and navigation [18]. This case study involves re-architecting Computer Science's information services so that the information can be rendered to suit the target device.

To this end, we (1) separate information content from the user-interface and make available the information as a service, and (2) construct applications for smart devices that consume the information service and render them to fit within the interface paradigm of the device. This is an approach taken by a number of news media providers to provide a richer experience to the readers.

A number of data sources that supply key information content from the current site have been identified. These enable separation of information content from the presentation.

A. Service Operations

The descriptions of the operation contracts supported by the service are as follows.

1) Get a list of offered courses. The URL corresponding to this service operation resembles *http://www.site.org/css/courses*.

2) Get a list of staff IDs (or keys). The URL corresponding to this service operation resembles *http://www.site.org/css/people*.
3) Given the ID (or key), further details, such as email, phone number and picture, of staff can be obtained in the form of a vCard [19]. The URL corresponding to this service operation resembles *http://www.site.org/css/vcard?id=jbon077* where *jbon007* is an ID within the list of staff IDs.
4) Get a home screen image. The home screen image is one that changes from invocation to invocation. The image is randomly picked from a repository of images. The URL corresponding to this service operation resembles *http://www.site.org/css/himg*. This service operation returns an image (of type PNG, JPG or GIF, which can be inferred from HTTP Content-Type header).
5) Get an RSS feed of active seminars. The URL corresponding to this service operation resembles *http://www.site.org/rss?c=seminars*. The RSS feeds have the MIME type `application/rss+xml`.
6) Get an RSS feed of active events. The URL corresponding to this service operation resembles *http://www.site.org/rss?c=events*.
7) Get an RSS feed of current news items. The URL corresponding to this service operation resembles *http://www.site.org/rss?c=news*.

The service employs some of the performance and security considerations outlined in sections III and IV.

- All of the service operations except operation 4 (home image) return compressed (gzipped) data. Note that since PNG, GIF and JPG images are already compressed, further compression is unlikely.
- The custom objects (course list and people list) are serialized to JSON.
- All operations permit client-side caching
- Given the simplicity of the operations, there is no server side caching. However, there is provision to turn on server-side caching should this become necessary.
- IP-based access restriction is supported. However, given the public nature of the data, the restriction is not turned on.
- Both basic and digest access authentication are supported. Again, given the public nature of the data, no authentication is turned on.

B. Clients

Rich mobile apps that render data provided by the service operations are then constructed. These apps fit within the UI paradigm of a smart device with a small screen (e.g., Smartphone).

The client app includes logical spaces displaying course information, staff details (including photo, email, and phone number), current seminars, current events, and published news stories.

Where possible, the client app provides simple smarts to render a rich user experience. These smarts include the following.

1) If an email is selected, the Mail app is fired to compose an email to the selected address.
2) If a phone number is selected and if the device is capable of making an outgoing call, an outgoing call is initiated.
3) Being able to add the contact details from the vCard to the Address Book.
4) Being able to add events and seminars to the Calendar.

VI. Summary and Conclusion

The “write-once run-anywhere” application development paradigm does not always provide the best user experience: the application may be slow (for instance, because of layering or poor virtualization); and the user-interface may be poor (for instance, due to being unable to use native device capabilities).

Using a service-oriented architecture gets close to the principle of the “write-once run-anywhere” paradigm, but still permits providing the best possible user experience. This is through striking a balance between shared code on the server and specialist code on the client. The client code could be native and could exploit native device capabilities.

This paper reviewed service-oriented architectures in the context of mobile and Internet applications with a particular emphasis on performance and security. Following is the summary of the key points.

- RESTful service-oriented architecture is seen to outperform SOAP-based service-oriented architecture in terms of simplicity and efficiency.
- When choosing the serialization format for custom objects, we must consider the times taken to (1) serialize the data, (2) transfer the serialized data, and (3) deserialize the received serialized data.
- HTTP caching can increase the performance of services, and caching naturally lends itself to RESTful services. Both server side and client side caching are possible and are highly recommended to reduce (1) network latency, (2) use of bandwidth, and (3) server load.
- Setting a cache validator (e.g., HTTP *ETag*) can increase the effectiveness of caching.
- All data transfers, including response and request, will benefit from compression. While responses can be compressed out-of-the-box, request compression will need custom handling. However, a well-designed service will predominantly use HTTP GET and thus may not always require request compression.
- User access control can be achieved using HTTP basic access authentication (with HTTPS) or using HTTP

digest access authentication. Both of these are lightweight protocols. In addition, IP-based access control can also be employed.
- Data can be secured from eavesdroppers using HTTPS.

The paper presented a case study of an application conforming to service-oriented architecture, and illustrated some of these key points using the case study.

References

[1] Z. Mednieks, L. Dornin, G. B. Meike, and M. Nakamura, *Programming Android*. O'Reilly Media, 2011.

[2] M. Neuburg, *Programming iOS 7*, 4th ed. O'Reilly Media, 2013.

[3] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification*. Oracle, 2013.

[4] A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde, *The C# Programming Language*, 4th ed. Addison-Wesley Professional, 2010.

[5] S. G. Kochan, *Programming in Objective-C*, 6th ed. Pearson Education, 2013.

[6] N. M. Josuttis, *SOA in Practice: The Art of Distributed System Design*. O'Reilly, 2007.

[7] H. Haas and A. Brown, *Web Services Glossary*, February 2004, W3C recommendation.

[8] D. Sprott and L. Wilkes, "Understanding service-oriented architecture," *The Architecture Journal*, pp. 10–17, January 2004.

[9] M. Gudgin *et al.*, *SOAP Version 1.2 Part 1: Messaging Framework*, 2nd ed., April 2007, W3C recommendation.

[10] S. Josefsson, "The base16, base32, and base64 data encodings," *The Internet Eng. Task Force RFC 4648*, October 2006.

[11] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[12] J. Webber, *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly, 2010.

[13] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly, 2007.

[14] T. Erl, B. Carlyle, C. Pautasso, and R. Balasubramanian, *SOA with REST: Principles, Patterns &Constraints for Building Enterprise Solutions with REST*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2012.

[15] D. Wessels, *Web Caching*. O'Reilly & Associates, Inc., 2001.

[16] R. Fielding *et al.*, "Hypertext transfer protocol – HTTP/1.1," *The Internet Eng. Task Force RFC 2616*, June 1999.

[17] R. Franks *et al.*, "HTTP authentication: Basic and digest access authentication," *The Internet Eng. Task Force RFC 2617*, June 1999.

[18] U. of Auckland, "Department of computer science," http://www.cs.auckland.ac.nz/. Last Visited February 2014.

[19] S. Perreault, "vCard format specification," *The Internet Eng. Task Force RFC 6350*, October 2006.