Project Report
ATC-383

Shared Information Access Services in SWIM Segment 2: An Architectural Assessment

O. Newell
B. Levasseur

31 October 2012

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

Prepared for the Federal Aviation Administration,
Washington, D.C. 20591

TECHNICAL REPORT STANDARD TITLE PAGE

1. Report No. ATC-383	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Shared Information Access Services in SWIM Segment 2: An Architectural Assessment		5. Report Date 31 October 2012 6. Performing Organization Code
7. Author(s) Oliver Newell and Brett Levasseur		8. Performing Organization Report No. ATC-383
9. Performing Organization Name and Address MIT Lincoln Laboratory 244 Wood Street Lexington, MA 02420-9108		10. Work Unit No. (TRAIS) 11. Contract or Grant No. FA8721-05-C-0002
12. Sponsoring Agency Name and Address Department of Transportation Federal Aviation Administration 800 Independence Ave., S.W. Washington, DC 20591		13. Type of Report and Period Covered Project Report 14. Sponsoring Agency Code

15. Supplementary Notes

This report is based on studies performed at Lincoln Laboratory, a federally funded research and development center operated by Massachusetts Institute of Technology, under Air Force Contract FA8721-05-C-0002.

16. Abstract

The System Wide Information Management (SWIM) program is a foundational program for the Federal Aviation Administration's (FAA) Next Generation Air Transportation System (NextGen) initiative, with a goal of providing a common, scalable information management infrastructure. Though some benefits were realized in SWIM Segment 1 from the use of common software infrastructure components (i.e., the Progress FUSE software suite), the actual reuse of service interfaces was limited. The focus of SWIM Segment 2 is increasingly on shared services, with a goal of improved interoperability as well as increased software reuse.

This report focuses on shared data access services, based on lessons learned in the SWIM Segment 1 Corridor Integrated Weather System (CIWS) SWIM Implementing Program (SIP) activity, the NextGen Network-Enabled Weather (NNEW) program, and a number of other Laboratory net-centric programs. The applicability of other information sharing architectures, such as the Web and content delivery overlay networks, to SWIM is also assessed. Based on this assessment, a number of recommendations are suggested to facilitate the development of shared services that are flexible enough to respond quickly to evolving NextGen requirements, while at the same time minimizing the overall SWIM software 'footprint.

17. Key Words		18. Distribution Statement This document is available to the public through the National Technical Information Service, Springfield, VA 22161.	
19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 65	22. Price

FORM DOT F 1700.7 (8-72) Reproduction of completed page authorized

ABSTRACT

The System Wide Information Management (SWIM) program is a foundational program for the Federal Aviation Administration’s (FAA) Next Generation Air Transportation System (NextGen) initiative, with a goal of providing a common, scalable information management infrastructure. Though some benefits were realized in SWIM Segment 1 from the use of common software infrastructure components (i.e., the Progress FUSE software suite), the actual reuse of service *interfaces* was limited. The focus of SWIM Segment 2 is increasingly on *shared services*, with a goal of improved interoperability as well as increased software reuse.

This report focuses on shared data access services, based on lessons learned in the SWIM Segment 1 Corridor Integrated Weather System (CIWS) SWIM Implementing Program (SIP) activity, the NextGen Network-Enabled Weather (NNEW) program, and a number of other Laboratory net-centric programs. The applicability of other information sharing architectures, such as the Web and content delivery overlay networks, to SWIM is also assessed. Based on this assessment, a number of recommendations are suggested to facilitate the development of shared services that are flexible enough to respond quickly to evolving NextGen requirements, while at the same time minimizing the overall SWIM software ‘footprint.’

This page intentionally left blank.

ACKNOWLEDGMENTS

The authors wish to thank Kajal Claypool, Carol Kelly, and Bill Moser of MIT Lincoln Laboratory for discussions related to information access service design. Thanks also to Aaron Braeckel and Rob Weingruber of the NCAR for their work and insights related to Open Geospatial Consortium standards.

This page intentionally left blank.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

1.1 SCOPE

The System Wide Information Management (SWIM) program is a foundational program for the FAA's NextGen initiative, with a goal of providing a common, scalable information management infrastructure that is interoperable with other government agencies as well as international partners (e.g., European Organization for the Safety of Air Navigation [Eurocontrol]). As with a number of other NextGen programs, the program is being addressed in multiple segments, spanning the period from 2007 to 2025.

SWIM Segment 1 focused on the concept of a Service-Oriented Architecture (SOA), primarily in the context of the technologies typically associated with that architecture. Though some benefits were realized from the use of common software infrastructure components (i.e., the Progress FUSE software suite), the actual reuse of service *interfaces* was limited. The CIWS and Integrated Terminal Weather System (ITWS) data access services for example, though based on some common components, had different interfaces, limiting the generality of client-side software. Based on input from R&D organizations [1] and industry [2], the focus of SWIM Segment 2 is increasingly on *shared services*, with a goal of improved interoperability as well as increased software reuse.

The term 'SOA' today is used in a very broad sense, at a higher level than Simple Object Access Protocol (SOAP), Representational State Transfer (REST), and Web Services. There is no single definition, and SOA can be viewed from a number of perspectives. Three of the most common are:

- ***Architectural View*** – From this perspective, SOA consists of a modular set of services that can be composed in a variety of ways to accomplish a desired function. Key goals associated with this view include system agility, loose coupling, and software reuse.

- ***Governance View*** – From this viewpoint, SOA is the set of policies and procedures that are used to manage a set of capabilities, typically referred to as *services*. The focus on governance is in some sense a counterweight to the assumed agility of a SOA, to prevent IT 'sprawl.'

- ***Technical View*** – In this lower-level view, SOA is associated with certain technologies, such as eXtensible Markup Language (XML), SOAP, Web Services Description Language (WSDL), and the WS-* specification family.

SWIM takes all three of these viewpoints into consideration. In terms of detailed guidance and SWIM-supplied products, more emphasis to date has been placed on the Technical and Governance Views. Architectural guidance with respect to shared services has tended to remain at a fairly abstract level, focusing instead on defining a modular component hierarchy. Particularly with respect to data access services, the responsibility for architecting the services in SWIM Segment 1 was allocated to the various SWIM-implementing programs. This was by design, in order to gather lessons-learned from a variety of different implementations. In order to achieve the goals of interoperability, scalability, and software reuse, a set of common service interfaces based on these lessons-learned is needed.

A simplified view of the notional SOA component hierarchy, adapted from the SWIM Enterprise Architecture System View 4 (SV-4) artifact, is shown in Figure 1. The individual components shown in the SOA Core Services layer, particularly the messaging components, are relatively mature. The higher-level components in the support services layer, the data access services, are less well understood, resulting in a number of open questions. What does the set of information access service interfaces in this category look like? How many types of services are there? What is the most appropriate granularity for the interfaces? How do those services leverage the set of lower level components? Finally, how can these services best be deployed in a National Airspace System (NAS)-wide system-of-systems?

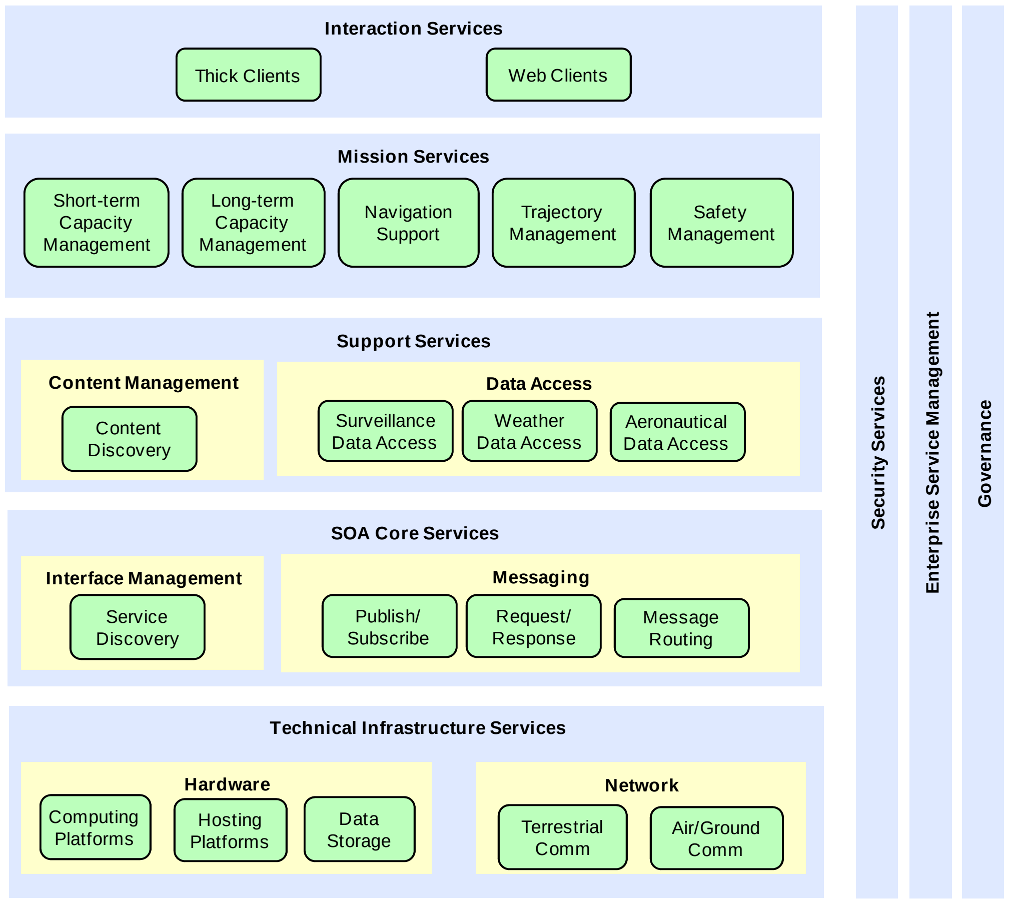

Figure 1. Simplified version of SOA component hierarchy for NextGen.

This report focuses on shared data access services, based on lessons learned in the SWIM Segment 1 CIWS SIP activity, the NextGen Network-Enabled Weather (NNEW) program, and a number of other Laboratory net-centric programs. Given that systems that provide a global information space are not new, the applicability of other information sharing architectures (e.g., the Web, Content Delivery Networks) is also discussed. The overall goal is to arrive at a coherent model for a set of shared information access services that builds upon existing best practices in both the SOA and traditional Web communities, and are well-aligned with the overall SWIM approach.

This report does not focus on infrastructure concerns that are largely orthogonal to the core information sharing functionality, such as security and monitoring. These are important areas, but are well addressed by a number of other ongoing efforts.

1.2 SOA BACKGROUND

SOA originally grew out of the Enterprise space and, as is the case with numerous computing technologies, was associated with a significant amount of vendor hype. Critics countered that SOA was a re-spinning of previous distributed object technologies (Common Object Request Broker Architecture [CORBA], Distributed Component Object Model [DCOM]) that focused on the Remote Procedure Call (RPC) paradigm, bypassing many of the lessons learned building the Internet and the Web. Roy Fielding, co-creator of the Hypertext Transfer Protocol (HTTP) and coiner of the term *Representational State Transfer* (REST) [3], had this to say in 2002 [4].

"The Web creates more business value, every day, than has been generated by every single example of an RPC-like interface in the entire history of computers...

…If this thing is going to be called Web Services, then I insist that it actually have something to do with the Web... [1]

Roy T. Fielding, Chairman, The Apache Software Foundation (fielding@apache.org)"

Over the years, the importance of some of the principles behind the Internet and the Web to information sharing at large scales has increasingly been recognized in the SOA community. A number of terms have sprung up in conjunction with this thinking, including 'RESTFul SOA,' 'Resource-Oriented Architecture' (ROA), and even 'Web-Oriented Architecture' (WOA).

[1] With respect to the highlighted last point, Dr. Fielding's main objection was that SOAP-based services simply tunneled over HTTP to pass through corporate firewalls, but they were not true participants in 'The Web' in the sense of providing hyperlinked, browsable, crawlable information. The use of the term 'Web Services' was therefore a misnomer, and likely to result in confusion.

Anne Thomas Manes, a well-known SOA evangelist from Burton Group, puts it this way:

“SOA isn’t about REST or SOAP or WS services, or ESB, or XML gateways or any of that technology. SOA is about architecture and doing the analysis of your systems to determine what are the right services to build. And then building those services and making sure that the services you deliver actually deliver business value. What I find is that most organizations are simply doing service oriented integration, which means that they are doing integration in the same way they’ve always done it, at least from an architectural perspective, just using this immensely slow and inefficient middleware to do it.” [5]

A November, 2008 Gartner post summarizing the WOA paper, *Web-Oriented Architecture: Putting the Web Back in Web Services*, makes similar points regarding the canonical SOA approach:

- *“More often than not, the WS-* protocol toolkit is unconsciously misused to create needlessly specialized interfaces.”*
- *“Interfaces based on WS-* specifications should be constrained by WOA, especially the generic interface constraints.”*
- *“Application neutrality should be the principal goal of an interface, and implementation neutrality should be a secondary goal.”* [6]

Expanding on the application neutrality point, the article continues:

“The key to shared use (reuse) is a generic, application-neutral protocol, such as the Atom Publishing Protocol (APP) or Google’s GData Protocol. Conversely, the more application-specific a protocol is, the less shareable it is. With sufficient generality, the most powerful kind of reuse becomes possible: serendipitous reuse. So important is this kind of reuse that Tim Berners-Lee and Roy T. Fielding have highlighted it an essential characteristic of the Web.” [6]

These are important points, and very relevant to SWIM as it moves forward. A key point to emphasize is that for many, the conversation has moved on from debates about *SOA vs. REST* to discussions about *RESTful SOA*, implying a combination of the strengths of the Web with the strengths of a modular, component-based approach. Others have referred to it a ‘Pragmatic SOA’ [7].

This morphing of the SOA concept over time is illustrated in Figure 2. The focus early on was on breaking down stovepipes between systems. Architectural stovepipes created as a side effect of many SOA implementations were often overlooked. More recently, the need to align new technologies with tried-and-true approaches has been recognized, and SOA has become an umbrella term that embraces multiple architectural styles, each of which has its place in a large net-centric environment. This increased alignment promises benefits in the form of fewer total services as well as shared code between implementations of the different architectural styles.

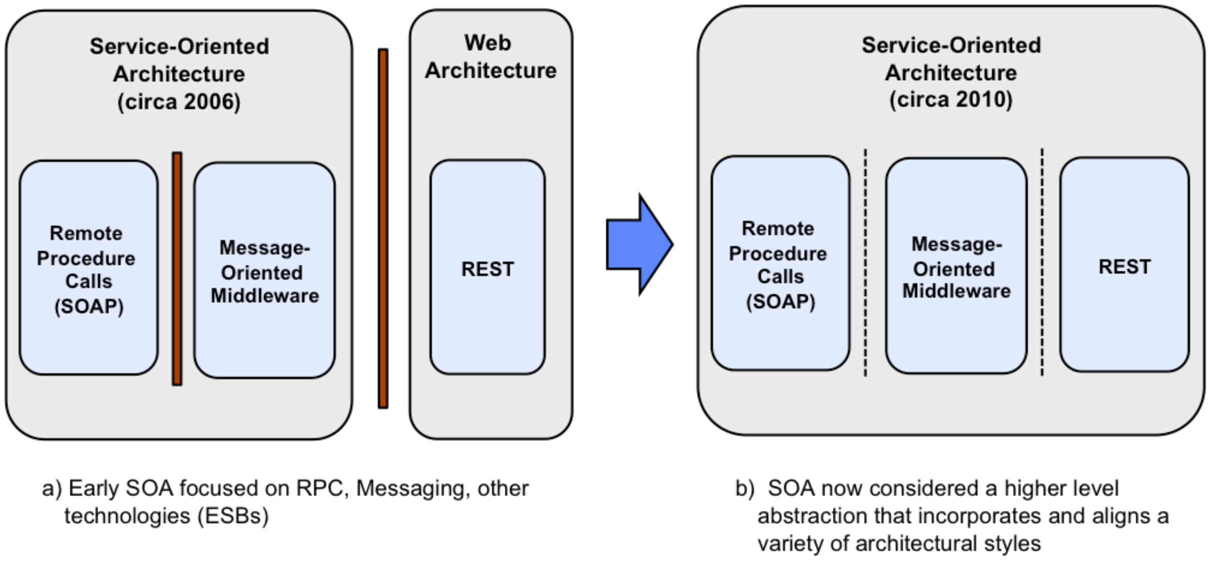
Service-Oriented Architecture (circa 2006)
Remote Procedure Calls (SOAP)
Message-Oriented Middleware
Web Architecture
REST
Service-Oriented Architecture (circa 2010)
Remote Procedure Calls (SOAP)
Message-Oriented Middleware
REST
a) Early SOA focused on RPC, Messaging, other technologies (ESBs)
b) SOA now considered a higher level abstraction that incorporates and aligns a variety of architectural styles

Figure 2. Evolution of SOA concept over time.

