Technology and Recent Development of XML Web Services

Hazem M. El-Bakry, Alaa M. Riad, Qusay F. Hassan, Ahmed E. Hassan, Nikos Mastorakis

Faculty of Computer Science & Information Systems, Mansoura University, EGYPT
E-mail: helbakry20@yahoo.com

Faculty of Engineering, Mansoura University, EGYPT

Technical University of Sofia, BULGARIA

Abstract:

Service oriented architecture (SOA) is a flexible set of design principles used during the phases of systems development and integration. Upon deployment, an SOA-based architecture will provide a loosely-integrated suite of services that can be used within multiple business domains. SOA also generally provides a way for consumers of services, such as web-based applications, to be aware of available SOA-based services. For example, several disparate departments within a company may develop and deploy SOA services in different implementation languages, and their respective clients use a well understood, well defined interface to access them. XML is commonly used for interfacing with SOA services, though this is not required. It has been becoming one of the most widely used methodologies for building and integrating different types of software applications. This because the extreme benefits that it offers to their adopters including agility, dynamicity, and loose-coupling. These benefits are usually missed in traditional software terminologies and practices. XML Web Services is the most used technology for realizing SOA because it is easy to use. Furthermore, it allows high interoperability between different systems due to its dependency on standards that are widely accepted and supported by almost all large software vendors. However, XML Web Services suffers from a number of drawbacks such as low performance, bad utilization of hardware resources, and high network latency. These pitfalls may prevent some adopters from utilizing SOA in large and complex systems. Therefore, these issues should be first addressed and resolved before leveraging it into real-time systems. In this paper, an experimental evaluation for the performance of XML Web Services in real-time business systems is presented. In addition, this study offers some tactics and strategies that might be used to enhance the overall performance of XML Web Services. Furthermore, web service development as well as design analysis, modeling, and methodologies are presented. Moreover, the management and security of SOA are discussed.

Keywords: SOA, XML Web Services, Problem Root Causes, Design Analysis and Methodologies, Modeling, Governance, Security, Systems Integration, Performance Evaluation, Optimization Tactics and Strategies.

I. Introduction

A web service (also webservice) is traditionally defined by the W3C as "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards." Web services today are frequently just Application Programming Interfaces (API) or web APIs that can be accessed over a network, such as the Internet, and executed on a remote system hosting the requested services [35].

SOA allows different ways to develop applications by combining services. The main premise of SOA is to erase application boundaries and technology differences. As applications are opened up, how we can combine these services securely becomes an issue. Traditionally, security models have been hardcoded into applications and when capabilities of an application are opened up for use by other applications, the security models built into each application may not be good enough. Over the last few years, SOA has gained momentum from almost all players in software market for building and integrating systems, especially complex ones that demand continuous changes to meet market ever-changing requirements [1,2,10].

Several emerging technologies and standards address different aspects of the problem of security in SOA. Standards such as WS-Security, SAML, WS-Trust, WS-Secure Conversation and WS-Security Policy focus on the security and identity management aspects of SOA implementations that use Web Services. Technologies such as the Virtual Organization in Grid Computing, Application-oriented networking (AON) and XML Gateways are addressing the problem of SOA security in the larger context as well [47,48].
XML Gateways are hardware or software based solutions for enforcing identity and security for SOAP, XML, and REST based web services, usually at the network perimeter. An XML gateway is a dedicated application which allows for a more centralized approach to security and identity enforcement, similar to how a protocol firewall is deployed at the perimeter of a network for centralized access control at the connection and port level.

XML Gateway SOA Security features include PKI, Digital Signature, Encryption, XML Schema Validation, Antivirus, and Pattern Recognition. Regulatory certification for XML gateway security features are provided by FIPS and DoD [82-92].

Technically, many technologies could be used for realizing and implementing service-oriented systems including message queuing, remote procedure calls (RPCs), Common Object Request Broker (CORBA), and Common Object Model (COM) [4]. However, XML Web Services is the most used technology for realizing SOA due to a number of factors including [5]:

- **Ease-of-Use**: Using XML Web Services does not require deep technical knowledge, as it is very easy to learn and use, especially if compared with other tools like CORBA and COM.
- **Support**: Because it has appeared as a result of cooperation between large software vendors such as Microsoft, IBM, BEA, and Sun Microsystems, it is supported in almost all software tools, frameworks, and programming languages.
- **Modularity**: It is modular by nature, so, it is easy to encapsulate logic in terms of modules that could be deployed and used separately according to business needs.
- **Compose-ability**: It is very easy to aggregate a number of XML Web Services to construct a new one that covers more complex needs.
- **Low Costs**: It is much cheaper than traditional and proprietary technologies.
- **Commonality with SOA Model**: The architectural model of XML Web Services is almost identical to the basic model of SOA, as they both have service provider, service consumer, service registry, service contract, and service itself.

Unfortunately, the aforementioned advantages of XML Web Services were not enough to enable SOA adopters to use it effectively in real-time business systems, as it still suffers from a number of problems such as low performance, bad utilization for hardware resources, and high network latency.

These issues must be first addressed and resolved to allow optimum utilization for SOA.

SOA implementations rely on a mesh of software services. Services comprise unassociated, loosely coupled units of functionality that have no calls to each other embedded in them. Each service implements one action, such as filling out an online application for an account, or viewing an online bank-statement, or placing an online booking or airline ticket order. Instead of services embedding calls to each other in their source code they use defined protocols that describe how services pass and parse messages, using description metadata [82-92].

SOA developers associate individual SOA objects by using orchestration. In the process of orchestration the developer associates software functionality (the services) in a non-hierarchical arrangement (in contrast to a class hierarchy) using a software tool that contains a complete list of all available services, their characteristics, and the means to build an application utilizing these sources.

Underlying and enabling all of this requires metadata in sufficient detail to describe not only the characteristics of these services, but also the data that drives them. Programmers have made extensive use of XML in SOA to structure data that they wrap in a nearly exhaustive description-container. Analogously, the Web Services Description Language (WSDL) typically describes the services themselves, while the SOAP protocol describes the communications protocols. Whether these description languages are the best possible for the job, and whether they will become/remain the favorites in the future, remain open questions. As of 2008 SOA depends on data and services that are described by metadata that should meet the following two criteria [79-81]:

1. The metadata should come in a form that software systems can use to configure dynamically by discovery and incorporation of defined services, and also to maintain coherence and integrity.
2. The metadata should come in a form that system designers can understand and manage with a reasonable expenditure of cost and effort.

SOA aims to allow users to string together fairly large chunks of functionality to form ad hoc applications that are built almost entirely from existing software services. The larger the chunks, the fewer the interface points required to implement any given set of functionality; however,
very large chunks of functionality may not prove sufficiently granular for easy reuse. Each interface brings with it some amount of processing overhead, so there is a performance consideration in choosing the granularity of services. The great promise of SOA suggests that the marginal cost of creating the n-th application is low, as all of the software required already exists to satisfy the requirements of other applications. Ideally, one requires only orchestration to produce a new application.

For this to operate, no interactions must exist between the chunks specified or within the chunks themselves. Instead, humans specify the interaction of services (all of them unassociated peers) in a relatively *ad hoc* way with the intent driven by newly emergent requirements. Thus the need for services as much larger units of functionality than traditional functions or classes, lest the sheer complexity of thousands of such granular objects overwhelm the application designer. Programmers develop the services themselves using traditional languages like Java, C, C++, C# or COBOL [34].

SOA services feature loose coupling, in contrast to the functions that a linker binds together to form an executable, to a dynamically linked library or to an assembly. SOA services also run in "safe" wrappers (such as Java or .NET) and in other programming languages that manage memory allocation and reclamation, allow *ad hoc* and late binding, and provide some degree of indeterminate data typing [47].