This page intentionally left blank.

2. SURVEY OF RELEVANT INFORMATION SHARING ARCHITECTURES

The need to share information at Internet scales is not new, and numerous approaches exist. The high-level Web architecture is perhaps the most obvious example. Content Delivery Networks (CDNs), typically overlaid on the Internet, are another. Google Earth, Google Maps, the Amazon S3 storage service, and Open Search are additional examples. This section assesses some of the architectural concepts associated with these approaches, and attempts to identify concepts that are potentially useful to SWIM.

2.1 WEB ARCHITECTURE

2.1.1 Representational State Transfer

Representational State Transfer, or REST, is the term most often associated with the high level architecture of the Web. The term was coined in Roy Fielding's classic thesis, *Architectural Styles and the Design of Network-Based Software Architectures* [3]. The term refers in part to the fact that the Web can be thought of as a giant state machine, where a click on a link is equivalent to a transition to a new 'state,' consisting of a Web page representation (typically HTML) and a set of links that define the allowable state transitions from that page to other pages. When browsing, the entire state at each node is transferred to a browser via the downloaded HTML 'representation' (page), hence the term 'Representational State Transfer.'

There is obviously more to the architecture of the Web than URL-based state transitions, and the term REST, similar to the term SOA, is most often used as a higher-level umbrella term that refers to a whole set of architectural principles. Four of the key principles are:

- *Globally Unique Identifiers for information items*
- *Linked information*
- *Stateless communication*
- *Uniform interface*

The benefits of Globally Unique Identifiers (GUIDs), in the form of URLs, are familiar to all users of the Web. A simple string-based URL, passed via email, is sufficient to allow data content of some kind to be located and downloaded from a global information space. The fact that URLs are globally unique allows for straightforward implementation of information caches throughout the network, a fact that is commonly exploited on both the server side and the client (browser) side, as well as elsewhere in the network (caching HTTP proxy). The potential benefits of caching to SWIM are the same as for the Web – improved performance and lower overall bandwidth usage. Though URLs are the dominant form of GUIDs for the Web, GUIDs in any appropriate form help enable this type of caching behavior.

GUIDs also help enable the use of linked information. In the case of the Web, users are familiar with using those links to progressively obtain more information on a particular subject. The inclusion of links, rather than actual copies of information, facilitates traversal of large information spaces without downloading large amounts of unwanted information. Though information access patterns in SWIM are not as ad-hoc as in the case of the general Web, the use of links is still very relevant. For example, weather occurring over a particular runway can refer to a runway using a GUID-based link, rather than including more detailed information, such as the runway geometry. Decision support tools can choose to follow that link to obtain the detailed geometry if needed, or ignore the link if more detailed information is not needed. This helps minimize the total amount of information that needs to be communicated via SWIM.

The focus on stateless protocols using the request/response message exchange pattern is key to the scalability of the Web. Given that the majority of information accesses are read-only, the use of stateless protocols allows for requests, even successive requests from a single user, to be serviced in a completely independent manner. This makes for very straightforward implementation of load balancing, either at a single location or in a more distributed topology. Loads on individual servers are also reduced due to the lack of need to maintain session and connection state information.

In real-time information access scenarios, the use of stateless protocols that make use of the request/response message exchange pattern can be problematic, due to the requirement to poll for updates. The use of polling introduces a certain overhead on the server side, as well as an inherent access latency based on the selected polling interval. The bottom line for the Web is that there is no single 'best' solution – whether or not a stateless approach is appropriate depends on the needs of a particular application. With respect to SWIM, the lesson is the same – support for both stateful publish/subscribe messaging and stateless request/response style messaging will be needed, depending on the application. Any discussion of a SWIM Enterprise Service Bus (ESB) should assume that support for publish/subscribe-style messaging is only part of the messaging picture.

The benefits of the Web's uniform interface are intuitively obvious to users – given a single browser application one can browse and download all types of content using the HTTP protocol, even content whose format is unknown from the browser perspective. The general utility and adaptability of the HTTP protocol is evidenced by the fact that the current version of the core HTTP specification remains at 1.1, the version standardized in 1999. The lessons of HTTP, especially in light of SOA approaches which are, in a sense, competing, are important enough to warrant a more detailed discussion. This is presented in the next section.

2.1.2 The Uniform Interface

The conventional SOA design approach is to decompose a problem (often related to data access) into a set of functions, and create a service that supports those functions via a set of operations. A common tendency is to map low-level functions more or less directly to service operations, resulting in relatively fine-grained service interfaces. An example of this tendency is provided in the first interface example in Thomas Erl's book, *SOA – Principles of Service Design* [8], depicted in Figure 3.

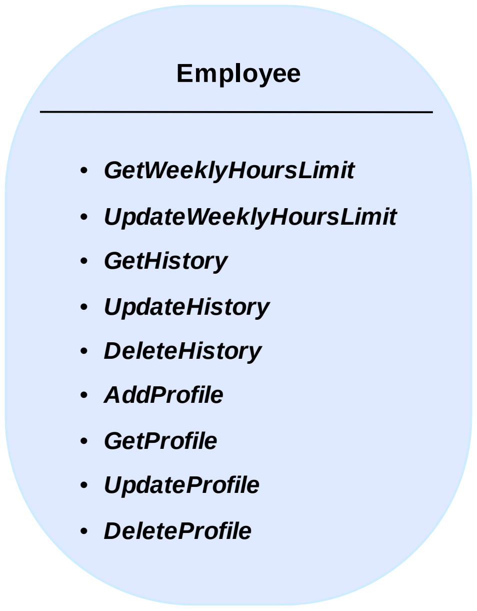

Figure 3. An example of a service interface (adapted from 'SOA – Principles of Service Design')

In Erl's text, it is noted that *"Several of its capabilities are reminiscent of traditional CRUD (create, read, update, delete)."* This is only mentioned in passing however – the relevance of the CRUD principle to shared service interfaces and scalability is not discussed.

A language analogy is useful here. Languages distinguish between verbs and nouns, and, generally speaking, a verb can be applied to many nouns. Following a food theme, some possible sentence fragments are:

- get *salad*
- *get chips*
- *get hamburger*
- *get milk*
- *consume salad*
- *consume chips*

- *consume hamburger*
- *consume milk*

The separation of verbs and nouns in language minimizes the number of definitions needed to communicate a concept. In the example above, a dictionary requires definitions for two verbs and four nouns. A language that did not distinguish between verbs and nouns but maintained a separate concept for each function (getSalad, getChips, ...) would require a total of 8 definitions. In the general former case, the number of definitions needed is given by (#verbs + #nouns), whereas in the latter case it is (#verbs × # nouns). As the number of verbs and nouns grows, this obviously becomes significant. If, for example, there are 10 verbs and 100 nouns in a system, the total number of definitions required in the former case is only 110, rather than 1000 in the latter case. A dictionary for a language structured using the latter approach would quickly become unwieldy, as illustrated in Figure 4.

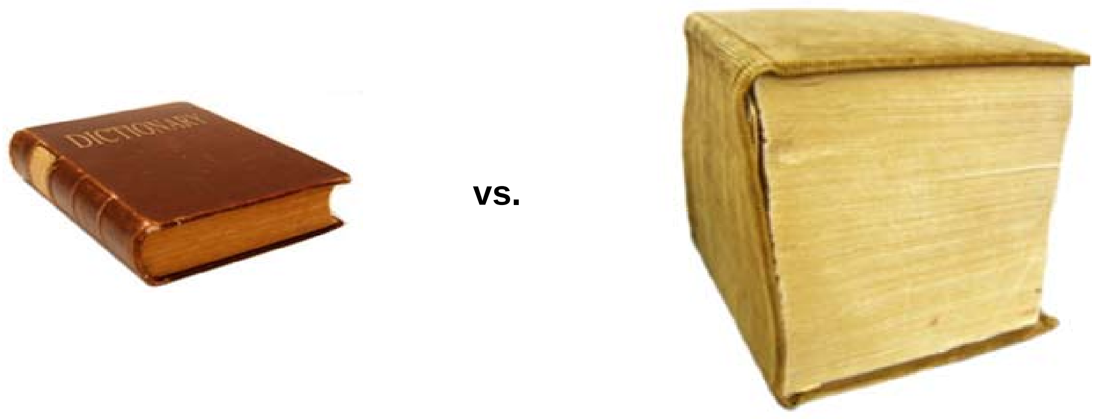

Figure 4, Separation of verbs and nouns in language reduces the number of total terms that must be defined in a concept dictionary.

Related to the reduction in the number of definitions needed is the fact that the separation of verbs from nouns results in the *serendipitous reuse* referred to in the previous section, as one can use a general verb like *get* to apply to abstract nouns like *thing*, without necessarily knowing what *thing* is in advance.

The principle of separation of verbs from nouns, coupled with the recognition that four verbs in particular, Create, Read, Update, and Delete (CRUD), are commonly encountered in software systems, led to the establishment of CRUD as both an acronym and a useful design pattern for interfaces that are intended to be very general. The table below illustrates a number of common interfaces that naturally evolved in accordance with this principle. All of the examples shown have been in wide use for multiple decades, demonstrating the robustness and utility of the approach.

TABLE 1

Common Interfaces Mapped to the CRUD Pattern

VERB	**INTERFACE**			
	HTTP	**FTP**	**SQL**	**UNIX/POSIX API**
CREATE	POST/PUT	PUT	INSERT	create()/open()/write()
READ	GET	GET	SELECT	read()
UPDATE	PUT	PUT	UPDATE	write()
DELETE	DELETE	DELETE	DELETE	unlink()

The HTTP form of CRUD: POST, GET, PUT, DELETE, is often referred to as the REST 'Uniform Interface.' This uniform interface provides numerous benefits that are familiar to all Web users, such as the ability to download a file of any type via a Web browser, without the browser having to be aware of the file format. Just as important to the overall functionality of the web is the fact that the uniform interface is cache-friendly. Deployed caching systems can remain unaware of the types of data being cached, and new data types do not require deployment of updated caching software.

There is nothing inherent in the SOA approach that prevents alignment with the CRUD principle. The Amazon S3 data storage service (see Figure 5) has two interfaces, one based on traditional HTTP, and one based on SOAP. In both cases, the interfaces support the same 4 verbs, and are data-agnostic. This benefits the service in two ways – not only is the service data agnostic, but since the SOAP and 'classic' HTTP interfaces are, in a sense, impedance matched, supporting both interface styles with a single underlying service implementation becomes very straightforward. Alignment of the interface approaches provides benefits in the form of software reuse.

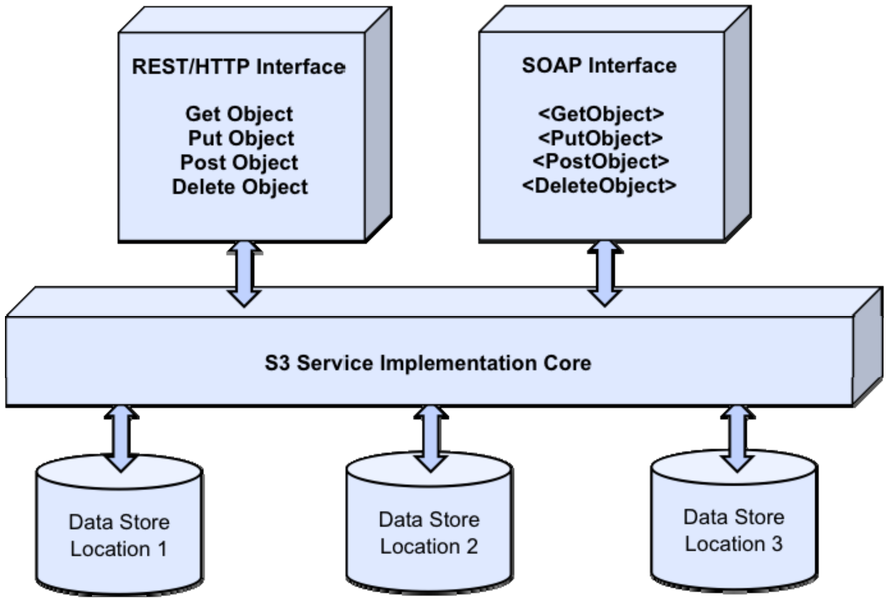

Figure 5. Amazon Simple Storage Service (S3). Service provides scalable, triple redundant data storage as a network service. Traditional REST-style as well as SOAP-style interfaces are supported. Both interfaces are aligned with the CRUD principle, allowing them to be supported by common logic in the service core.

A counter example is the Universal Description, Discovery and Integration (UDDI) interface specification, which follows the model in Erl's example. A subset of the operations supported by UDDI is shown in Table 2. As can be seen, the UDDI Application Programming Interface (API) does not follow the CRUD pattern, resulting in a relatively rigid interface. The addition of support for any new data type requires the creation of 4 new operations for that data type. The approach also, as noted earlier, results in a large interface 'dictionary,' reflected in the 420 page UDDI specification.

TABLE 2

UDDI Interface Fine-Grained Operations

find_business	**save_business**
find_binding	**save_binding**
find_service	**save_service**
find_tModel	**save_tModel**
get_businessDetail	**delete_business**
get_bindingDetail	**delete_binding**
get_serviceDetail	**delete_service**
get_tModelDetail	**delete_tModel**

It is worth noting that the HP Systinet product, when adding a repository capability and interface, chose to not extend the UDDI interface, but to use a REST-style interface (based on ATOM) instead. This is a real-world example of the move to the more RESTful-SOA approach.

A second counter example is that of the initial proposed En Route Automation Modernization (ERAM) interfaces for SWIM. These interfaces, as described in draft Web Service Description Documents (WSDDs) [9,10], follow the fine-grained anti-pattern presented in Erl's text, rather than separating the nouns and verbs and adopting a more uniform approach. As a result, multiple WSDD documents that essentially describe the same *create*, *read*, *update*, and *delete* semantics (albeit for different data types) exist, and applications that wish to access ERAM data of different types must include support for these multiple interfaces, in addition to multiple data types. This runs counter to the notion of *shared services*. The resulting interface complexity, if extended to other data types in SWIM, will result in higher overall system development and maintenance costs.