As of 2008, increasing numbers of third-party software companies offer software services for a fee. In the future, SOA systems may consist of such third-party services combined with others created in-house. This has the potential to spread costs over many customers and customer uses, and promotes standardization both in and across industries. In particular, the travel industry now has a well-defined and documented set of both services and data, sufficient to allow any reasonably competent software engineer to create travel-agency software using entirely off-the-shelf software services. Other industries, such as the finance industry, have also started making significant progress in this direction [34-49].

SOA as an architecture relies on service-orientation as its fundamental design-principle [40-51]. If a service presents a simple interface that abstracts away its underlying complexity, users can access independent services without knowledge of the service's platform implementation [41-48].

SOA relies on services exposing their functionality via interfaces that other applications and services can read to understand how to utilize those services.

**Requirements**

In order to efficiently use a SOA, one must meet the following requirements:

- Interoperability between different systems and programming languages that provides the basis for integration between applications on different platforms through a communication protocol. One example of such communication depends on the concept of messages. Using messages across defined message channels decreases the complexity of the end application, thereby allowing the developer of the application to focus on true application functionality instead of the intricate needs of a communication protocol.
- Desire to create a federation of resources. Establish and maintain data flow to a Federated database system. This allows new functionality developed to reference a common business format for each data element.

**Principles**

The following guiding principles define the ground rules for development, maintenance, and usage of the SOA [36-65]:

- reuse, granularity, modularity, composability, componentization and interoperability.
- standards-compliance (both common and industry-specific).
- services identification and categorization, provisioning and delivery, and monitoring and tracking.

The following specific architectural principles for design and service definition focus on specific themes that influence the intrinsic behaviour of a system and the style of its design [50]:

- **Service encapsulation** – Many web services are consolidated for use under the SOA. Often such services were not planned to be under SOA.
- **Service loose coupling** – Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.
• **Service contract** – Services adhere to a communications agreement, as defined collectively by one or more service-description documents.

• **Service abstraction** – Beyond descriptions in the service contract, services hide logic from the outside world.

• **Service reusability** – Logic is divided into services with the intention of promoting reuse.

• **Service composability** – Collections of services can be coordinated and assembled to form composite services.

• **Service autonomy** – Services have control over the logic they encapsulate.

• **Service optimization** – All else equal, high-quality services are generally preferable to low-quality ones.

• **Service discoverability** – Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.

• **Service relevance** – Functionality is presented at a granularity recognized by the user as a meaningful service.

The following references provide additional considerations for defining a SOA implementation:

- SOA Reference Architecture provides a working design of an enterprise-wide SOA implementation with detailed architecture diagrams, component descriptions, detailed requirements, design patterns, opinions about standards, patterns on regulation compliance, standards templates etc.

- Life cycle management SOA Practitioners Guide Part 3: Introduction to Services Lifecycle introduces the services lifecycle and provides a detailed process for services management through the service lifecycle, from inception to retirement or repurposing of the services. It also contains an appendix that includes organization and governance best-practices, templates, comments on key SOA standards, and recommended links for more information.

In addition, one might take the following factors into account when defining a SOA implementation:

- efficient use of system resources

- service maturity and performance

- EAI (Enterprise Application Integration)

Web services can implement a service-oriented architecture. Web services make functional building-blocks accessible over standard Internet protocols independent of platforms and programming languages. These services can represent either new applications or just wrappers around existing legacy systems to make them network-enabled.

Each SOA building block can play one or both of two roles [79-81]:

1. **Service Provider**

The service provider creates a web service and possibly publishes its interface and access information to the service registry. Each provider must decide which services to expose, how to make trade-offs between security and easy availability, how to price the services, or (if no charges apply) how/whether to exploit them for other value. The provider also has to decide what category the service should be listed in for a given broker service and what sort of trading partner agreements are required to use the service. It registers what services are available within it, and lists all the potential service recipients. The implementer of the broker then decides the scope of the broker. Public brokers are available through the Internet, while private brokers are only accessible to a limited audience, for example, users of a company intranet. Furthermore, the amount of the offered information has to be decided. Some brokers specialize in many listings. Others offer high levels of trust in the listed services. Some cover a broad landscape of services and others focus within an industry. Some brokers catalog other brokers. Depending on the business model, brokers can attempt to maximize look-up requests, number of listings or accuracy of the listings. The Universal Description Discovery and Integration (UDDI) specification defines a way to publish and discover information about Web services. Other service broker technologies include (for example) ebXML (Electronic Business using eXtensible Markup Language) and those based on the ISO/IEC 11179 Metadata Registry (MDR) standard.

2. **Service consumer**

The service consumer or web service client locates entries in the broker registry using various find operations and then binds to the service provider in order to invoke one of its web services. Whichever service the service-consumers need, they have to take it into the brokers, then bind it with respective service and then use it. They can access multiple services if the service provides multiple services.
Implementors commonly build SOAs using web services standards (for example, SOAP) that have gained broad industry acceptance. These standards (also referred to as Web Service specifications) also provide greater interoperability and some protection from lock-in to proprietary vendor software. One can, however, implement SOA using any service-based technology, such as Jini, CORBA or REST.

Other SOA concepts
Architectures can operate independently of specific technologies. Designers can implement SOA using a wide range of technologies, including:

- SOAP, RPC
- REST
- DCOM
- CORBA
- Web Services
- WCF (Microsoft's implementation of WebService forms a part of WCF)

Implementations can use one or more of these protocols and, for example, might use a file-system mechanism to communicate data conforming to a defined interface-specification between processes conforming to the SOA concept. The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, without a service having foreknowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks. Many implementers of SOA have begun to adopt an evolution of SOA concepts into a more advanced architecture called SOA 2.0. Elements of SOA, by Dirk Krafiżig, Karl Banke, and Dirk Slama. SOA meta-model, The Linthicum Group, 2007

SOA enables the development of applications that are built by combining loosely coupled and interoperable services [34-39].

These services inter-operate based on a formal definition (or contract, e.g., WSDL) that is independent of the underlying platform and programming language. The interface definition hides the implementation of the language-specific service. SOA-based systems can therefore function independently of development technologies and platforms (such as Java, .NET, etc). Services written in C# running on .NET platforms and services written in Java running on Java EE platforms, for example, can both be consumed by a common composite application (or client). Applications running on either platform can also consume services running on the other as web services that facilitate reuse. Managed environments can also wrap COBOL legacy systems and present them as software services. This has extended the useful life of many core legacy systems indefinitely, no matter what language they originally used.

SOA can support integration and consolidation activities within complex enterprise systems, but SOA does not specify or provide a methodology or framework for documenting capabilities or services.

High-level languages such as BPEL and specifications such as WS-CDL and WS-Coordination extend the service concept by providing a method of defining and supporting orchestration of fine-grained services into more coarse-grained business services, which architects can in turn incorporate into workflows and business processes implemented in composite applications or portals.

As of 2008 researchers have started investigating the use of Service Component Architecture (SCA) to implement SOA.

Some enterprise architects believe that SOA can help businesses respond more quickly and cost-effectively to changing market-conditions. This style of architecture promotes reuse at the macro (service) level rather than micro (classes) level. It can also simplify interconnection to – and usage of – existing IT (legacy) assets.

In some respects, one can regard SOA as an architectural evolution rather than as a revolution. It captures many of the best practices of previous software architectures. In communications systems, for example, little development has taken place of solutions that use truly static bindings to talk to other equipment in the network. By formally embracing a SOA approach, such systems can position themselves to stress the importance of well-defined, highly inter-operable interfaces [31-46].

Some have questioned whether SOA simply revives concepts like modular programming (1970s), event-oriented design (1980s) or interface/component-based design (1990s). SOA promotes the goal of separating users (consumers) from the service implementations. Services can therefore be run on various distributed platforms and be accessed across networks. This can also maximize reuse of services.

SOA is an architectural and design discipline conceived to achieve the goals of increased interoperability (information exchange, reusability, and composability), increased federation (uniting
resources and applications while maintaining their individual autonomy and self-governance), and increased business and technology domain alignment.