2.2 MESSAGE-ORIENTED MIDDLEWARE

2.2.1 Background

The concept of Message-Oriented Middleware (MOM) is well-aligned with the goals of SOA due to focus on loosely coupled systems. Two popular (MOM) variants are the Java Messaging Service (JMS) and the Data Distribution Service (DDS), which grew out of different spaces. The JMS standard has its roots in the Enterprise space and is an attempt to provide a uniform application programming interface to historically diverse messaging solutions available from a number of vendors. DDS grew out of the defense and aerospace sectors, was initially focused more on hard real-time, fault-tolerant defense-related applications, particularly shipboard battle control systems.

Though similar in some respects, there are significant differences between the two standards. JMS is largely focused on the messaging layer, treating data in a relatively opaque fashion (though some support is provided to distinguish between textual and binary types). The standard does not specify a wire-level protocol. Different JMS implementations, such as Apache ActiveMQ and Oracle's JMS solution, for example, each utilize separate wire formats, resulting in limited interoperability between different vendor solutions. Apache ActiveMQ itself supports a number of different wire formats, such as the ActiveMQ 'OpenWire' protocol and the more general 'Stomp' protocol. At Wide Area Network (WAN) scales, JMS implementations typically rely on a set of distributed message brokers, configured in a hub and spoke topology.

DDS, like JMS, originally lacked a wire-level protocol, with multiple vendors each supporting their own implementation. Over time, driven in part by open systems requirements of the defense community, a wire-level protocol was developed. The first version of the wire-level protocol was released in 2006, two years after the core DDS specification. DDS, being more concerned with fault-tolerance as the 'norm' in battle systems, is based on more of a peer-to-peer topology than JMS, with no reliance on intermediate message brokers (points of failure). At WAN scales, DDS still essentially functions in a peer-to-peer manner, though some implementations introduce active DDS-specific router components at the LAN/WAN transition points.

DDS differs from JMS in that the specification is not confined to the messaging layer but reaches into the application layer, requiring that data structures be defined in accordance with the DDS standard. DDS borrows some of its data structure framework, such as the use of the Interface Definition Language (IDL) from the CORBA standard, another work maintained by the Object Management Group. This requirement may be a stumbling block for DDS adoption within the FAA, at least in the agency-wide sense.

Of these two standards, SWIM has focused to date on JMS. The lack of an on-the-wire standard is recognized as an important issue, the resolution of which is considered a worthy long-term goal [see Section 2.2.3 on the emerging Advanced Message Queueing Protocol (AMQP) standard]. In the relatively near term, JMS implementations from multiple vendors are envisioned to interoperate via JMS adaptors. Within the Eurocontrol community, DDS is seeing some adoption. One example is its selection as a core technology within the European Flight Data Processor (FDP), scheduled to be available in the 2015 time-frame. Interoperability with Eurocontrol systems in the future will likely require some awareness of DDS on the FAA's part, even if no FAA systems choose to adopt it.

2.2.2 Extending the Uniform Interface for Publish/Subscribe

The importance of the uniform interface based on the 4 basic CRUD operations, following the principle of separation of nouns and verbs, was discussed previously. The question with respect to integration of the MOM approach is how well does the approach align with the uniform interface principle? Very naturally, it turns out. MOM-based systems typically treat data in an opaque fashion, using the well known verbs ‘publish’ and ‘subscribe,’ typically along with a few other verbs such as ‘renew’ and ‘unsubscribe.’ Table 3 is an expanded version of Table 2, illustrating the overall uniform interface mapping when the publish and subscribe verbs are added.

TABLE 3

Uniform Interface Including Publish/Subscribe

VERB	INTERFACE				
	HTTP	FTP	SQL	UNIX/POSIX API	JMS
CREATE	POST/PUT	PUT	INSERT	creat()/open()/write()	
READ	GET	GET	SELECT	read()	
UPDATE	PUT	PUT	UPDATE	write()	
DELETE	DELETE	DELETE	DELETE	unlink()	
PUBLISH	NOTIFY*	—	—	send()	send, publish
SUBSCRIBE	SUBSCRIBE*	—	—	recv()	receive

* Indicates new uncommon use for this particular application.

It should be noted that some in the REST community maintain that the 4 CRUD verbs *are* the uniform interface, and all interactions should be done using them. It is possible, it is argued, to convey the semantics of the verb *subscribe* using the verb *create* coupled with a *resource* named ‘subscription.’ This is indeed possible, but is, in essence, a form of ‘verb tunneling’ through a CRUD-limited verb set. The underlying motivation for the approach at least partially results from the fact that though it is possible to define additional HTTP verbs (the WebDAV HTTP extension is one well-known example), HTTP firewalls are often configured to pass only the 4 well-known verbs. This is an HTTP-protocol driven argument, which, while perhaps practical, we consider to be an implementation detail. In terms of higher-level architectural abstraction, we prefer to view the incorporation of the MOM style as an additional set of verbs. How the verbs are actually mapped to a particular protocol such as HTTP is a lower-level issue.

2.2.3 Advanced Message Queuing Protocol

The importance of standard wire protocols to interoperability cannot be overstated. In addition to addressing basic interoperability concerns, they provide: insulation from vendor lock-in, language independence, and (typically) availability of open source implementations. The Stomp protocol is an example of a simple on-the-wire publish/subscribe standard, which, though relatively limited in its capabilities, has seen significant adoption. The benefits of the on-the-wire standardization approach are evident from the Stomp Web page [11]:

Pick the right Stomp client for your particular language or platform...

- *C*
- *C++*
- *C# and .Net*
- *Delphi*
- *Erlang*
- *Flash*
- *Java*
- *Objective-C*
- *Perl*
- *PHP*
- *Python*
- *Ruby*
- *Smalltalk*

Clearly reflected in this set of available client side packages is the fact that, though Java is currently an important and popular language, we do not exist in a Java-only world. One can expect, for example, that future NextGen applications will often be rapidly composed using higher level scripting languages, perhaps leveraging modules written in Java but not restricted to Java. The availability of a wire-level protocol for publish/subscribe is key to enabling such usage.

The Advanced Message Queuing Protocol (AMQP) is a protocol that intends to fill this gap. RedHat, one of the members of the AMQP working group, puts it this way on their Web page [12]:

"There is a large gap in the current networking protocol standards; nearly every case has been tackled except the most common guaranteed-delivery messaging middleware. Such middleware is the underpinning of both back and middle office systems, in every automated business process in medium to large enterprises, and is a necessary utility. JMS offers a partial solution, but is limited by its reliance on Java and its inability to specify any wire-level interoperability. A solution is this space is required to be an open work and be able to be used from any platform and any language.

This lack of open protocol standards means that interoperability between existing proprietary solutions is poor, and new entrants to the market have to reinvent even the most basic facilities before they can begin to add their own innovations. The AMQProtocol can be integrated into existing products or provide open interoperability for API's like JMS. The AMQProtocol can be used with most of the current messaging and Web Service Specifications such as JMS, SOAP, WS-Security, WS-Transactions, and many more, complimenting the existing work in the industry. AMQP will also provide specified routing to and from multicast for subnet optimizations or grid type deployments."

Since the initiation of the effort, the AMQP working group has made steady progress, with the first release of a draft specification in 2006, follow-up releases in 2008, and a version 1.0 draft in 2009. There are multiple implementations of AMQP-compliant systems in existence today, both of the open source and commercial variety. The Apache QPid project, based in part on code donated by Red Hat, is intended to fully implement the 1.0 specification. The Apache JMS implementation, ActiveMQ, is planning on supporting AMQP in a future release (likely leveraging the QPid project's work), according to the project's Web page.

There is a possible relationship between DDS and AMQP. The DDS wire standard is User Datagram Protocol (UDP)-based today, but there is interest in also implementing a Transmission Control Protocol (TCP)-based protocol for use at WAN scales. Given that AMQP builds upon TCP, there is some interest in the DDS community in adopting AMQP as the TCP wire-level protocol for DDS. If this is possible, and JMS vendors also implement AMQP, this would result in a nice, coherent system, with AMQP serving as the HTTP-equivalent for MOM-based data distribution. We note that such an outcome, though desirable, is by no means assured.

Given the importance of messaging to NextGen and SWIM, we recommend some level of prototyping be conducted using DDS and AMQP, to better understand the pros and cons of the technologies. This is not a critical near-term need, but lessons-learned, even ones that do not result in technology adoption, would likely be beneficial in a SWIM context, particularly when it comes to understanding the Eurocontrol approach to data distribution in more detail.

2.2.4 MOM Support for Filtering

JMS and DDS both support filtering on a number of header properties, using a Structured Query Language (SQL) 92-like syntax. In JMS, *selectors* are used to configure the filtering capability. An example filter expression from the ActiveMQ JMS documentation that filters on the standard *JMSType* property, as well as two custom properties, is given below:

```
JMSType = 'car' AND color = 'blue' AND weight > 2500
```

All JMS implementations support this basic filtering functionality. Some implementations have additional filtering capabilities that examine not only the JMS headers. ActiveMQ, for example, supports basic XPath filtering expressions for messages whose type is XML. DDS also supports some level of content-based filtering.

While certainly useful, these filtering capabilities are lacking in that there is no support for spatial filtering. A more comprehensive approach to filtering, capable of supporting the request/response service interaction model as well as the publish/subscribe approach, is desirable. This is the topic of the next section.

2.3 FILTERED DATA ACCESS

In a system with a large variety of data types, a large geographical extent, and a need to access potentially large volumes of archived as well as real-time data, the need for filtered access to the data will always be present. This is certainly true of the NAS, and early references to SWIM often included the phrase '*What you want, When you want it, Where you want it,*' echoing the Department of Defense's (DoD) '*What,When,Where*' phrasing. Ideally, this filtering capability would be as common as possible across all data access services, to achieve the SOA interoperability and software reuse benefits.

A list of common filtering requirements, arranged in order of increasing complexity, is shown below:

- Filtering by data type (flight track, weather intensity)
- Filtering by string properties (site identifier)
- Filtering by simple time properties (data observations between time T1 and T2)
- Filtering by simple spatial properties (bounding box, radius, polygon, named region)
- Filtering by complex time properties (weather forecast for time T2, generated at T1)
- Filtering by complex spatial properties (access weather data along a flight corridor)

As the filtering operations become more complex, they generally become more difficult to implement. In addition, as the complexity grows, the generic quality of the filtering operation in the sense of being data-agnostic is often reduced. This leads to a situation where not all data providers will wish to incur the overhead of a data access service that supports all the listed types of filtering. A coherent approach to managing this is to use a composable, 'core + extensions' approach. In this approach a filtering syntax is composed of a core that is implemented across the board, and additional capabilities are included as needed on a service-by-service basis. In order to achieve a good level of interoperability, a number of conformance *profiles*, specifying common groups of extensions along with details regarding which optional operations are implemented, are established within an organization.

The core + extensions approach is not new, and has been successfully adopted in a number of real-world cases. For example, the OpenSearch specification, in defining a common, cross-browser, search API, breaks down search capabilities into a common core and extensions as shown below [13]. Base-level interoperability is provided by the common core, and finer-grained capabilities for smaller communities-of-interest are provided via the extensions. Numerous other specifications follow this approach, including many of the Open Geospatial Consortium (OGC) specifications as well as the WS-Notification specification.

- OpenSearch Core 1.1
  - OpenSearch Description Document
  - OpenSearch URL Template Syntax
  - OpenSearch Query Element
  - OpenSearch Response Elements

- OpenSearch Extensions
  - OpenSearch Referrer Extension
  - OpenSearch Relevance Extension
  - OpenSearch Parameter Extension
  - OpenSearch Geo Extension
  - OpenSearch Time Extension
  - OpenSearch Commerce Extension
  - OpenSearch Mobile Extension

Following the core + extensions approach, the filtering capabilities described above are depicted in Figure 6 using a layered, core + extensions model. In the case of temporal and spatial filtering support, the more complex operations (i.e., 3-D) tend to build upon the simpler operations, resulting in a multi-layered hierarchy. Note that as is the case with object-oriented programming, it is a good practice to avoid deep hierarchies (inheritance) to maintain flexibility with respect to how the extensions can be composed into the desired groupings for a particular application.

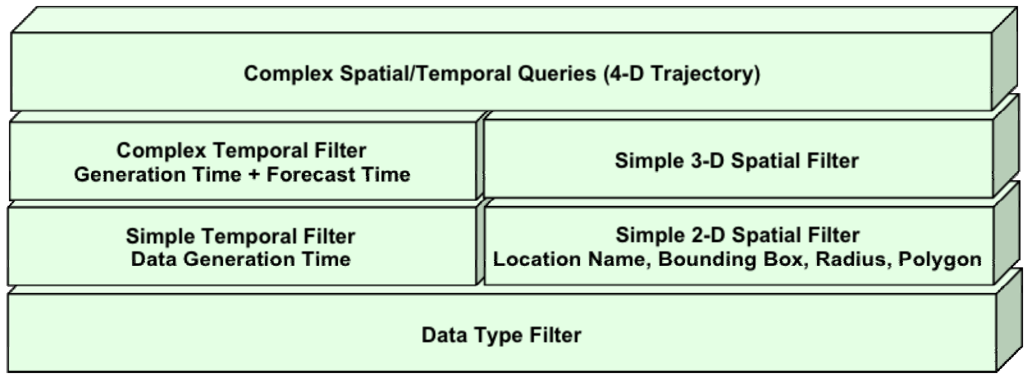
Complex Spatial/Temporal Queries (4-D Trajectory)
Complex Temporal Filter Generation Time + Forecast Time
Simple 3-D Spatial Filter
Simple Temporal Filter Data Generation Time
Simple 2-D Spatial Filter Location Name, Bounding Box, Radius, Polygon
Data Type Filter

Figure 6. Filtering components in a core + extensions model

The filtering discussion has thus far been quite abstract. Two queries are provided below as examples of how the notion of core + extensions can be commonly implemented in actual practice. These are both based on the OGC Web Feature Service, which leverages the OGC Filter Specification. In the first query, the caller is interested in weather cells that fall within a certain time period and bounding box. All operations shown are supported by the Web Feature Service (WFS) core specification.

```
<wfs:GetFeature
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:wx="http://www.eurocontrol.int/wx/1.1"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:gml="http://www.opengis.net/gml/3.2" >

  <wfs:Query typeNames="wx:Contour">
    <fes:Filter>
      <fes:And>
        <!-- Contour sub-type -->
        <fes:PropertyIsEqualTo>
          <fes:ValueReference>gml:name</fes:ValueReference>
          <fes:Literal>http://www.ll.mit.edu/2009/storm.owl#WinterVILContour</fes:Literal>
        </fes:PropertyIsEqualTo>

        <!-- Temporal query elements -->
        <fes:After>
          <fes:ValueReference>wx:obsOrFcstTime/gml:TimeInstant</fes:ValueReference>
          <gml:TimeInstant gml:id="TI1">
            <gml:timePosition>2010-06-07T21:50:30Z</gml:timePosition>
          </gml:TimeInstant>
        </fes:After>
        <fes:Before>
          <fes:ValueReference>wx:obsOrFcstTime/gml:TimeInstant</fes:ValueReference>
          <gml:TimeInstant gml:id="TI2">
            <gml:timePosition>2010-06-07T22:01:00Z</gml:timePosition>
          </gml:TimeInstant>
        </fes:Before>

        <!-- Spatial query elements -->
        <fes:Intersects>
          <fes:ValueReference>wx:geometry/gml:Curve</fes:ValueReference>
          <gml:Envelope srsName="urn:ogc:def:crs:EPSG::4326" srsDimension="2">
            <gml:lowerCorner>20.0 -120.0</gml:lowerCorner>
            <gml:upperCorner>40.0  -95.0</gml:upperCorner>
          </gml:Envelope>
        </fes:Intersects>

      </fes:And>
    </fes:Filter>
  </wfs:Query>

</wfs:GetFeature>
```

In the second example, the WFS filtering capabilities have been extended to include queries relevant to NextGen trajectory-based operations. The extension is specified in its own XML schema with its own namespace (shown in italics), and referenced in the query in a similar fashion to the core namespace elements.

```
<wfs:GetFeature service="WFS" version="2.0.0"
  xmlns:wfs="http://www.opengis.net/wfs/2.0"
  xmlns:wx="http://www.eurocontrol.int/wx/1.1"
  xmlns:fes="http://www.opengis.net/fes/2.0"
  xmlns:faa-ext="http://www.faa.gov/fes/faa-ext/1.0"
  xmlns:gml="http://www.opengis.net/gml/3.2" >

  <wfs:Query typeNames="wx:Contour">
    <fes:Filter>
      <fes:And>
        <!-- Contour sub-type -->
        <fes:PropertyIsEqualTo>
          <fes:ValueReference>gml:name</fes:ValueReference>
          <fes:Literal>http://www.ll.mit.edu/2009/storm.owl#WinterVILContour</fes:Literal>
        </fes:PropertyIsEqualTo>

        <!-- Complex trajectory intersection query (FAA extension) -->
        <faa-ext:IntersectsTrajectory>
          <!-- Temporal and spatial properties for intersection test -->
          <fes:ValueReference>wx:obsOrFcstTime/gml:TimeInstant</fes:ValueReference>
          <fes:ValueReference>wx:geometry/gml:Curve</fes:ValueReference>
          <!-- 4-D trajectory (time + lat/long/alt)-->
          <gml:Curve srsName="urn:ogc:def:crs:EPSG::4791" srsDimension="4">
            <gml:segments>
              <gml:LineStringSegment>
                <gml:posList count="88">
                  2010-06-07T21:50:30Z  21.1599 -83.1743 1000.0
                  ... 87 more 4-D trajectory points
                </gml:posList>
              </gml:LineStringSegment>
            </gml:segments>
          </gml:Curve>
        </faa-ext:IntersectsTrajectory>

      </fes:And>
    </fes:Filter>
  </wfs:Query>

</wfs:GetFeature>
```

The practice of using separate XML namespaces for community extensions, as shown here, helps ensure that the composable nature of the specifications is maintained over time. The extensions appear to be first-class citizens, siblings to the core operations rather than existing in a parent-child relationship. This is made possible by the use of a relatively open XML content model, with well-defined extension mechanisms that do not place particularly stringent constraints on where the extensions are placed, as long as they abide by the underlying filtering language framework. The use of an open content model is key to the creation of reusable XML data models, as evidenced by a number of popular standards that have adopted the approach (e.g., the ATOM Really Simple Syndication [RSS] specification). Note that the importance of this approach to helping models evolve gracefully over time is recognized in the latest release of XML Schema (1.1), which contains explicit support for open content models without having to manually sprinkle a schema with references to XML *other* elements (as is currently the case in XML Schema 1.0)

In order for the principle of core + extensions to work properly, service instances must have a means of providing metadata describing which extensions are supported, to allow clients to select the service that fits their requirements. This metadata is reasonably static in nature, and can be managed at design time or at run time, depending on the needs of a particular NextGen application. With respect to the examples shown, the OGC services do provide such a mechanism via the *GetCapabilities* operation.

2.4 CONTENT DELIVERY NETWORKS

2.4.1 Background

The FAA Telecommunications Infrastructure (FTI) includes a modern optical backbone, shown in Figure 7. The backbone currently consists of a mixture of 2.5 GBps and 10 GBps links, with the higher rate generally used in the eastern third of the country to reflect the higher air traffic density in that region. The command center and all enroute centers are directly connected to the backbone, as is the FAA Technical Center. FAA Terminal Radar Approach Controls (TRACON) and Towers generally fan out from these backbone nodes or a number of other points-of-presence.

The network bandwidth is a shared resource, and supports legacy point-to-point interfaces, voice communications, and other protocols in addition to IP traffic. The breakdown of the bandwidth into the various categories is shown in Figure 8 (note that at the time of this writing, the Quality-of-Service levels shown are not yet in common use). As can be seen from the figure, only 1 Gbps of bandwidth is allocated to IP traffic, even in the case of the 10 Gbps links. Of this, FTI estimates that 650 Mbps is actually usable, with the remainder reserved for protocol overhead.

A question for SWIM is how to best architect shared data access services to make optimum use of this underlying network, minimizing data access latencies while at the same time minimizing the overall bandwidth requirement. This problem is not unique to SWIM, and is encountered in other public, commercial, and government (e.g., DoD) networks. One class of solutions that addresses this issue is the Content Delivery Network (CDN), typically consisting of a set of active components that reside in the network below the application layer and serve to accelerate Web access. Many CDNs are by design transparent to data providers and consumers alike, behaving more or less identically to the ‘plain-old-Internet,’ only with higher performance. CDNs belonging to this group are often referred to as *overlay networks*.

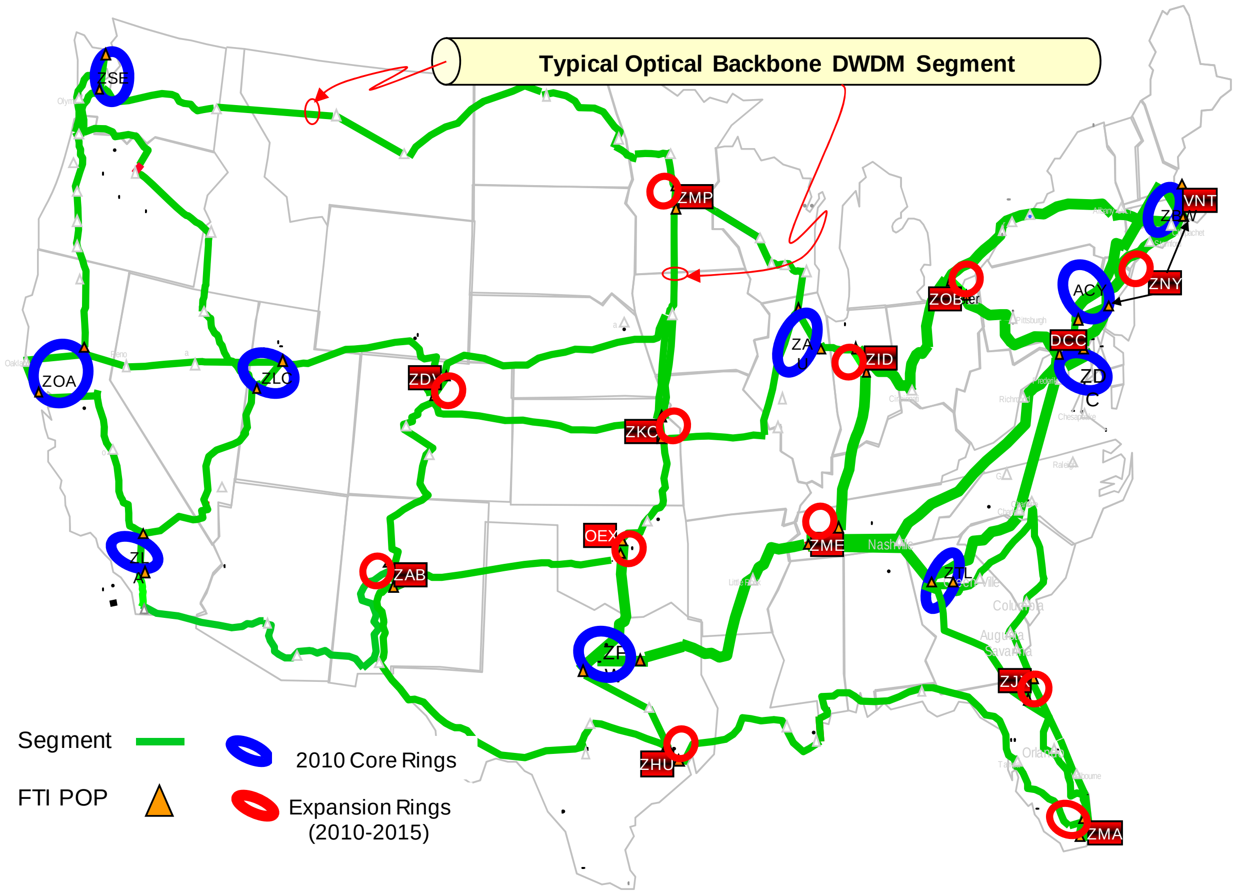

Figure 7. FTI optical network backbone. Backbone is a combination of 2.5 Gbps links in regions with less air traffic and 10 Gbps in the more heavily trafficked eastern third of the country. The command center and all en-route centers are on the backbone. TRACON and Tower facility connect via those and other FTI points-of-presence.

Notably, FTI does not support multicast protocols at the network level. This is due to a number of factors, not the least of which is the need for routers to maintain tables of relatively complex multicast routing information, leading to resource issues for routers as well as an increase in network administration maintenance overhead. This reluctance to support multicast is true of many networks, not just FTI. Given the potential for cost savings related to bandwidth in cases where a large amount of data is fanned out across the network, this limitation may be worth revisiting, if for nothing else to ensure that all programs understand in detail why multicast is not supported.

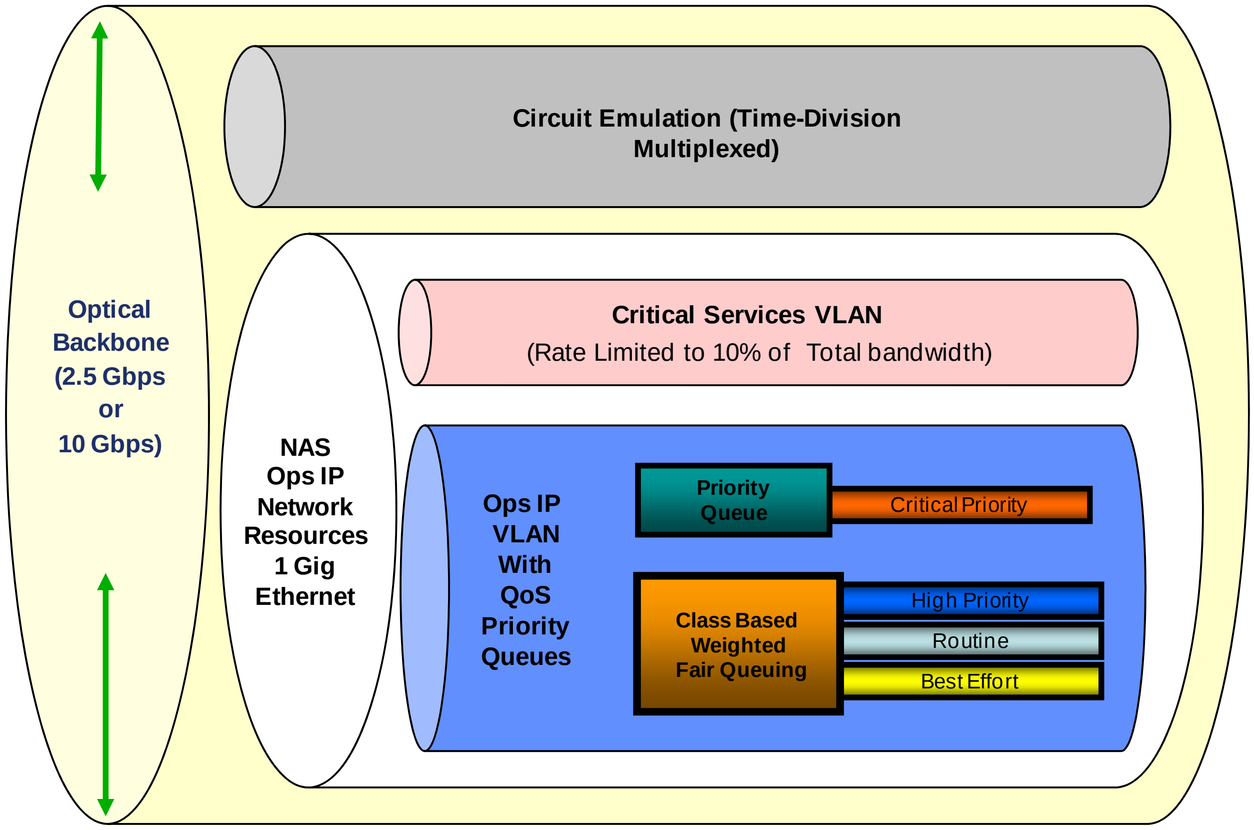
Circuit Emulation (Time-Division Multiplexed)
Optical Backbone (2.5 Gbps or 10 Gbps)
NAS Ops IP Network Resources 1 Gig Ethernet
Critical Services VLAN
(Rate Limited to 10% of Total bandwidth)
Ops IP VLAN With QoS Priority Queues
Priority Queue
Critical Priority
Class Based Weighted Fair Queuing
High Priority
Routine
Best Effort

Figure 8. Network bandwidth partitioning. A significant portion of available bandwidth is reserved for point-to-point communications, addressing sensor communications as well as voice communications. Currently, 650 MBps is available for Internet Protocol traffic.

2.4.2 Akamai

Akamai is a commercial CDN in wide use as a general network accelerator, primarily focused on static Web page content. Though Akamai is branching out into a number of different solution spaces, their core solutions continue to depend on a deployed infrastructure that includes over 27,000 servers deployed in more than 70 countries worldwide. As shown in Figure 9, the basic concept is relatively straightforward – the caching of data on servers that are closer to end users results in reduced data access times. In addition, Web site owners need not scale their systems to handle large numbers of users, as the scalability is provided by the Akamai infrastructure. End users are insulated from the details of the infrastructure in large part by the use of Domain Name System (DNS) features that allow for clients to be automatically redirected to a local caching server when accessing an Akamai-enabled Web site. The majority of end users remain completely unaware of the infrastructure between them and the Web site's origin server.