Service-Oriented Architecture (SOA) is an architectural approach (or style) for constructing complex software-intensive systems from a set of universally interconnected and interdependent building blocks, called services.

SOA realizes its business and IT benefits through utilizing an analysis and design methodology when creating services. This methodology ensures that services remain consistent with the architectural vision and roadmap, and that they adhere to principles of service-orientation. Arguments supporting the business and management aspects from SOA are outlined in various publications [40-43].

A service comprises a stand-alone unit of functionality available only via a formally defined interface. Services can be some kind of "nano-enterprises" that are easy to produce and improve. Also services can be "mega-corporations" constructed as the coordinated work of subordinate services.

Services generally adhere to the following principles of service-orientation [79-81]:

- abstraction
- autonomy
- composability
- discoverability
- formal contract
- loose coupling
- reusability
- statelessness

A mature rollout of SOA effectively defines the API of an organization.

Reasons for treating the implementation of services as separate projects from larger projects include [79-81]:

1. Separation promotes the concept to the business that services can be delivered quickly and independently from the larger and slower-moving projects common in the organization. The business starts understanding systems and simplified user interfaces calling on services. This advocates agility.
2. Separation promotes the decoupling of services from consuming projects. This encourages good design insofar as the service is designed without knowing who its consumers are.
3. Documentation and test artifacts of the service are not embedded within the detail of the larger project. This is important when the service needs to be reused later.

An indirect benefit of SOA involves dramatically simplified testing. Services are autonomous, stateless, with fully documented interfaces, and separate from the cross-cutting concerns of the implementation. The industry has never been exposed to this circumstance before.

If an organization possesses appropriate defined test data, then when a service is being built, a corresponding stub is built that reacts to the test data. A full set of regression tests, scripts, data, and responses is also captured for the service. The service can be tested as a 'black box' using existing stubs corresponding to the services it calls. Test environments can be constructed where the primitive and out-of-scope services are stubs, while the remainder of the mesh is test deployments of full services. As each interface is fully documented, with its own full set of regression test documentation, it becomes simple to identify problems in test services. Testing evolves to merely validating that the test service operates according to its documentation, and in finding gaps in documentation and test cases of all services within the environment. Managing the data state of idempotent services is the only complexity.

Examples may prove useful to aid in documenting a service to the level where it becomes useful. The documentation of some APIs within the Java Community Process provide good examples. As these are exhaustive, staff would typically use only important subsets. The 'ossjsa.pdf' file within JSR-89 exemplifies such a file [79-81].

Challenges in adopting SOA

One obvious and common challenge faced involves managing services metadata. SOA-based environments can include many services that exchange messages to perform tasks. Depending on the design, a single application may generate millions of messages. Managing and providing information on how services interact can become complex. This becomes even more complicated when these services are delivered by different organizations within the company or even different companies (partners, suppliers, etc). This creates huge trust issues across teams, and hence SOA Governance comes into picture.
Another challenge involves the lack of testing in SOA space. There are no sophisticated tools that provide testability of all headless services (including message and database services along with web services) in a typical architecture. Lack of horizontal trust requires that both producers and consumers test services on a continuous basis. SOA’s main goal is to deliver Agility to Businesses. Therefore it is important to invest in a testing framework (build or buy) that would provide the visibility required to find the culprit in the architecture. Business agility requires SOA services to be controlled by the business goals and directives as defined in the Business Motivation Model (BMM).

Another challenge relates to providing appropriate levels of security. Security models built into an application may no longer suffice when an application exposes its capabilities as services that can be used by other applications. That is, application-managed security is not the right model for securing services. A number of new technologies and standards have started to emerge to provide more appropriate models for security in SOA.

As SOA and the WS-* specifications practitioners expand, update and refine their output, they encounter a shortage of skilled people to work on SOA-based systems, including the integration of services and construction of services infrastructure.

Interoperability becomes an important aspect of SOA implementations. The WS-I organization has developed Basic Profile (BP) and Basic Security Profile (BSP) to enforce compatibility [46-92]. WS-I has designed testing tools to help assess whether web services conform to WS-I profile guidelines. Additionally, another charter has been established to work on the Reliable Secure Profile.

Significant vendor hype surrounds SOA; this can create expectations that may not be fulfilled. Product stacks continue to evolve as early adopters test the development and runtime products with real-world problems. SOA does not guarantee reduced IT costs, improved systems agility or faster time-to-market. Successful SOA implementations may realize some or all of these benefits depending on the quality and relevance of the system architecture and design [62-71].

Internal IT delivery organizations routinely initiate SOA efforts, and some of these improperly introduce concepts to the business so it remains misunderstood. The adoption starts meeting IT delivery needs instead of those of the business, resulting in an organization with superlative laptop provisioning services, instead of one that can quickly respond to market opportunities. Business leadership also becomes convinced that the organization is executing well on SOA.

One of the most important benefits of SOA is its ease of reuse. Therefore accountability and funding models must ultimately evolve within the organization. A business unit needs to be encouraged to create services that other units will use. Conversely, units must be encouraged to reuse services. This requires a few new governance components:

- Each business unit creating services must have an appropriate support structure in place to deliver on its service-level obligations, and to support enhancing existing services strictly for the benefit of others. This is typically quite foreign to business leaders.
- Each business unit consuming services accepts the apparent risk of reusing services outside their own control, with the attendant external project dependencies, etc.
- An innovative funding model is needed as incentive to drive these behaviors above Business units normally pay the IT Organization to assist during projects, and then to operate the environment. Corporate incentives should discount these costs to service providers, and create internal revenue-streams from consuming business units to the service provider. These streams should be less than the costs of a consumer simply building it the old-fashioned way. This is where SOA deployments can benefit from the SaaS monetization architecture [35-70]

**Criticisms of SOA**

Some criticisms of SOA depend on conflating SOA with web Services. For example, some critics claim SOA results in the addition of XML layers, introducing XML parsing and composition. In the absence of native or binary forms of Remote Procedure Call (RPC), applications could run slower and require more processing power, increasing costs. Most implementations do incur these overheads, but SOA can be implemented using technologies (for example, Java Business Integration (JBI)) that do not depend on remote procedure calls or translation through XML. At the same time, emerging open-source XML parsing technologies (such as VTD-XML) and various XML-compatible binary formats promise to significantly improve SOA performance [35-76].
Stateful services require both the consumer and the provider to share the same consumer-specific context, which is either included in or referenced by messages exchanged between the provider and the consumer. This constraint has the drawback that it could reduce the overall scalability of the service provider if the service-provider needs to retain the shared context for each consumer. It also increases the coupling between a service provider and a consumer and makes switching service providers more difficult. Ultimately, some critics feel that SOA services are still too constrained by applications they represent [32-70].

Another concern relates to the ongoing evolution of WS-* standards and products (e.g., transaction, security), and SOA can thus introduce new risks unless properly managed and estimated with additional budget and contingency for additional proof-of-concept work.

Some critics regard SOA as merely an obvious evolution of currently well-deployed architectures (open interfaces, etc).

IT system designs sometimes overlook the desirability of modifying systems readily. Many systems, including SOA-based systems, hard-code the operations, goods and services of the organization, thus restricting their online service and business agility in the global marketplace.

The next step in the design process covers the definition of a Service Delivery Platform (SDP) and its implementation. In the SDP design phase one defines the business information models, identity management, products, content, devices, and the end-user service characteristics, as well as how agile the system is so that it can deal with the evolution of the business and its customers.

The work presented here also investigates the performance of XML Web Services development in building and integrating real-time business systems taking a hypothetical scenario for banking solutions as an example to these systems in order to compare found results with those of using other traditional methods. Additionally, it discusses root causes of found problems in order to give some tactics and strategies that could be leveraged to make better use of XML Web Services.

II. Web Service Development

In common usage the term refers to clients and servers that communicate over the Hypertext Transfer Protocol (HTTP) protocol used on the web. Such services tend to fall into one of two camps: Big Web Services and RESTful Web Services.

"Big Web Services" use Extensible Markup Language (XML) messages that follow the Simple Object Access Protocol (SOAP) standard and have been popular with traditional enterprise. In such systems, there is often a machine-readable description of the operations offered by the service written in the Web Services Description Language (WSDL). The latter is not a requirement of a SOAP endpoint, but it is a prerequisite for automated client-side code generation in many Java and .NET SOAP frameworks (frameworks such as Spring, Apache Axis2 and Apache CXF being notable exceptions). Some industry organizations, such as the WS-I, mandate both SOAP and WSDL in their definition of a web service.

"Big Web Services" use Extensible Markup Language (XML) messages that follow the Simple Object Access Protocol (SOAP) standard and have been popular with traditional enterprise. In such systems, there is often a machine-readable description of the operations offered by the service written in the Web Services Description Language (WSDL). The latter is not a requirement of a SOAP endpoint, but it is a prerequisite for automated client-side code generation in many Java and .NET SOAP frameworks (frameworks such as Spring, Apache Axis2 and Apache CXF being notable exceptions). Some industry organizations, such as the WS-I, mandate both SOAP and WSDL in their definition of a web service.

"Big Web Services" use Extensible Markup Language (XML) messages that follow the Simple Object Access Protocol (SOAP) standard and have been popular with traditional enterprise. In such systems, there is often a machine-readable description of the operations offered by the service written in the Web Services Description Language (WSDL). The latter is not a requirement of a SOAP endpoint, but it is a prerequisite for automated client-side code generation in many Java and .NET SOAP frameworks (frameworks such as Spring, Apache Axis2 and Apache CXF being notable exceptions). Some industry organizations, such as the WS-I, mandate both SOAP and WSDL in their definition of a web service.

"Big Web Services" use Extensible Markup Language (XML) messages that follow the Simple Object Access Protocol (SOAP) standard and have been popular with traditional enterprise. In such systems, there is often a machine-readable description of the operations offered by the service written in the Web Services Description Language (WSDL). The latter is not a requirement of a SOAP endpoint, but it is a prerequisite for automated client-side code generation in many Java and .NET SOAP frameworks (frameworks such as Spring, Apache Axis2 and Apache CXF being notable exceptions). Some industry organizations, such as the WS-I, mandate both SOAP and WSDL in their definition of a web service.

More recently, Representational State Transfer (RESTful) web services have been regaining popularity, particularly with Internet companies. By using the PUT, GET and DELETE HTTP methods, alongside POST, these are often better integrated with HTTP and web browsers than SOAP-based services. They do not require XML messages or WSDL service-API definitions.

Web API is a development in web services (in a movement called Web 2.0) where emphasis has been moving away from Simple Object Access Protocol (SOAP) based services towards more direct Representational State Transfer (REST) style communications [61-65]. Web APIs allow the combination of multiple web services into new applications known as mashups [34-53].
used in the context of web development, web API is typically a defined set of Hypertext Transfer Protocol (HTTP) request messages along with a definition of the structure of response messages, usually expressed in an Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format.

A highly dynamic and loosely coupled environment increases not only the probability of deviation situations that occur during the execution of composite services, but also the complexity in exception handling [35-83]. Due to the distributed nature of SOA, loosely coupled feature of web services, the monitoring and exception handling issues about web services in SOA context is still an open research issue.

When running composite web services, each sub service can be considered autonomous. The user has no control over these services. Also the web services themselves are not reliable; the service provider may remove, change or update their services without giving notice to users. The reliability and fault tolerance is not well supported; faults may happen during the execution. Exception handling in the context of web services is still an open research issue.

Other approaches with nearly the same functionality as web services are Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM) or Sun Microsystems's Java/Remote Method Invocation (RMI).

To improve interoperability of web services, the WS-I publishes profiles. A profile is a set of core specifications (SOAP, WSDL, ...) in a specific version (SOAP 1.1, UDDI 2, ...) with some additional requirements to restrict the use of the core specifications. The WS-I also publishes use cases and test tools to help the deployment of profile compliant web services. The WS is the editing web service.

Additional specifications, WS

Some specifications have been developed or are currently being developed to extend web services capabilities. These specifications are generally referred to as WS-* specifications. Here is a non-exhaustive list of these WS-* specifications.

- **WS-Security**: Defines how to use XML Encryption and XML Signature in SOAP to secure message exchanges, as an alternative or extension to using HTTPS to secure the channel.
- **WS-Reliability**: An OASIS standard protocol for reliable messaging between two web services.
- **WS-Transaction**: A way of handling transactions.
- **WS-Addressing**: Is a standard way to insert address in the SOAP header.

Some of these additional specifications have come from the W3C. There is much discussion around the organization's participation, as the general Web and the Semantic Web paradigms appear to be at odds with much of the Web Services vision. This has surfaced most recently in February 2007, at the W3C Workshop on Web of Services for Enterprise Computing. Some of the participants advocated a withdrawal of the W3C from further WS-* related work, and a focus on the core Web. Web Services is a software systems that could published, located and binding in the web using XML protocols.

In contrast, OASIS has standardized many Web service extensions, including Web Services Resource Framework and WSDM.

**Styles of use**

Web services are a set of tools that can be used in a number of ways. The three most common styles of use are RPC, SOA and REST.

**Remote procedure calls**

RPC Web services present a distributed function (or method) call interface that is familiar to many developers. Typically, the basic unit of RPC Web services is the WSDL operation. The first Web services tools were focused on RPC, and as a result this style is widely deployed and supported. However, it is sometimes criticized for not being loosely coupled, because it was often implemented by mapping services directly to language-specific functions or method calls. Many vendors felt this approach to be a dead end, and pushed for RPC to be disallowed in the WS-I Basic Profile.

![Fig. 2: Architectural elements involved in the XML-RPC.](image-url)
III. Design methodologies

Web services can also be used to implement architecture according to SOA concepts, where the basic unit of communication is a message, rather than an operation. This is often referred to as "message-oriented" services.

SOA Web services are supported by most major software vendors and industry analysts. Unlike RPC Web services, loose coupling is more likely, because the focus is on the "contract" that WSDL provides, rather than the underlying implementation details.

Middleware Analysts use Enterprise Service Buses which combine message-oriented processing and Web Services to create an Event-driven SOA. One example of an open-source ESB is Mule, another one is Open ESB [38-50].

Represenational state transfer

Finally, Representational State Transfer (REST) attempts to describe architectures which use HTTP or similar protocols by constraining the interface to a set of well-known, standard operations (like GET, POST, PUT, DELETE for HTTP). Here, the focus is on interacting with stateful resources, rather than messages or operations. An architecture based on REST (one that is 'RESTful') can use WSDL to describe SOAP messaging over HTTP, which defines the operations, can be implemented as an abstraction purely on top of SOAP (e.g., WS-Transfer), or can be created without using SOAP at all.

WSDL version 2.0 offers support for binding to all the HTTP request methods (not only GET and POST as in version 1.1) so it enables a better implementation of RESTful Web services [27-43]. However, support for this specification is still poor in software development kits, which often offer tools only for WSDL 1.1.

Design methodologies

Web services can be written in two ways:

- A developer using the "bottom up method" first writes the implementing class in a programming language, and then uses a WSDL generating tool to expose its methods as a web service [26]. This is often the simpler approach.
- A developer using the "top down method" first writes the WSDL document and then uses a code generating tool to produce the class skeleton, which she later completes.

This way is more difficult but produces cleaner designs [27].

Criticisms

Critics of non-RESTful Web services often complain that they are too complex and based upon large software vendors or integrators, rather than typical open source implementations. There are open source implementations like Apache Axis and Apache CXF.