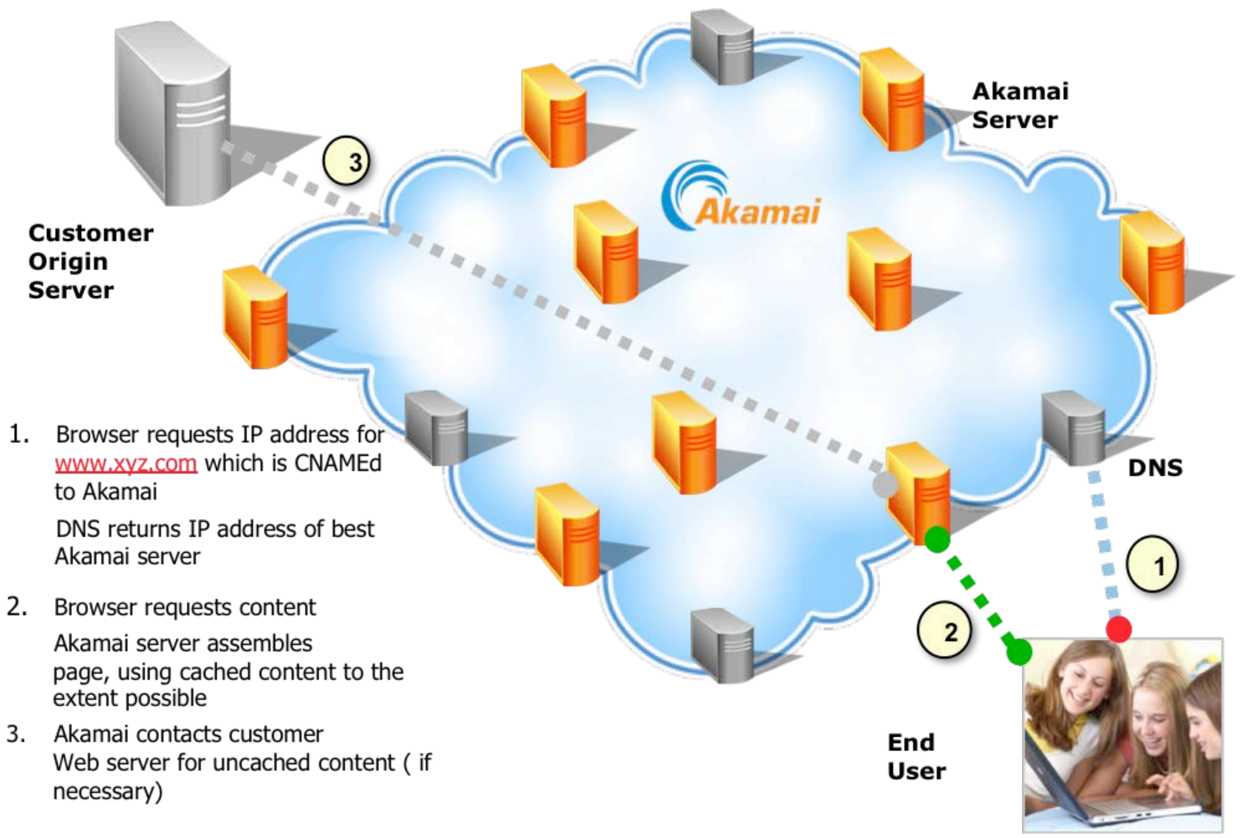
Customer Origin Server
Akamai Server
Akamai
DNS
End User
3
1
2
1. Browser requests IP address for www.xyz.com which is CNAMEd to Akamai
DNS returns IP address of best Akamai server
2. Browser requests content
Akamai server assembles page, using cached content to the extent possible
3. Akamai contacts customer Web server for uncached content (if necessary)

Figure 9. Akamai Content Delivery Network. Content is cached near the network edge to improve access times and reduce load on customer Web sites. Over 25,000 Akamai edge servers are deployed throughout the Internet at key locations.

Akamai is relatively sophisticated in that the deployed servers are used to measure and monitor performance of the various paths throughout the network, and the information is used to dynamically adjust the routing to maximize performance. A number of other optimizations, such as the use of persistent TCP connections between Akamai servers to avoid TCP connection startup overhead, are employed as well.

Though most well-known for its overlay on the public Internet, Akamai does have a presence in private networks as well. It is used by the DoD, for example, to accelerate Web performance at DoD installations worldwide.

2.4.3 Other Content Delivery Networks

The number of CDNs continues to grow over time. Driven to a large extent by the explosion of streaming media, other CDNs such as Limelight and EdgeCast have entered the space and become major players. The newer CDNs tend to focus more on provisioning large data centers with very wide data pipes than relying solely on the public Internet backbone. The practicality and relatively low cost (compared with a number of years ago) of building out a private backbone makes this possible. Basically, as core network bandwidth increases propagate closer to the edge, the need for large numbers of servers very near the end consumer is reduced. The need to isolate a CDN's customers from large numbers of consumers remains as before, leading to the large data center approach.

2.4.4 CDNs and the End-to-End Principle

By adding components to the network between the application layer and the network router layer, some might argue that CDNs in some sense violate the Internet's 'End-to-End' principle, which recommends that the core network (i.e., below the application level) is kept as simple as possible to maintain the flexibility and large-scale interoperability of the network. Creating dependencies on additional functionality within the core can indeed result in CDN-induced 'stovepipes,' where applications depend on features of a single CDN implementation in order to function. This is obviously an undesirable outcome.

In recent years, numerous articles have been written questioning whether the end-to-end principle still applies in today's network environment. Putting those discussions aside, it is generally agreed that performance enhancements that are difficult to achieve at network endpoints can be considered valid exceptions to the principle. The 1981 defining paper *End-to-end arguments in system design* [14] states:

> *"...The principle, called the end-to-end argument, suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level. Examples discussed in the paper include bit error recovery, security using encryption, duplicate message suppression, recovery from system crashes, and delivery acknowledgement.* ***Low-level mechanisms to support these functions are justified only as performance enhancements****."*

Since the focus of CDNs is enhancing of network performance, their introduction of intermediary data transport components is by this definition not a violation of fundamental Internet design principles. Further, the relative transparency of the components in many CDN implementations means that if they are removed, the applications will continue to function, albeit at a lower performance level.

2.4.5 CDNs and SWIM

Historically focused on Web applications, CDNs have largely addressed content of a primarily static nature. Though this is changing to some extent when streaming of live, or near-live content, this is still largely true today. The core SWIM concern with respect to data access services tends to be more dynamic – surveillance data for example changes in the 1-second time frame, and much of the available weather content changes on the 30 second to 5 minute timescale. This is not true of all information relevant to SWIM – aeronautical information such as runway geometries is, by contrast, relatively static. Nevertheless, CDNs are in some sense not ideally matched to the problems that SWIM is trying to solve. In addition, popularly available CDNs are not 'SWIM-compliant' in the sense of being well aligned with the SOA technology base.

Though not necessarily directly usable, there are a number of concepts embodied in CDNs that are relevant to SWIM:

- ***Partitioning a data delivery network into origin servers and distribution servers***. This improves overall data access performance, while at the same time reducing demands on both the data provider and the intermediate network. A network of distributed message brokers in a publish/subscribe system is closely aligned to this concept, though to be most effective the overall idea should apply to both request/response and publish/subscribe message exchange patterns.

- ***The use of an indirect run-time lookup mechanism to determine the most appropriate distribution server to contact to access a particular dataset.*** A number of CDNs leverage the DNS subsystem to satisfy this performance-related objective, as well as managing the location of backup servers in case a server at one location should fail. Akamai, for example, leverages the DNS Canonical Name (CNAME) facility for this purpose. This strategy may be more or less directly reusable within SWIM. An alternative or complementary DNS facility that may be preferable to the use of CNAME in some respects is the DNS Service Record (SRV) facility. DNS SRV records are finer grained than CNAMES, allowing for service type-specific routing strategies to be employed rather than just domain-specific strategies. If adopted, either approach may place certain requirements on the DNS subsystem. If richer metadata than DNS typically provides is needed at runtime to implement the desired lookup behavior, a higher level capability based on an XML-based registry is another alternative.

- ***Instrumentation of network paths to measure network performance in real time.*** This is potentially useful in the early detection of fault conditions. Some CDNs (e.g., Akamai) use this information to route data via the best path in a highly dynamic fashion. This is likely ***not*** a good approach for SWIM, as some of the key security capabilities envisioned for the NAS are expected to rely in part on the relatively static nature of the data flows in the FAA environment. Data flows that were more dynamically routed would render this strategy ineffective.

The trend in CDNs towards a smaller number of large data centers in place of a large number of deployed servers, driven by increases in overall bandwidth and the gradual build-out of high bandwidth links closer to the network edge, is of interest to SWIM in that it indicates that the number of locations with deployed SWIM data access server hardware may be relatively small (at least when compared with the number of Towers and TRACONs). Increasingly, the solution for small, remote locations with limited numbers of systems that access data may be to increase the size of the link to the facility, rather than installing a caching edge server.

2.5 GOOGLE EARTH/GOOGLE MAPS

While the focus of CDNs is on efficient delivery to potentially large numbers of consumers, the focus of Google Earth and Google Maps is a little different – access to pieces of a large multi-resolution dataset over a wide area network. This is certainly relevant to SWIM and related programs such as Aeronautical Information Management (AIM) and NNEW. Google Maps and Google Earth have different design goals and are discussed separately.

2.5.1 Google Maps

The goal of Google Maps to access and display 2-D multi-resolution earth imagery via an ordinary Web browser. The capability needs to scale to very large numbers of users, delivering only the data needed to the end users to save on network bandwidth. At the same time, the load on the server-side infrastructure must be minimized as much as possible.

The approach taken in Google Maps is to balance these two requirements by using image tiling. Users of Google Maps are familiar with this. When first bringing up a particular view, one can often observe individual tiles being progressively loaded. Figure 10 illustrates a set of notional tiles that are composed together to form a typical map view. As the view is rapidly panned or zoomed, additional tiles are loaded on demand. The tiles are cached once loaded, so subsequent pans back to a previous location require minimal interaction with the server.

Google Maps is a blend of an RPC style interface and a 'pure' REST interface, as can be seen by the fact that the URL in the browser bar remains http://maps.google.com throughout a map browsing session. URL links that encapsulate the current view settings are available via the Link hyperlink in the upper right. This allows the current view to be easily emailed to others, or included as a reference in a complex Web service interaction.

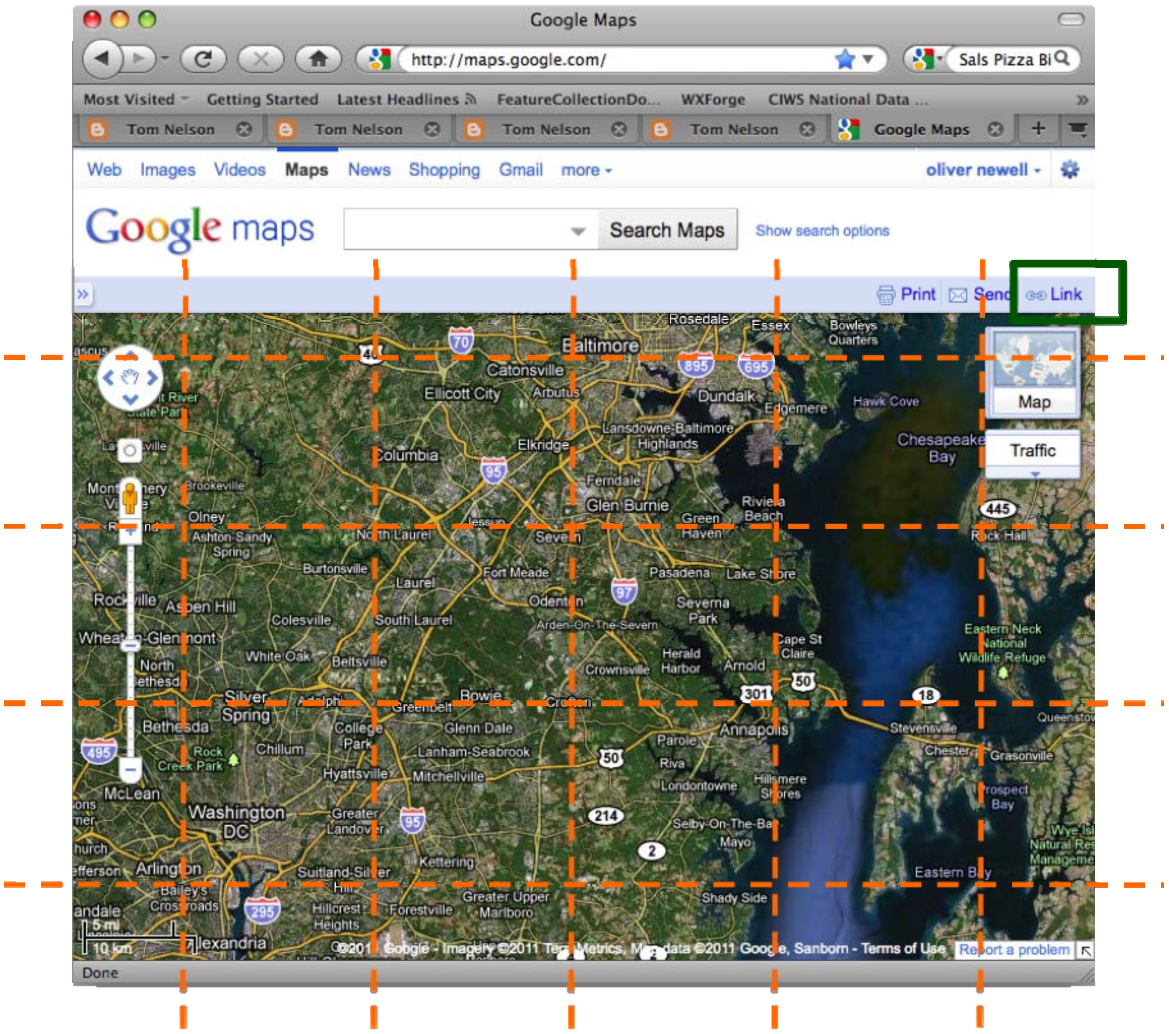

Figure 10. Google Maps sample view. Multi-resolution imagery is stored as tiles to help balance server-side and client-side workload, as well as to leverage caching capability of Web infrastructure. A REST-style URL link capability is provided to allow references to a particular view/settings combination to be easily mailed.

Behind the use of tiling is the notion that consumers are restricted in their queries to spatial regions that are quantized – in other words, the server does not have to provide potentially N unique views for N clients. This saves resources on the server side, allowing a server of a certain size to serve a greater number of clients. From the client perspective, due to the quantization of spatial queries, some of the data received in a response may lie outside the requested region. It then becomes the responsibility of the client to filter out that extra data. In display-oriented applications this is usually trivial, as clipping data lying outside a view window is generally supported by the underlying graphics package. In other applications, a separate client side filtering component is needed. In some cases, the spatial filtering component may be sharable by both the server and the client, as depicted in Figure 11. This is component sharing at a finer-grained level than the service level, but still aligned with the modular, composable SOA philosophy. Note that this approach applies not only to gridded data, but non-gridded data as well.

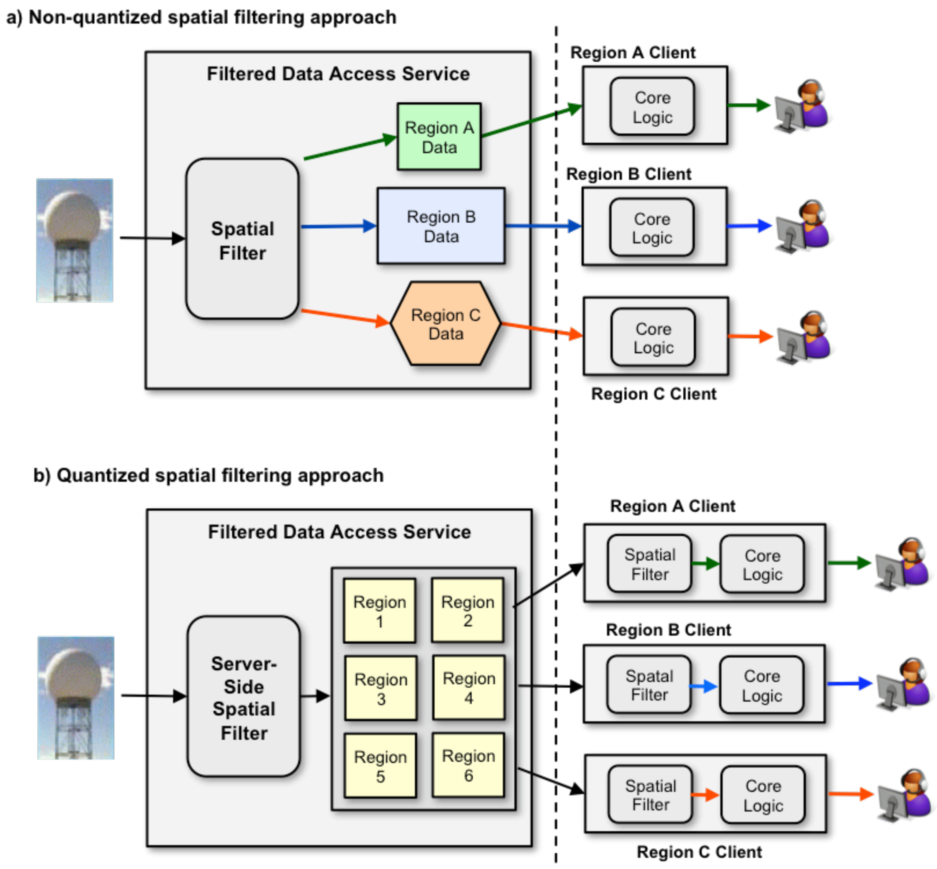

Figure 11. Quantized approach to spatial filtering, abstracted from the tiling approach utilized by Google Maps. Unique spatial filtering on a per-client basis results in a heavy load on the server side. Quantizing the filtering operation on the server side, as shown in b), lessens the load on the server side but does potentially require additional filtering on the client side to achieve the desired result. In the ideal case, a suitably configurable filtering software module can be largely shared by the server and client.