One key concern of the REST Web Service developers is that the SOAP WS toolkits make it easy to define new interfaces for remote interaction, often relying on introspection to extract the WSDL and service API from Java, C# or VB code. This is viewed as a feature by the SOAP stack authors (and many users) but it is argued that it can increase the brittleness of the systems, since a minor change on the server (even an upgrade of the SOAP stack) can result in different WSDL and a different service interface. The client-side classes that can be generated from WSDL and XSD descriptions of the service are often similarly tied to a particular version of the SOAP endpoint and can break if the endpoint changes or the client-side SOAP stack is upgraded. Well designed SOAP endpoints (with handwritten XSD and WSDL) do not suffer from this but there is still the problem that a custom interface for every service requires a custom client for every service.

There are also concerns about performance due to Web services' use of XML as a message format and SOAP and HTTP in enveloping and transport, however emerging XML parsing/indexing technologies, such as VTD-XML, promise to address those XML-related performance issues.

Similar efforts

Several other approaches exist to solve the set of problems that Web services address, both preceding and contemporary to it. RMI was one of many middleware systems that have seen wide deployment. More ambitious efforts like CORBA and DCOM attempted to affect distributed objects, which Web services implementations sometimes try to mimic. More basic efforts include XML-RPC, a precursor to SOAP that was only capable of RPC, and various forms of HTTP usage without SOAP.

Service-oriented modeling

Service-oriented modeling is the discipline of modeling business and systems, for the purpose of designing and specifying service-oriented business
systems within a service-oriented architecture [84-86].

Any service-oriented modeling methodology typically includes a modeling language that can be employed by both the 'problem domain organization' (the Business), and 'solution domain organization' (the Information Technology Department), whose unique perspectives typically influence the 'service' development life-cycle strategy and the projects implemented using that strategy.

Service-oriented modeling typically strives to create models that provide a comprehensive view of the analysis, design, and architecture of all 'Software Entities' in an organization, which can be understood by individuals with diverse levels of business and technical understanding. Service-oriented modeling typically encourages viewing software entities as 'assets' (service-oriented assets), and refers to these assets collectively as 'services'.

**Popular approaches to service-oriented modeling**

There are many different approaches that have been proposed for service modeling, including SOMA and SOMF.

**Service-oriented analysis and design (SOMA)**

IBM announced Service-Oriented Modeling and Architecture (SOMA) as the first publicly announced SOA-related methodology in 2004. SOMA refers to the more general domain of service modeling necessary to design and create SOA. SOMA covers a broader scope and implements service-oriented analysis and design (SOAD) through the identification, specification and realization of services, components that realize those services (a.k.a. "service components"), and flows that can be used to compose services.

SOMA includes an analysis and design method that extends traditional object-oriented and component-based analysis and design methods to include concerns relevant to and supporting SOA. It consists of three major phases of identification, specification and realization of the three main elements of SOA, namely, services, components that realize those services (aka service components) and flows that can be used to compose services.

SOMA is an end-to-end SOA Method for the identification, specification, realization and implementation of services (including information services), components, flows (processes/composition). SOMA builds on current techniques in areas such as domain analysis, functional areas grouping, variability-oriented analysis (VOA) process modeling, component-based development, object-oriented analysis and design and use case modeling. SOMA introduces new techniques such as goal-service modeling, service model creation and a service litmus test to help determine the granularity of a service.

SOMA identifies services, component boundaries, flows, compositions, and information through complementary techniques which include domain decomposition, goal-service modeling and existing asset analysis.

**Service-oriented modeling framework (SOMF)**

The Service-Oriented Modeling Framework (SOMF) has been proposed by author Michael Bell as a service-oriented modeling language for software development that employs disciplines and a universal language to provide tactical and strategic solutions to enterprise problems [72-92].
Service-oriented modeling is a SOA framework that identifies the various disciplines that guide SOA practitioners to conceptualize, analyze, design, and architect their service-oriented assets. The Service-Oriented Modeling Framework (SOMF) offers a work structure or "map" depicting the various components that contribute to a successful service-oriented modeling approach. It illustrates the major elements that identify the "what to do" aspects of a service development scheme. The model enables practitioners to craft a project plan and to identify the milestones of a service-oriented initiative. SOMF also provides a common modeling notation to address alignment between business and IT organizations.

SOMF is designed to address the following principles:

- business traceability
- architectural best-practices traceability
- technological traceability
- SOA value proposition
- software assets reuse
- SOA integration strategies
- technological abstraction and generalization
- architectural components abstraction

SOMF is a service-oriented development life cycle methodology. It offers a number of modeling practices and disciplines that contribute to a successful service-oriented life cycle management and modeling.

It illustrates the major elements that identify the "what to do" aspects of a service development scheme. These are the modeling pillars that will enable practitioners to craft an effective project plan and to identify the milestones of a service-oriented initiative—either a small or large-scale business or a technological venture.

The provided image thumb (on the right hand side) depicts the four sections of the modeling framework that identify the general direction and the corresponding units of work that make up a service-oriented modeling strategy: practices, environments, disciplines, and artifacts. Remember, these elements uncover the context of a modeling occupation and do not necessarily describe the process or the sequence of activities needed to fulfill modeling goals. These should be ironed out during the project plan – the service-oriented development life cycle strategy – that typically sets initiative boundaries, timeframe, responsibilities and accountabilities, and achievable project milestones [71-84].

SOA life cycle modeling

Service-oriented modeling is driven by the development process of services. This approach enables business and information technology professionals to focus on deliverables that correspond to a specific service-oriented life cycle stage and event.

Life cycle modeling activities

Service-oriented Modeling and Architecture (SOMA) consists of the phases of identification, specification, realization, implementation, deployment and management in which the fundamental building blocks of SOA are identified then refined and implemented in each phase. The fundamental building blocks of SOA consists of services, components, flows and related to them, information, policy and contracts.

The service-oriented modeling framework (SOMF) introduces five major life cycle modeling activities that drive a service evolution during design-time and run-time. At the design-time phase a service originates as a conceptual entity (conceptual service), later it transforms into an SOA unit of analysis (analysis service), next it transitions into a contractual and logical entity (design service), and finally is established as a concrete service (solution service). The following identify the major contributions of the service-oriented modeling activities:

- Service-oriented discovery & analysis modeling: Discover and analyze services for granularity, reusability, interoperability, loose-coupling, and identify consolidation opportunities.
- Service-oriented business integration modeling: Identify service integration and
alignment opportunities with business domains’ processes (organizations, products, geographical locations)

- Service-oriented logical design modeling: Establish service relationships and message exchange paths. Address service visibility. Craft service logical compositions. Model service transactions
- Service-oriented logical architecture modeling: Integrate SOA software assets. Establish SOA logical environment dependencies. Foster service reuse, loose coupling and consolidation.

Modeling styles

How can an SOA practitioner model a computing environment? In what type of forms can a group of services be arranged to enable an efficient integrated computing landscape? What would be the best message routes between a service consumer and provider? How can interoperability hurdles be mitigated?

SOMF provides four major SOA modeling styles that are useful throughout a service life cycle (conceptualization, discovery and analysis, business integration, logical design, conceptual and logical architecture). These modeling styles: Circular, Hierarchical, Network, and Star, can assist an SOA modeler with the following modeling aspects [84-86]:

- Identify service relationships: contextual and technological affiliations
- Establish message routes between consumers and services
- Provide efficient service orchestration and choreography methods
- Create powerful service transaction and behavioral patterns
- Offer valuable service packaging solutions

In the illustration on the right you will find the major four service-oriented modeling styles that SOMF offers. Each pattern identifies the various approaches and strategies that one should consider employing when modeling an SOA environment [84,85].

- **Circular Modeling Style**: enables message exchange in a circular fashion, rather than employing a controller to carry out the distribution of messages. The Circular Style also offers a conceptual method to affiliating services.
- **Hierarchical Modeling Style**: offers a relationship pattern between services for the purpose of establishing transactions and message exchange routes between consumers and services. The Hierarchical pattern founds parent/child associations between services.
- **Network Modeling Style**: this pattern establishes “many to many” relationship between services, their peer services, and consumers. The Network pattern accentuates on distributed environments and interoperable computing networks.
- **Star Modeling Style**: the Star pattern advocates arranging services in a star formation, in which the central service passes messages to its extending arms. The Star modeling style is often used in “multi casting” or “publish and subscribe” instances, where “solicitation” or “fire and forget” message styles are involved.
Modeling assets

The service-oriented modeling framework (SOMF) introduces three major service formations. These structures are software entities that habitually exist in our computing environments:

- **Atomic service**: an indivisible software component that is too granular and executes fewer business or technical functionalities. An atomic formation is also a piece of software that typically is not subject to decomposition analysis activities and its business or technological functionality does not justify breakdown to smaller components. Examples: customer address service and checking account balance service.

- **Composite service**: a composite service structure aggregates smaller and fine-grained services. This hierarchical service formation is characteristically known as coarse-grained entity that encompasses more business or technical processes. A composite service may aggregate atomic or other composite services. Examples: customer checking service that aggregates smaller checking account and savings account services. An application that is composed of sub-systems, an ESB that is composed of routing, orchestration, and data transformation components.

- **Service cluster**: this is a collection of distributed and related services that are gathered because of their mutual business or technological commonalities. A service cluster both affiliates services and combines their offerings to solve a business problem. A cluster structure can aggregate atomic as well as composite formations. Examples: A Mutual Funds service cluster that is composed of related and distributed mutual funds services. A Customer Support service cluster that offers automated bill payments, online account statements, and money transfer capabilities.

Fig. 7: The major service formations.

- **Atomic service**: an indivisible software component that is too granular and executes fewer business or technical functionalities. An atomic formation is also a piece of software that typically is not subject to decomposition analysis activities and its business or technological functionality does not justify breakdown to smaller components. Examples: customer address service and checking account balance service.

- **Composite service**: a composite service structure aggregates smaller and fine-grained services. This hierarchical service formation is characteristically known as coarse-grained entity that encompasses more business or technical processes. A composite service may aggregate atomic or other composite services. Examples: customer checking service that aggregates smaller checking account and savings account services. An application that is composed of sub-systems, an ESB that is composed of routing, orchestration, and data transformation components.

- **Service cluster**: this is a collection of distributed and related services that are gathered because of their mutual business or technological commonalities. A service cluster both affiliates services and combines their offerings to solve a business problem. A cluster structure can aggregate atomic as well as composite formations. Examples: A Mutual Funds service cluster that is composed of related and distributed mutual funds services. A Customer Support service cluster that offers automated bill payments, online account statements, and money transfer capabilities.

The SOMF also introduces a simple notation to describe the three major service formations, as illustrated in the below image: Atomic Service, Composite Service, and Service Cluster.

**Modeling notation**

The service-oriented discovery and analysis notation represents eight simple icons, each of which enables us to operate on a service (atomic or composite service) or on a group of services (service clusters) in different ways.

These operations promote core SOA best practices. Let us take a look at the below illustration to find out what these analysis operations mean and how they can be employed to draft an SOA analysis proposition diagram.

Fig. 9: Service-oriented analysis notation
Here is a short description for these symbols:

- **Aggregated**: signifies service containment
- **Subtracted**: identifies service retirement or elimination
- **Unified**: indicates service consolidation
- **Decomposed**: depicts service decoupling
- **Intersected**: illustrates a junction point between two or more service groups (known as service clusters)
- **Overlapped**: describes common functionality and processes that services provide (typically between service clusters)
- **Transformed**: represents transformation of a service type to another (atomic to composite, etc)
- **Comment**: a place to put notes

Service-Oriented Conceptualization

The service-oriented modeling framework (SOMF) advocates that practitioners devise conceptual services to bridge the communication gaps in an organization. These conceptual entities foster the creation of a common language, a vocabulary that can be used during projects to encourage asset reuse and consolidation. One of the most important aspects of the conceptual service paradigm is that it provides direction and defines the scope of a project or a business initiative.

The conceptualization process then identifies six major “tools” that can facilitate the development of conceptual services [84-92].

- **Concept Attribution**: this activity pertains to the collection of software products attributes that both describe service’s features and lead to the discovery of organizational taxonomy
- **Concept Classification**: the categorization effort contributes to separation of concerns and the establishment of service identities. This is the process of identifying service dissimilarities
- **Concept Association**: Unlike the classification activity, the association effort enables the discovery of service relationship. These can be business or technological affiliations
- **Concept Clustering**: this discipline is about grouping related conceptual services that collaboratively provide a solution. Clustering is a conceptual operation that can encompass local, remote, and virtual services
- **Concept Generalization**: to raise the abstraction level of a conceptual service, the generalization method increases the conceptual scope of a solution. This approach is typically used when a service is coarse-grained (too granular)
- **Concept Specification**: the specification activity enables architects, modelers, and developers to reduce the abstraction level of a service and thus reduce its granularity scope to finer-grained.

SOA Governance

SOA governance is a concept used for activities related to exercising control over services in an SOA. SOA governance can be seen as a subset of IT governance which itself is a subset of Corporate governance. The focus is on those resources to be leveraged for SOA to deliver value to the business. SOA requires a number of IT support processes as well as organizational processes that will also involve the business leaders. SOA needs a solid foundation that is based on standards and includes policies, contracts and service level agreements. The business is expected to be able to use services to build and change the organisations business process quickly. To do so, a degree of granularity in the services available will be required. Consequently an SOA increases the need for good governance as it will help assign decision-making authorities, roles and responsibilities and bring focus to the organizational capabilities needed to be successful [73,74].

The definitions of SOA governance agree in its purpose of exercising control, but differ in the responsibilities it should have. Some narrow definitions focus on imposing policies and monitoring services, while other definitions use a broader business-oriented perspective.

Anne Thomas Manes defines governance as: “The processes that an enterprise puts in place to ensure that things are done ... in accordance with best practices, architectural principles, government regulations, laws, and other determining factors. SOA governance refers to the processes used to govern adoption and implementation of SOA” [84-86].

The specific focus of SOA governance is on the development of services that add value to the business, effective SOA governance must cover the people, processes, and technologies involved in the entire SOA life cycle from business point of view.
and connectivity and reuse from IT point of view, thus aligning business with IT.

To quote Anne Thomas Manes again: “SOA is about behavior, not something you build or buy. You have to change behavior to make it effective.”[73]. Gartner defines SOA Governance as “Ensuring and validating that assets and artifacts within the architecture are acting as expected and maintaining a certain level of quality [74].

ISO 38500 describes a framework with six guiding principles for corporate governance of information technology and a model for directors to govern IT with three main tasks: evaluate, direct and control. ISO 38500 differentiates between "Governance", "Management" and "Control".

Some typical governance issues that are likely to emerge in a SOA are:

- **Delivering value to the stakeholders**: investments are expected to return a benefit to the stakeholders - this is equally true for SOA
- **Compliance to standards or laws**: IT systems require auditing to prove their compliance to regulations like [Sarbanes-Oxley]. In a SOA, service behavior is often unknown
- **Change management**: changing a service often has unforeseen consequences as the service consumers are unknown to the service providers. This makes an impact analysis for changing a service more difficult than usual.
- **Ensuring quality of services**: The flexibility of SOA to add new services requires extra attention for the quality of these services. This concerns both the quality of design and the quality of service. As services often call upon other services, one malfunctioning service can cause damage in many applications.

Some key activities that are often mentioned as being part of SOA governance are [73-75]:

- Managing the portfolio of services: planning development of new services and updating current services
- Managing the service lifecycle: meant to ensure that updates of services do not disturb current service consumers
- Using policies to restrict behavior: rules can be created that all services need to apply to, to ensure consistency of services
  - Monitoring performance of services: because of service composition, the consequences of service downtime or underperformance can be severe. By monitoring service performance and availability, action can be taken instantly when a problem occurs.