2.5.2 Google Earth

Google Earth supports a more complex data access interface than Google Maps due to its 3-D nature. 3-D datasets are larger, and the workload associated with accessing and viewing data from a variety of perspectives in a dynamic fashion is significantly higher. In order to function efficiently, the data access protocol is closely aligned with data structures that work well with commodity graphics hardware and software (i.e., OpenGL).

Figure 12 illustrates one of the issues that arise in a 3-D application. Depending on the viewer perspective, information at one location in a view window can have significantly different spatial resolution than at another location. In the sample shown, the locations labeled Far Field and Near Field depict the problem. For network efficiency reasons, one does not want to download data at a higher resolution than really needed in the far field. The data access protocol used by Google Earth accounts for this and only the data that is needed is transferred.

The complexity of Google Earth is also evident in the fact that simple REST-style URL references to a particular configured view are not supported, as they are with Google Maps. Instead, one can download a reference to the current view in the form of a Keyhole Markup Language (KML) file – the Google Earth equivalent to HTML for a browser. This prevents the simple sharing of URLs with others as a way to share Google Earth views, but sharing is at least possible, with full fidelity of the original view preserved.

Though the Google Earth application and the KML standard will likely see use in NextGen going forward, in terms of general purpose low-level data access service interfaces there appears to be little in the way of lessons learned that can be leveraged, due to the highly optimized, special purpose nature of the interfaces.

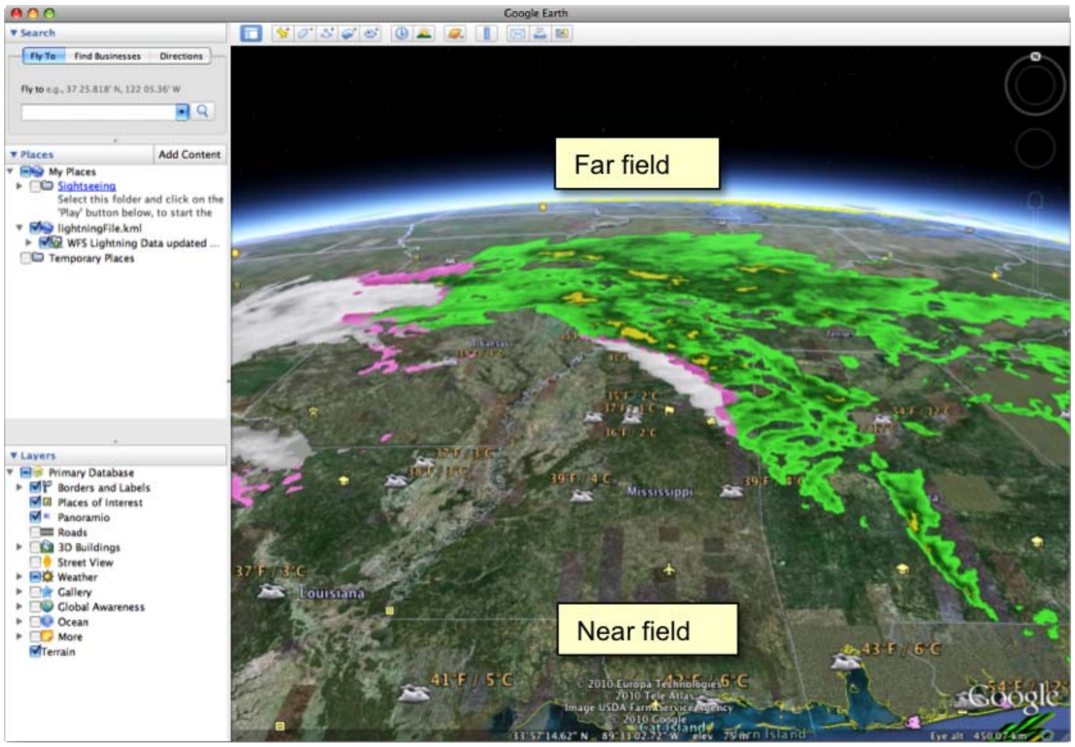

Figure 12. Google Earth sample view. When viewing 3-D data, the desired resolution at different points in the view window is different. Google Earth data access mechanisms take this into account, and download only the resolution needed for each region of the view.

2.5.3 Architectural Survey Summary

This section examined a number of architectural styles and data distribution approaches, with a goal of aligning the concepts embodied in each that are relevant to SWIM under a common SOA ‘umbrella.’ Key architectural principles for consideration by SWIM include the separation of verbs from nouns in general-purpose interfaces, augmented by the use of common filtering semantics that conform to a ‘core + extensions’ model. Following these principles will help avoid the tendency to construct a large, diverse set of ‘point-to-point’ data access services that are stove-piped both with respect to data types and messaging patterns. This helps meet the shared services goal of SWIM Segment 2.

Though minimization of the number of different data access services is desirable, the examples discussed and others clearly illustrate that there is not a ‘one size fits all’ solution for NextGen. Furthermore, a chosen solution for a particular set of data providers and consumers may change over time to reflect changing requirements. The next section discusses these issues in more detail.

3. MODULAR SHARED DATA ACCESS SERVICES

3.1 DATA ACCESS SERVICE HIERARCHY

In a system with such wide-ranging requirements as the NAS, there is no ‘one-size-fits-all’ shared data access service. A number of common interface patterns do exist, however. These can be placed into a hierarchy that spans simple unfiltered data access to the more complex data access that will be needed in a NextGen Trajectory-Based Operations (TBO) environment. A notional set of abstract service types, accompanied by concrete instance examples, is shown in Figure 13. At one end of the spectrum are simple data access mechanisms providing access to (for the most part) unfiltered data. These services do not typically support access to rich metadata via separate, metadata-specific operations.

Service interfaces that support richer metadata, such as ATOM and other forms of feeds, reside at the next level in the hierarchy. These tend to build upon the lower level services, and provide access to metadata and data via operations that are separate from the core data access operations. Services that provide content filtering capabilities, such as those provided by the OGC data access services and the Joint METOC Broker Language (JMBL), reside at a higher level.

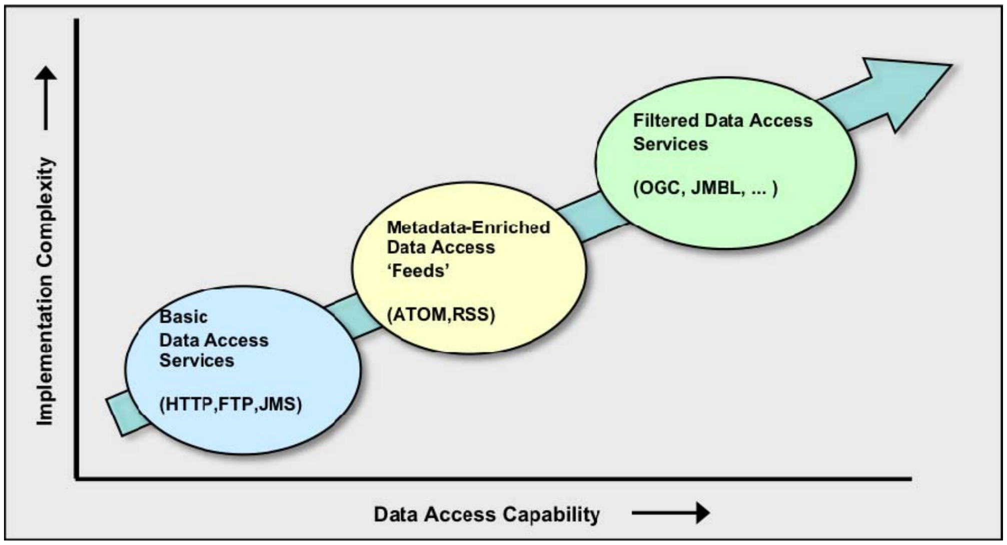
Implementation Complexity
Data Access Capability
Basic Data Access Services (HTTP,FTP,JMS)
Metadata-Enriched Data Access ‘Feeds’ (ATOM,RSS)
Filtered Data Access Services (OGC, JMBL, ...)

Figure 13. Types of data access services.

Note that we refer to HTTP (and File Transfer Protocol [FTP]) as a ‘service’ here even though it is not often viewed in that light. Given that JMS *is* commonly referred to as a service, and HTTP is in some sense a request/response analog to the JMS publish/subscribe data access mechanism, we feel this is appropriate.

Generally speaking, with an increase in capability comes an increase in implementation complexity and effort. It is certainly not desirable to implement a more complex service unless/until the capability is truly needed. On the other hand, given the agility and software reuse goals of NextGen, it is desirable to pursue an approach up-front that allows for services to gracefully evolve, most typically in the direction from simpler to more complex, but also potentially in the reverse direction. With an appropriate set of modular components, it is possible to achieve this flexibility in practice. As an example, the following section ‘grows’ a service from a simple publish/subscribe service that serves only real-time data to a service that supports complex filtered access to both real-time and historical data.

3.2 MODULAR SERVICE EVOLUTION

This section provides an example of an evolutionary approach to construction of data access services, building from a bare-bones publish/subscribe service to a more full-featured service that supports both request/response and publish/subscribe, as well as content filtering. Though the data is shown as originating from a Next-Generation Radar (NEXRAD) sensor, the type of data is largely irrelevant to the discussion, though it is assumed that the volume of data is large enough that content filtering is desirable in some cases.

The baseline service (see Figure 14) used as a starting point is a simple service to provide access to data from a set of one or more radar sensors. Access to data from separate sensors is supported via the use of separate publish/subscribe ‘topics’ for each sensor. Other than a message broker and a suitable adaptor for the radar sensor, little else is needed. The assumption here is that any necessary archiving of data is provided by the sensor system itself, separately from the data distribution infrastructure. Access to data in past, even the recent past, is not provided.

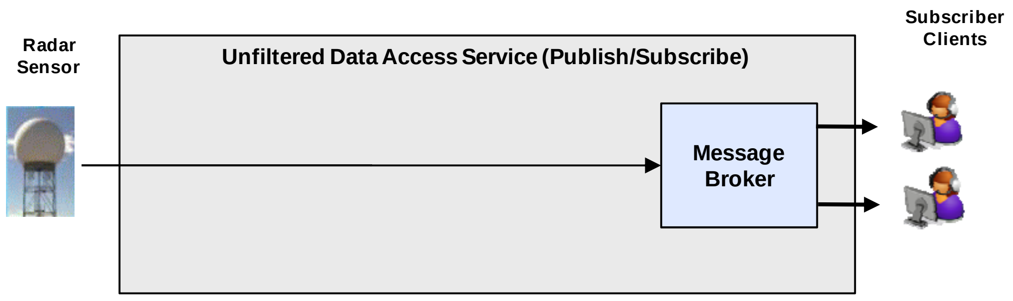

Figure 14. Basic publish/subscribe-based data access service for radar data.

Publish/subscribe-based systems impose a certain overhead due to the fact that the server side must maintain a certain amount of state for each connected client. For scalability reasons, it is often useful to support a simple request/response mechanism as well, due to the stateless nature enabled by that style of interaction. One example where this is relevant to SWIM is in providing data to external consumers. Adopting a stateless interaction model wherever possible lowers the overall burden on the SWIM infrastructure.

Figure 15, below, augments the previous service with a simple HTTP-based service endpoint that provides access to the latest data snapshot. Note that the data store shown is only used for the latest data snapshot, and an interface to access historical data is not provided.

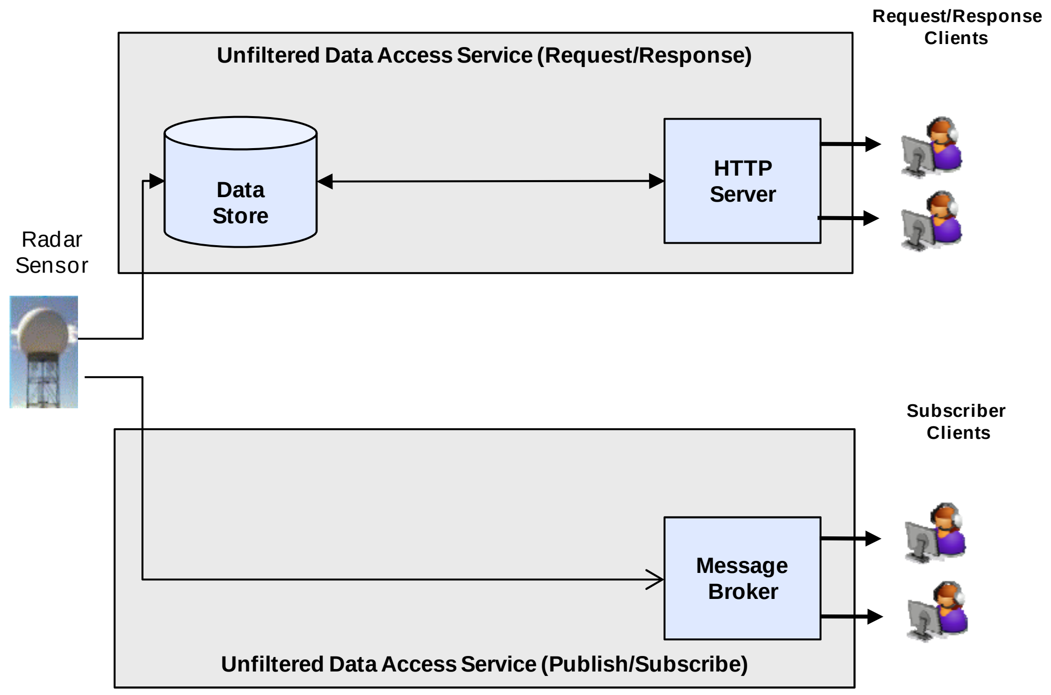
Request/Response Clients
Unfiltered Data Access Service (Request/Response)
Data Store
HTTP Server
Radar Sensor
Subscriber Clients
Message Broker
Unfiltered Data Access Service (Publish/Subscribe)

Figure 15. Basic request/response and publish/subscribe unfiltered data access services. These are simple and may be sufficient in cases where a client requires access to all the data of a particular type.

Though network bandwidth costs continue to drop, it is still impractical to broadcast all data to all consumers on the network. Aside from basic partitioning of data by sensor or product type, the need for access to a spatial subset of the data for a particular region is common. The addition of this capability to both the publish/subscribe and request/response services is shown in Figure 16.

In the FAA context, the need for a particular spatial subset is often quite static – a given TRACON, for example, may require a certain data product only within a certain radius of that TRACON. This is a situation that does not change, at least not frequently. The configuration of the spatial filtering component in the example reflects this, consisting of a static set of configuration files. In the ideal case, the spatial filtering software component is a reusable component that can be used in the publish/subscribe case and the request/response case. If that is not possible for technical reasons, the use of a common filtering language for both cases is certainly desirable.