**SOA Security**

(SOA) allows different ways to develop applications by combining services. The main premise of SOA is to erase application boundaries and technology differences. As applications are opened up, how we can combine these services securely becomes an issue. Traditionally, security models have been hardcoded into applications and when capabilities of an application are opened up for use by other applications, the security models built into each application may not be good enough.

Several emerging technologies and standards address different aspects of the problem of security in SOA. Standards such as WS-Security, SAML, WS-Trust, WS-SecureConversation and WS-SecurityPolicy focus on the security and identity management aspects of SOA implementations that use Web Services. Technologies such as the Virtual Organization in Grid Computing, Application-oriented networking (AON) and XML Gateways are addressing the problem of SOA security in the larger context as well.

XML Gateways are hardware or software based solutions for enforcing identity and security for SOAP, XML, and REST based web services, usually at the network perimeter. An XML gateway is a dedicated application which allows for a more centralized approach to security and identity enforcement, similar to how a protocol firewall is deployed at the perimeter of a network for centralized access control at the connection and port level.

XML Gateway SOA Security features include PKI, Digital Signature, Encryption, XML Schema Validation, Antivirus, and Pattern Recognition. Regulatory certification for XML gateway security features are provided by FIPS and DoD [76-78].

**IV. Root Causes**

Many studies have discussed the performance of XML Web Services from different perspectives, and they all concluded that poor performance goes back to a number of reasons including:
• **XML Data Format:** The technology of XML Web Services depends on XML (Extensible Markup Language) for representing data being transmitted between different systems and nodes. As known, XML is a tag-based language rich with different capabilities that allow easy and powerful integration between different systems. For example, it offers different mechanisms for validating, querying, and transforming data [6]. These capabilities made XML the preferred choice of software vendors who look for interoperability and loose-coupling between integrated systems. However, this richness causes data files to be bloated with long named tags, complex data structures, and big amounts of plain-text data that make their generation and processing very complex, heavy, and slow.

• **Encoders/Decoders:** XML Web Services use encoders to transform data into a sequence of bytes before transmission, and on the other side, they use decoders to return transmitted data to its original form. ASCII was one of the most used encoding formats in legacy systems. Depending on ASCII format in XML Web Services is very expensive and slow especially for numerical values and floating points [7].

• **Parsing Techniques:** XML being transmitted between different nodes must be parsed and validates before any further processing. Using inefficient parsing techniques may require much hardware resources (including RAM and processing cycles) and long times to complete required tasks.

• **Serialization/De-serialization:** Serialization (marshaling) is the process that converts the state of objects into a form that can be transmitted over network media (such as wires and Wi-Fi) between different nodes. Conversely, de-serialization (de-marshalizing) process is responsible for bringing the state information to original formats. Efficient serialization techniques must be very fast and generate serialized data in compact formats. Depending on bad serialization techniques may generate very large data outputs that might clog network during transmission process. As mentioned, XML Web Services serialize data using XML format, and because XML is text-based format, then generated messages are always very large in size if compared to original data (before serialization) [7,8].

• **Transport Protocol:** XML Web Services use Simple Object Access Protocol (SOAP) as a lightweight communication framework that is based on XML. Although SOAP messages can use any transport protocol to send requests and receive responses, it uses HTTP as a default transport protocol. HTTP is a request-response protocol that supports only synchronous interaction between clients and servers, and this makes it ill-suited for message-based communications that require asynchronous interactions [9].

• **Network Infrastructure:** There is no question that, network is one of the most important factors for the success of any client/server implementations, and thus, depending on weak and slow networks can lead to unreliable, inefficient, and intermittent interactions between clients and servers. Furthermore, slow networks may lead to less utilization for available processing power because CPUs will wait longer times until data arrives to be processed.

• **Extra Elements in Software Stack:** In service-oriented systems, XML Web Services are defined in a separate layer that accepts clients’ requests to be formatted before sending them to underlying components. As known, the more layers and processing logic defined in any software architecture, the slower performance of overall system. This is due to extra processing that is needed to reach final element in software stack including discovery, reflection, initialization, instantiating, and invocations of needed components and objects as well as transformations for incoming requests and outgoing data formats.

**Technical Scenario**

Integrating different systems together is a very common scenario in software field, and leveraging SOA for building and integrating different business systems like banking solutions offers many advantages over other traditional integration methods including simplicity, dynamicity, agility, and loose-coupling between integrated systems [3]. For this reason, we are going to illustrate one of these scenarios in subsequent sections. Assume that we have the following two banking systems that need to be integrated together with minimum impacts on underlying architecture and components:

• **Core-Bank:** It is a legacy monolithic system that acts as a back-end for other operational (front-end) systems. Core-bank system consists of different business modules including (customer management, deposits, foreign exchange, commercial loans, Islamic loans, etc). Additionally, this system offers some non-functional features such as transaction support, role-based security, and auditing and logging for user actions. The core-bank system allows users to send requests and receive responses via its GUI while all processing occurs into database itself. This
A database consists of a large number of precompiled stored procedures, and each stored procedure takes a huge number of input parameters to process them and return generated result sets.

- **Loan-Origination**: It is a front-end system built with .NET framework 2.0 and C# language and is responsible for allowing bank clients to issue loans, define installments, schedule payments, etc. Because all information about bank clients and loans are stored into the core-banking system, we have to integrate the loan-origination system with it. To enable this integration, we will build an XML Web Service that comprises a large number of methods responsible for accepting requests from different client applications including the Loan-Origination application, and passing them to stored procedures that reside under the core-banking system. To enable efficient use of the new XML Web Service, software components should be built to wrap available stored procedures and encapsulate their logic. These components will divide underlying business logic that is scattered over different stored procedures into modules that could be easily used and modified whenever needed [11, 12]. Defined components could be realized (designed and built) with any modern programming environment (such as J2EE or .NET) that supports advanced features like OOP, RAD, XML Web Services, etc. After realizing needed components, they will be placed into a new layer that resides between the new XML Web Service and underlying stored procedures to act as a mediator/wrapper that accepts different invocations from XML Web Service and turn them into formats that could be accepted by stored procedures. Fig. 10 illustrates a high level view for integration architecture of our systems, whereas, Fig. 11 illustrates simple request and response messages that are handled by the system.

![Fig. 10: Integration Architecture](image-url)
V. Performance Evaluation

Due to high dependency on XML Web Services, the illustrated architecture slightly suffers from low performance. In fact, leveraging SOA is always hampered by large sizes of XML files being transferred between clients and servers. These large data files always clog network and drain almost all hardware resources including RAM, CPU, and storage infrastructure. In original architecture the core-banking system was entirely built over a set of stored procedures to execute needed business logic, but it now depends on a service layer that acts as a wrapper that receives client requests and maps them to appropriate stored procedure(s). Fig. 12 illustrates a comparison between the total times needed for receiving the response message of the request that gets basic information about one bank client using both methods.

---

**Fig. 11: An Example for Request/Response Messages**

<table>
<thead>
<tr>
<th>Request Message</th>
<th>Response Message</th>
</tr>
</thead>
</table>
| ```xml
  <Request ID="86C58720-42A0-1069-A2E8-08002B30309D"
  Date="1/1/2008 4:30:35 PM">
    <CustomerProfile>
      <CustomerID>10028160</CustomerID>
      <BankID>10</BankID>
      <BranchID>2</BranchID>
    </CustomerProfile>
  </Request>
``` | ```xml
  <Response ID="866D4166-DDA2-42AC-926A-C16F9127C302"
  Date="1/1/2008 4:30:40 PM">
    <CustomerProfile>
      <CustomerID>10028160</CustomerID>
      <CustomerName>Quasay Fadhel</CustomerName>
      <CustomerStatus>Active</CustomerStatus>
      <CustomerType>V.I.P</CustomerType>
      <Address>
        <Country>Egypt</Country>
        <City>Cairo</City>
        <Street>Free Zone, Nasr City</Street>
      </Address>
      <BranchID>2</BranchID>
      <BranchName>HSBC-Cairo-Al Mohandesseen</BranchName>
      <Account>
        <AccountNumber>000111012586</AccountNumber>
        <AccountType>Checking</AccountType>
        <AccountStatus>Open</AccountStatus>
      </Account>
      <CustomerProfile>
        <Response>
```

---
V. Optimization Tactics and Strategies

As illustrated, the original method that uses stored procedures is 4 times faster than the new XML Web Services method. To mitigate this problem, many experiments have been conducted to yield the following list of recommended tactics and techniques:

- **Utilize Better Encoders/Decoders:** Utilizing or even customizing more optimized encoders/decoders and encoding formats can save much of time needed for preparing data. UTF8 is a well known and standard encoding format that has been tuned to replace the traditional slow ASCII format. It is now known to be one of the fastest encoding options available in software market that supports almost all commonly used characters as well as special characters [13].