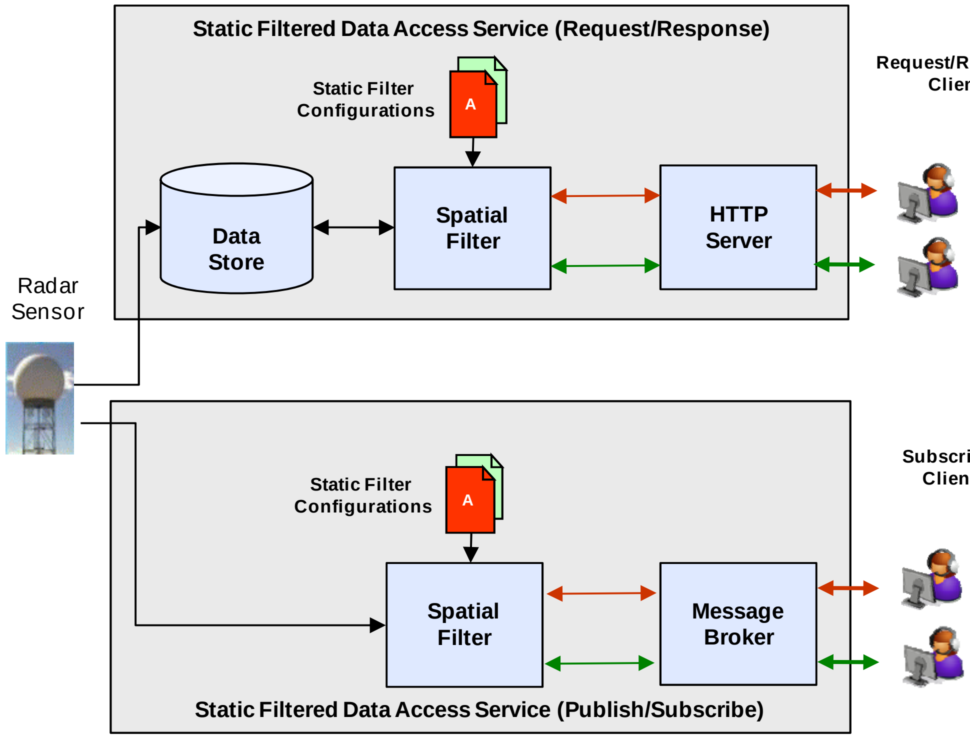
Static Filtered Data Access Service (Request/Response)
Static Filter Configurations
A
Request/Response Clients
Data Store
Spatial Filter
HTTP Server
Radar Sensor
Static Filter Configurations
A
Subscriber Clients
Spatial Filter
Message Broker
Static Filtered Data Access Service (Publish/Subscribe)

Figure 16. Incorporation of spatial filtering capability. In this configuration, the filtering choices are made by the data providers and statically configured. Information regarding the statically configured data 'channels' is communicated to clients via a Web site or registry. The spatial/temporal filtering component is common, though instances of the component are not shared.

Future NextGen applications are envisioned to require access to data in the recent past (trends) as well as in the future (weather and flight constraint forecasts). In addition to relatively recent data (on the scale of hours), access to historical data on a multi-day timescale is sometimes required for things such as accident investigation. Though these capabilities could continue to be supported on a system-by-system basis, the basic functionality is similar across multiple application domains, and some cost savings could be achieved by addressing it in a common way via SWIM.

Figure 17 illustrates the addition of a basic temporal filtering element to the previous example. Since publish/subscribe access is focused on access to real-time data, the basic temporal filter is not as relevant to that case, so the capability is only shown as being incorporated into the request/response service variant. Given the type of access envisioned – access to historical data on demand – static configuration of the filtering component is no longer appropriate and a service interface to the filter is provided. Following the composable theme of this example, the core filtering language used for the request/response service interface remains the same as for the previously, statically configured case. The necessary temporal extensions are incorporated, following the 'core + extensions' model described previously. The filtering capability of the request/response service therefore remains largely common with the publish/subscribe variant, and software reuse is still practical.

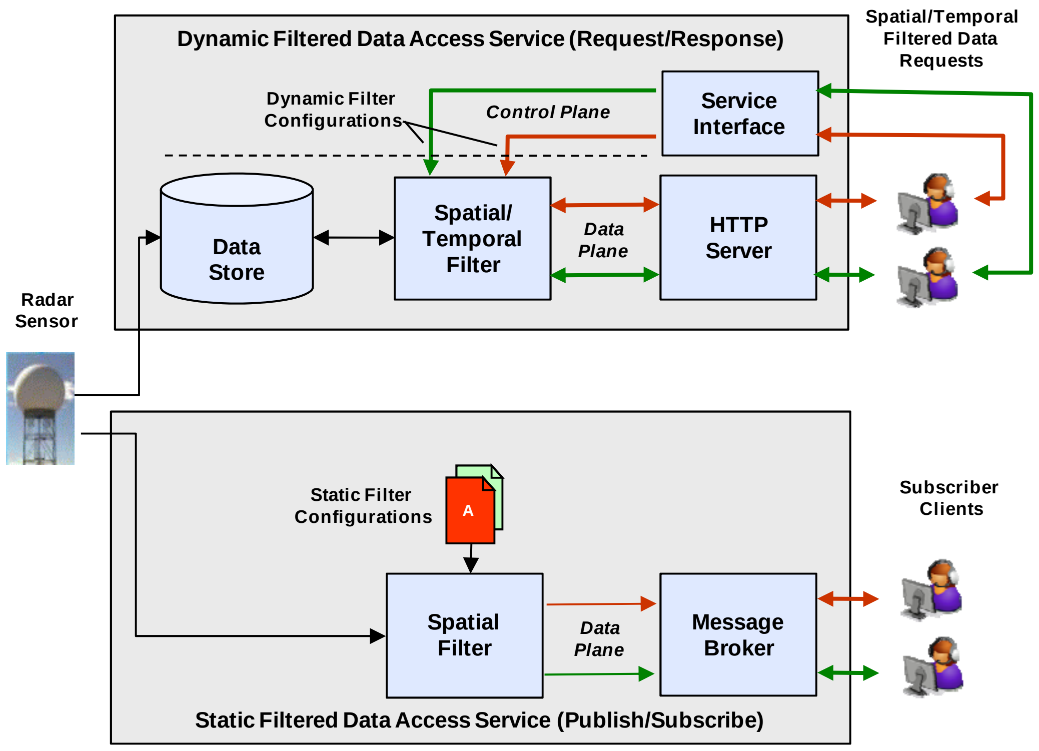
Dynamic Filtered Data Access Service (Request/Response)
Spatial/Temporal Filtered Data Requests
Dynamic Filter Configurations
Control Plane
Service Interface
Data Store
Spatial/ Temporal Filter
Data Plane
HTTP Server
Radar Sensor
Static Filter Configurations
A
Subscriber Clients
Spatial Filter
Data Plane
Message Broker
Static Filtered Data Access Service (Publish/Subscribe)

Figure 17. Incorporation of temporal filtering capability. In this configuration, the filtering choices are made by the data providers and statically configured. Information regarding the statically configured data 'channels' is communicated to clients via a Web site or registry. The spatial/temporal filtering component is common, though instances of the component are not shared.

The final step in the service evolution is to add more complex filtering capabilities, possibly including trajectory-based spatial queries and more sophisticated temporal queries. In this case, some of the more dynamic temporal queries *do* apply to the publish/subscribe model, and the static filter configuration is replaced with a service interface. Figure 18 depicts a single, composed, service interface that supports both request/response and publish/subscribe interface semantics. This type of interface composition is supported by the common SOA technologies such as WSDL and XML. In a sense, it can be thought of as a build-time service composition, rather than a run-time composition of multiple services to achieve some higher-level function. An alternate implementation would be to maintain separation of the request/response and publish/subscribe service instances, while still sharing some of the software components. We feel that a combined service has several advantages, including potential simplification of service management due to fewer separate endpoints. The OGC data access service interfaces, augmented with the OGC publish/subscribe extension (work in progress), follow this pattern.

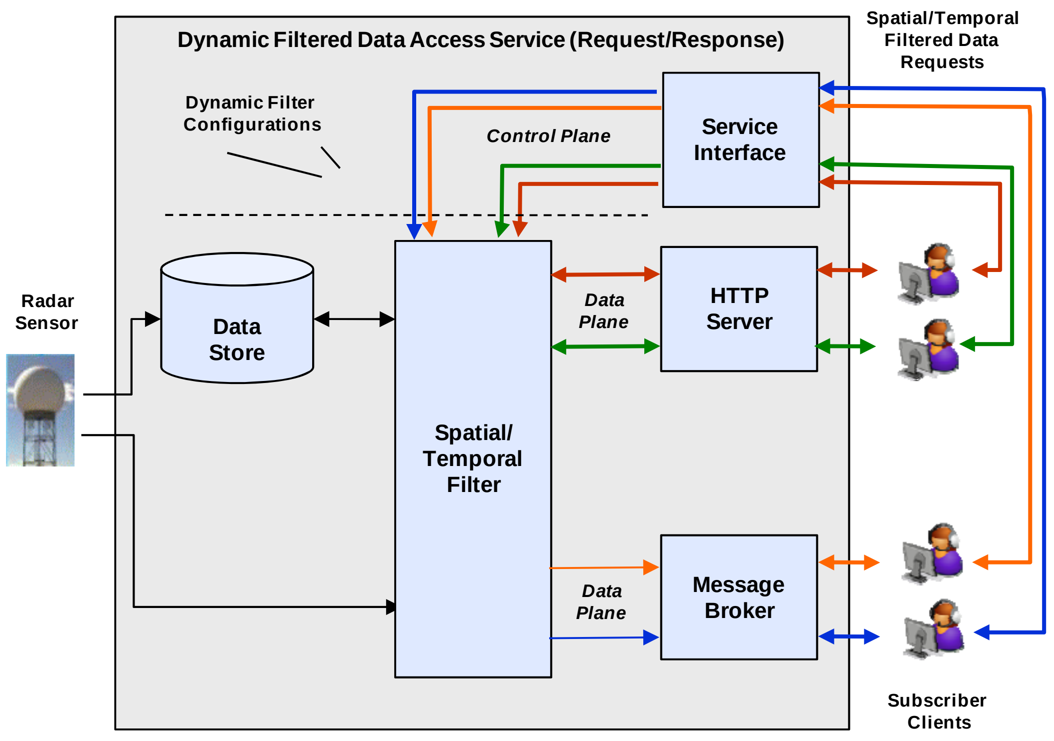

Figure 18. Integrated filtered data access service. In this configuration, the service interface supports both request/response and publish/subscribe interface semantics. The filtering component is shared. Examples of this style include instances of OGC or JMBL data access services that support a publish/subscribe extension. Particularly in the case of subscription-based access to real-time data flows, this can be viewed as a control plane/data plane architecture, where filtering and messaging components in the data plane are dynamically configured via a control plane (the query interface).

3.3 MODULAR SERVICE SUMMARY

The touted strength of SOA is the increased agility and software reuse that is achievable. With some architectural pre-planning, SOA concepts can be leveraged to allow for incremental development and deployment of shared SWIM data access services that support a spectrum of functionality, as described in this example. Following this model would allow for SWIM customers to select data access service features on an as-needed basis, without incurring undue complexity.

In order to achieve this end result, future prototyping efforts should explicitly include the capability to transform and extend a service's capabilities over time, in a similar fashion to that shown here. This is critical to proving the agility of the overall approach. To date, relatively little testing and verification of this SOA 'agility dimension' has been conducted.

This page intentionally left blank.

4. MISCELLANEOUS TOPICS

4.1 SOAP

4.1.1 Background

SOAP, originally an acronym for “Simple Object Access Protocol,” began life as a way of implementing network-scale remote procedure calls (RPCs) using XML. In some ways, it was the XML analog of CORBA, using a more flexible, albeit more verbose, data model. SOAP typically relied (and continues to rely) on lower level protocols, such as HTTP, for transport. The use of HTTP as a transport was driven to some extent by the desire to have the protocol be firewall-friendly, allowing remote procedure calls to be executed at WAN scales.

Over time, as the limitations of the RPC model at Internet scales became more recognized by the SOAP community, the usage of SOAP shifted from the RPC model to the ‘Document-oriented’ model, in which SOAP primarily functions as a general-purpose application-level transport protocol layer in much the same way as HTTP and JMS. Used in this manner, SOAP became a building block for a number of higher-level Web Service (WS-*) specifications, such as WS-Security and WS-Policy. Rather than encoding service operations, SOAP functioned as a carrier of security and policy information, in the SOAP header block. A typical SOAP header containing Security Assertion Markup Language (SAML) metadata is depicted below:

```
POST /SAMLResponder/resolve HTTP/1.1
Host: samlpeHost.com
Content-Type: text/xml
Content-Length: 1524

<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/soap/envelope/
  http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <ws:Security
      xmlns:ws="http://schemas.xmlsoap.org/ws/2002/04/secext"
      xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
      <saml:Assertion
        xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
        AssertionID="asb93/valkesdia=" Issuer="www.wx.ll.mit.edu"
        IssueInstant="2011-02-23T16:58:33.173Z">
        <saml:Conditions
          NotBefore="2011-02-23T16:53:33.173Z"
          NotOnOrAfter="2011-02-23T17:08:33.173Z"/>
        <saml:AuthenticationStatement
          AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:X.509"
          AuthenticationInstant="2011-02-23T16:57:30.000Z">
          <saml:Subject>
```

```
            <saml:NameIdentifier>Client</saml:NameIdentifier>
            <saml:SubjectConfirmation>
              <saml:ConfirmationMethod>
                urn:oasis:names:tc:SAML:1.0:cm:sender-vouches
              </saml:ConfirmationMethod>
            </saml:SubjectConfirmation>
          </saml:Subject>
        </saml:AuthenticationStatement>
        <ds:Signature>....(Omitted)...</ds:Signature>
      </saml:Assertion>
    </ws:Security>
  </soap:Header>
  <soap:Body>
    ...Message data (XML)...
  </soap:Body>
</soap:Envelope>
```

Though allowing for a richer metadata model (XML) than the simple "name, value" pairs most often used in HTTP and JMS headers, SOAP did introduce a problem. Historically, the majority of transport-level protocols are designed to treat the data being transported in an opaque fashion. This is true at virtually all levels of the Open Systems Interconnection (OSI) 7-layer model, and is key to a performant message transport stack. In the previous example, the HTTP protocol supports a Content-Length header with the length in bytes of the embedded SOAP content, allowing the content to be manipulated using simple memory copy operations. SOAP differs in that it enforces an XML model on the data being passed by the protocol. This difference between SOAP and the other layers in the OSI model is highlighted in Figure 19.

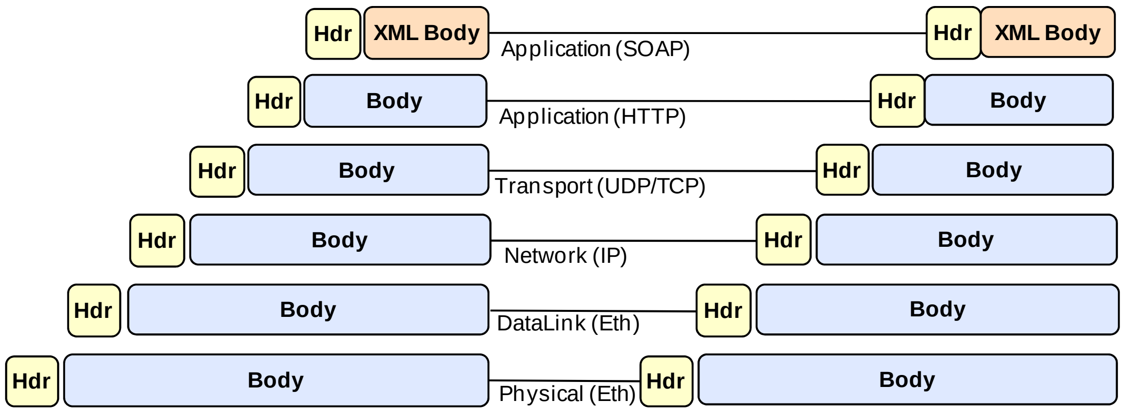

Figure 19. SOAP/HTTP, as represented in the OSI layered model.

The non-opaque nature of the message body in SOAP results in difficulties when working with SOAP and binary data. On the encoding side, binary data must be massaged into a form that is compatible with the XML model. On the decoding side, parsing of only the transport-related information in a SOAP header becomes awkward since many XML parsing tools are set up to parse the entire message, including the message body. This can result in significant overhead for SOAP-based routing logic if specialized parsers are not employed and messages are of a non-trivial size.

A number of approaches have been used over the years to address the issue of SOAP and binary data, including SOAP w/Attachments, Direct Internet Message Encapsulation (DIME), and the more recent SOAP Message Transmission Optimization Mechanism (MTOM). All these approaches are, in our opinion, 'repairs' for a poorly designed protocol, or perhaps more accurately, a protocol that ended up getting adopted for a different purpose than it was originally intended. Approaches like MTOM are functional, but awareness of the fact that if one uses plain HTTP without SOAP there is no need for MTOM is relevant to SWIM because it speaks to the long-term outlook for SOAP more generally. The relative complexity and XML-centric nature of SOAP has certainly contributed to the limited adoption of the technology in the Web community at large.

SOAP messaging *does* provide some capabilities related to security and policy that are not commonly supported in HTTP and JMS protocols, via its support for XML-based metadata such as SAML, eXtensible Access Control Markup Language (XACML), and WS-Policy in SOAP message headers. HTTP and JMS support a simpler metadata model, consisting of a set of name, value pairs. This difference often results in the perception that 'serious' enterprise services should use SOAP messaging on top of HTTP and/or JMS to handle security-related issues. On the other hand, from the software implementer's perspective there is a desire to keep things as simple as possible, and avoid the complexity of SOAP. These conflicting desires tend to fragment the developed solutions, leading to interoperability problems and security-related 'stovepipes.'

4.1.2 WS-Security Support

The value of SAML, XACML, and WS-Policy metadata is real and should be available to all three service interface styles. Rather than impose, for example, SOAP wrapping of messages flowing on JMS, one alternative worth considering is to include the information normally packaged in a SOAP header in custom HTTP or JMS headers. The HTTP and JMS specifications both support simple string headers in a flexible way, with no preset length limitation.

In terms of practical considerations, there are some implementation-related constraints to consider, such as the maximum header size of 8190 bytes supported by the Apache Web Server. Most browsers support much larger header fields, some limited only by system memory limits. At the low end is Firefox, which supports only a 10K byte header (still larger than the Apache server default maximum). Note that maximum header sizes are generally configurable to be larger if needed.

A non-SOAP variant of the example above using a bare-bones HTTP, REST-style service with a WS-Security header would look like:

```
POST /SAMLResponder/resolve HTTP/1.1
Host: samlpeHost.com
Content-Type: text/xml

WS-Security: <ws:Security xmlns:ws="http://schemas.xmlsoap.org/ws/2002/04/secext"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"><saml:Assertion
xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" AssertionID="asb93/valkesdia="
Issuer="www.wx.ll.mit.edu" IssueInstant="2011-02-23T16:58:33.173Z"><saml:Conditions
NotBefore="2011-02-23T16:53:33.173Z" NotOnOrAfter="2011-02-
23T17:08:33.173Z"/><saml:AuthenticationStatement
AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:X.509" AuthenticationInstant="2011-02-
23T16:57:30.000Z"><saml:Subject><saml:NameIdentifier>Client</saml:NameIdentifier><saml:SubjectConf
irmation><saml:ConfirmationMethod>urn:oasis:names:tc:SAML:1.0:cm:sender-
vouches</saml:ConfirmationMethod></saml:SubjectConfirmation></saml:Subject></saml:Authentication
Statement><ds:Signature>....(Omitted)...</ds:Signature></saml:Assertion></ws:Security>
Content-Length: 720

...Message data (any data type consistent with Content-Type header)...
```

A JMS-based variant of this strategy would look very similar. Assuming that header lengths are indeed not an issue, using this approach with REST and MOM style services results in more unified security capabilities across all services, while avoiding the overhead and complexity of SOAP and MTOM. A variant of the above approach would be to define a 'SOAP-Header' HTTP or JMS Header, and include the entire content of the SOAP header in that header. We recommend using individual HTTP/JMS headers for all top-level elements within the SOAP header, as illustrated above in the WS-Security case.

Modifications to software in XML gateways to support WS-Security for HTTP and JMS using this approach should be relatively trivial, since the content and formatting of the core WS-Security information remains the same. It is actually a simpler parsing job from a gateway device's perspective, since there is no need for a gateway to carefully parse only a portion of a SOAP Envelope – the SOAP header portion, in order to remain efficient.

We recognize that such an approach to WS-Security is likely to be controversial, especially given some of the tooling built up around SOAP. We are not advocating the elimination of SOAP as it is currently used, but rather recommending that its use not be extended into areas where its usage is not currently very common (e.g., JMS and REST-style services).

5. SUMMARY AND RECOMMENDATIONS

At a high level, SOA has always been about the construction of modular, composable, loosely-coupled systems, resulting in increased system agility and lower costs. Early SOA development efforts often focused on particular technologies, such as XML, SOAP, WSDL, UDDI, ESBs, and the WS-* family of standards – the technology framework within which services are developed and deployed. Less attention was paid to the architecture of the services themselves, resulting in poor interoperability as well as limited software reuse. In recent years, the focus has shifted to the service interfaces, and how the SOA approach can best blend with and complement existing architectural styles (e.g., REST) and implementations (e.g., the Web). This shift in thinking is recognized as desirable for SWIM Segment 2.

In this report, we focus on the topic of *shared information access services* and how lessons learned in the context of other information sharing architectures might be leveraged for SWIM Segment 2. Based on our assessment of existing approaches, we make the following interface-related recommendations:

- Maintain a clean separation of data access service interfaces and underlying data models wherever possible.

- At the lowest information access interface layer, abide by the principle of verb/noun separation to provide flexibility and minimize the size of the overall interface dictionary. Many interfaces can be efficiently expressed using a small number of verbs (e.g., create, read, update, delete), and Web, database, and operating system Application Programming Interfaces (APIs) that follow this principle have stood the test of time. A corollary to this point is that verb/noun combinations in base-level information access service operations should be considered an anti-pattern that can lead to the replacement of existing point-to-point communications with point-to-point Web Services (not scalable).

- Extend the base Create/Read/Update/Delete (CRUD) verb set to incorporate the *publish/subscribe/renew/unsubscribe* verbs, resulting in a total of 8 core verbs in SWIM data access services. This aligns the RPC and REST interface styles with MOM style interfaces, and enables construction of hybrid services that combine request/response access with publish/subscribe access. At a practical implementation level, a service interface that supports the basic CRUD pattern can be extended to support these additional verbs via composition with the WS-Notification interface specification. This approach was taken with the CIWS Data Distribution System (CDDS) service interfaces.

- Work to converge on a common set of filtering languages across all interface styles, RPC, REST, and MOM. The filtering languages should follow the *core + extension* principle, supporting simple *what, when, where* filtering in the core layer and adding more data type-specific (e.g., gridded vs. non-gridded data) queries in a number of layered extensions. This recommendation is based on lessons learned in the OpenSearch and OGC data access service (WFS, WCS) specifications. The OGC Filter specification used within the WFS data access service is one example of a filtering language that conforms to this principle.

- Architect data access services in a modular fashion, such that a simple, unfiltered publish/subscribe service may be incrementally built up to incorporate statically configured filtering, as well as more dynamically configured filtering (i.e., a Web Service-powered *control plane* for a publish/subscribe *data plane*). In order to achieve this goal, service interfaces, like their data model and filtering language counterparts, should abide by the *core + extensions* principle. For XML-based interfaces, the use of *open content models* as supported by ATOM and the recent XML Schema Version 1.1 specification should be considered as the means to provide the necessary extension points.

At the system level, the core information delivery aspect of SWIM is closely aligned with the goals of Content Delivery Networks (CDNs), such as Akamai, (Others). These networks, overlaid on the existing Internet, provide scalable data delivery and caching nearer the network edge to offload those concerns from data providers, in a manner that is largely transparent to data consumers. Though initially focusing on static Web page content, CDNs increasingly are used for more dynamic content, such as Internet-based video-on-demand services. Two key aspects of CDN solutions of interest to SWIM are the use of the origin server / distribution server concept to insulate data providers from problems of scale, and automatic redirection of origin server requests to the optimal (typically the closest) distribution server on a per-consumer basis, typically leveraging a customized DNS infrastructure. Recommendations based on CDNs include:

- Adopt a multi-tiered data access topology including the concept of origin servers and distribution servers. This is already the case to some extent with respect to the planned SWIM message broker topology.

- Consider the deployment of distribution server elements based on traditional Web technologies (e.g., HTTP caching proxies) in addition to a deployed set of publish/subscribe message brokers currently being considered. Given that a number of message brokers (e.g., Apache ActiveMQ) are not recommended for use with large messages, and ‘out-of-band’ delivery mechanisms are recommended, this may in fact be required to achieve efficient data ‘fan-out’ of large, common datasets (e.g., weather datasets) to multiple consumers at the network edge.

- Consider the use of DNS as a means to direct consumer requests to the closest distribution servers, as is commonly done in CDNs today. This may place certain requirements on a DNS implementation, over and above basic DNS, and is likely deserving of further study. An alternative redirection strategy at the low end would include simple static configuration files at each site. At the high end, a solution that accesses the necessary metadata from a runtime registry could be used.

Google Maps and Google Earth are two examples of data access services that build upon traditional Web protocols, providing access to subsets of 2 and 3-D imagery, respectively. The use of tiled imagery within Google Maps to reduce demands on the server side is relevant to SWIM, and suggests the use of a quantized set of available filtered regions to balance computational demands between the server side and the client side. In such an architecture, the client may receive more data than requested around the edge of a requested region, and be responsible for clipping the data to the exact spatial boundaries needed for a particular application. The benefit to the server side is the ability to pre-compute the quantized, filtered data sets that are typically needed to address the needs of multiple applications, reducing the overall computational workload needed to support large numbers of clients. Though most commonly used in conjunction with gridded data sets, this approach applies to non-gridded data access as well.

Shared information access services that can be leveraged across multiple domains hold promise with respect to overall NextGen system agility, reduced software development and maintenance costs, and reduced communication costs. Convergence on a set of common service interfaces that meet the needs of a variety of stakeholders, is, as demonstrated in SWIM Segment 1, a challenging proposition, one that is not likely to be achieved by SWIM implementing programs acting independently. In order to meet the goals of NextGen, SWIM Segment 2 should continue to place an emphasis on shared information access services, addressing not only the security and monitoring aspects of services, but the core data access functionality discussed in this report as well.

This page intentionally left blank.

GLOSSARY

AIM	Aeronautical Information Management
Amazon S3	Amazon Simple Storage Service
AMQP	Advanced Message Queueing Protocol
API	Application Programming Interface
APP	Atom Publishing Protocol
CDDS	CIWS Data Distribution System
CDN	Content Delivery Network
CIWS	Corridor Integrated Weather System
CNAME	Canonical Name
CORBA	Common Object Request Broker Architecture
CRUD	Create, Read, Update, Delete
DCOM	Distributed Component Object Model
DDS	Data Distribution Service
DIME	Direct Internet Message Encapsulation
DNS	Domain Name System
DoD	Department of Defense
DWDM	Dense Wavelength Division Multiplexing
ERAM	En Route Automation Modernization
ESB	Enterprise Service Bus
Eurocontrol	European Organization for the Safety of Air Navigation
FAA	Federal Aviation Administration
FDP	Flight Data Processor
FTI	FAA Telecommunications Infrastructure
FTP	File Transfer Protocol
GUID	Globally Unique Identifier
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
ITWS	Integrated Terminal Weather System
JMBL	Joint METOC Broker Language
JMS	Java Messaging Service
KML	Keyhole Markup Language
METOC	Meteorological and Oceanographic
MOM	Message-Oriented Middleware
MTOM	Message Transmission Optimization Mechanism
NAS	National Airspace System
NCAR	National Center for Atmospheric Research
NEXRAD	Next-Generation Radar

NextGen	Next Generation Air Transportation System
NNEW	NextGen Network-Enabled Weather
OGC	Open Geospatial Consortium
OSI	Open Systems Interconnection
POP	Point of Presence
REST	Representational State Transfer
ROA	Resource-Oriented Architecture
RPC	Remote Procedure Call
RSS	Really Simple Syndication
SAML	Security Assertion Markup Language
SIP	SWIM Implementing Program
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SRV	Service Record
SWIM	System Wide Information Management
TBO	Trajectory-Based Operations
TCP	Transmission Control Protocol
TRACON	Terminal Radar Approach Control
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
URL	Uniform Resource Locator
VLAN	Virtual Local Area Network
WAN	Wide Area Network
WCS	Web Coverage Service
WFS	Web Feature Service
WOA	Web-Oriented Architecture
WS	Web Services
WSDD	Web Service Description Document
WSDL	Web Services Description Language
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

REFERENCES

1. Jalleta, E., B. Kaul, and D. Thomson, *"SWIM Core Architecture Evolution Concepts"* Version 1.0, MITRE Technical Report MTR090193, July 2009.

2. Information Technology Association of America, *"FAA Swim Program: Segment 1 to Segment 2 Transition – Industry Input,"* December 2008.

3. Fielding, Roy Thomas, *"Architectural Styles and the Design of Network-based Software Architectures,"* Doctoral dissertation, University of California, Irvine, 2000.

4. Manes, A.T., and R.T. Fielding, *"Draft findings on unsafe methods,"* W3C Technical Architecture Group, April 2002. http://lists.w3.org/Archives/Public/www-tag/2002Apr/0235.html

5. Waters, J., *"A SOA Post Mortem with Anne Thomas Manes,"* Application Development Trends, March 2009. http://adtmag.com/articles/2009/03/03/a-soa-post-mortem-with-anne-thomas-manes.aspx

6. Gall, N., *"Putting the Web back into Web Services,"* Gartner Blog Post, [Online] November 19, 2008. http://blogs.gartner.com/nick_gall/2008/11/19/woa-putting-the-web-back-in-web-services/

7. Tilkov, S., *"Pragmatic SOA – Beyond Buzzwords and Flamewars,"* QCon, London, [Online] August 2010. http://www.infoq.com/presentations/Pragmatic-SOA

8. Erl, T., *"SOA – Principles of Service Design,"* Prentice Hall, 2008, p. 44.

9. Lockheed Martin, *"Flight Information Service – Web Service Design Document (Draft),"* July 2010.

10. Lockheed Martin, *"Flight Information Service Supporting the Departure Pre-Route Amendment – Web Service Design Document (Draft),"* October, 2009.

11. Codehaus.org, Stomp Protocol Home Page, [Online] http://stomp.codehaus.org

12. Red Hat, Inc., Advanced Message Queuing Protocol Specification, [Online] http://www.redhat.com/solutions/specifications/amqp

13. OpenSearch Organization, OpenSearch Home Page, [Online] http://www.opensearch.org/Home

14. Saltzer, J.H., D.P. Reed, and D.D. Clark, *"End-to-End Arguments in System Design,"* ACM Transactions in Computer Systems, 4 November 1984, Vol. 2, pp. 277–288.

This page intentionally left blank.