- **Leverage Binary XML:** As mentioned, most of current XML implementations depend on using plain-text format that causes data files to be very complex and large in size. W3C has announced that formatting XML data using binary format is more efficient for both network and hardware utilization [14]. Different techniques might be utilized to use binary data, for example, the data being transmitted between network nodes might be serialized (marshaled) and de-serialized (un-marshaled) using binary format instead of text-based XML format. Figures 13, 14 illustrate the results generated by a simple benchmark (windows forms) application that was written (using c# language and .NET Framework 3.5) to estimate the serialization time needed by binary and text-based serialization techniques on a PC that uses Intel Dual Core 2.6 GHz processor and 2 GB memory. As depicted, the application shows that XML serialization takes 290 milliseconds to serialize some of stored (dummy) information. The amount of data is represented in 20 rows and 5 columns (948 bytes), whereas, the binary serialization takes only 8 milliseconds to serialize the same data. This means that binary serialization in our scenario saves more than 97% of time needed by XML serialization. Certainly the serialization time will vary according to number of factors including amount and complexity of data being serialized, for instance, if we serialized 10,000 rows of the aforementioned data (569,808 bytes) on the same PC, the XML serialization will take 709 milliseconds versus 322 milliseconds for binary serialization, which means that binary serialization saves more than 55% of total serialization time. We should note that there is a fixed time that is needed in each method whatever the size and amount of data to initialize serialization process. This initialization operation is solely needed to read the schema of data being serialized. Fig. 14 illustrates a simple comparison between total times needed for binary serialization versus XML (text-based) serialization to return the response of a simple request.
Fig. 13: Performance of Binary Serialization vs. XML Serialization

Fig. 14: Binary Serialization vs. XML Serialization
• Apply Data Compression Techniques: Since XML is a text-based format, and XML documents always have too many white spaces, then using traditional compression algorithms such as ZIP/GZIP for compressing data transferring between clients and servers can get rid of many bytes of the volume out of data files. To apply compression technique on traveling data, both requestors and responders must understand the used compression algorithm to be able to recognize and use these data. This assurance should be identified and guaranteed by SOA governance team during preliminary implementation phases. Fig. 17 illustrates the results generated by a benchmark (console) application that has been written (using c# language and .NET Framework 3.5) to calculate total save in size of Response document that was illustrated in Fig. 13 using ZIP/GZIP algorithms. The results shows that the original size of Response document was 771 bytes, whereas, the size of compressed file is only 550 bytes which means that ZIP/GZIP algorithms save 221 bytes (approximately 29%) from original document size. Another way that could make XML documents smaller is to avoid long element and attribute names. For example, the Response document illustrated in Fig. 13 could be abbreviated as illustrated in Fig. 19. The size of new abbreviated Response document takes only 559 bytes with total save 212 bytes (approximately 28%) from original document size. This option will only be applicable where the human readability of request/response messages is not required. Combining two
techniques together in our scenario allowed us to eliminate about 57% (the most) of total document size which is excellent to our issue while keeping the great benefits of SOA including the ease-of-use and simplicity of XML Web Services and overall architecture. Certainly, these values may vary depending on the structure and size of original documents being compressed, however, saving total size of documents (especially complex and large ones) being processed and transmitted over network has with no doubt positive impact on memory consumption, network utilization, and transmission time. One note that we should take into consideration regarding to applying ZIP/GZIP algorithms is that processing compressed data may require more CPU usage at the receiving (last) node, as it will be a part of its responsibilities to unzip data in order to be able to use it in any further processing. This issue can be easily resolved by using more powerful servers that use high speed multi-processing cores.

![Image: Total Save in Response Document Using ZIP/GZIP Algorithms](image.png)

**Fig. 17: Total Save in Response Document Using ZIP/GZIP Algorithms**
Use Better Parsers: SAX and DOM are the most popular parsing techniques available in XML market. Many benchmarks on their throughputs have shown that they require slightly long times to parse XML files at different sizes and complexity levels. The long time is mainly needed because these parsers read data files more than once (at least two times) to be able to discover their structure and to validate entire data. Furthermore, big amount of memory might be needed to store extracted data during parsing phase, and this of course may degrade overall performance of used CPUs when no more memory is available (in paging and caching operations performed by operating system). To resolve this issue, we may depend on faster and more efficient techniques such as Virtual Token Descriptor XML (VTD-XML) which depends on “non-extractive” tokenization approach to parse XML files [15]. Additionally, using schema-specific parsers rather than general purpose parsers can greatly enhance the performance of parsing phase [16]. Using more enhanced parsing algorithms and tools can save time needed to parse data which in turn save the overall processing time and resources.

Pre-generate Serialization Assemblies: Many of development tools (including .NET 2.0 and latter) allow developers to pre-generate and cache serialization assemblies that could be deployed with applications to save time needed for discovering, extracting, and recognizing structures of objects being serialized and de-serialized [17].

Divide Large Files: It is known that large files always have bad impacts on the utilization of hardware resources (memory, processor, and network) and processing time. Dividing large and complex data files into smaller pieces (if
possible) can deliberately enhance both processing and network performance.

- **Install Silicon-based XML Engines**: Many silicon-based engines are available now to handle XML at higher speeds. These engines can be embedded into different network and hardware equipments including switches, routers, load balancers, PCI-cards and servers [18].

- **Apply Parallelism Techniques**: Data files could be yielded and processed in parallel using grid-based technologies that depend on multi-threaded systems (one thread for each process/sub-process) to allow faster processing and better utilization of available hardware resources including processing cycles, memory, and storage infrastructure [19, 20, 21].

- **Utilize High Speed Networks**: There are now Ethernet implementations that enable enterprises to have transmission speed that varies from traditional 10Mbits/s Ethernet to 100Gigabit Ethernet [22, 23, 24, 25]. Also, fiber channels and links could be installed to allow high speed and reliable transmission for data encapsulated into XML requests and replies.

**VI. Conclusion**

Design considerations and implementation of SOA have been presented. Different types of modeling have been discussed. Furthermore, the problems of XML Web Services have been solved. In addition, the evaluation for XML Web Services in real-time business systems has been given by illustrating a common scenario for two banking systems that need to be integrated together with contemporary SOA methodologies and terms. The use of XML Web Services in our scenario caused a number of problems including slow performance and bad utilization for hardware and network resources over RPC implementation (stored procedures) that is widely used in traditional point-to-point integration methods. These pitfalls could not be accepted under any circumstances in real-time business systems that receive and handle tons of requests every single second. For that reason, some tips and recommended actions have been given to allow efficient and better use of SOA and XML Web Services in building and integrating real-time business applications. Moreover, web service development has been discussed. Moreover, the management and security of SOA have been described.

**References**


[31] Position Paper For the Workshop on Web of Services for Enterprise Computing (submitted by Nick Gall)
[33] ongoing · WS-Pagecount
[34] Rethinking the Java SOAP Stack
[37] An alternative view, particularly after initial deployments, denies that SOAs properly ought to dictate physical implementation, so the formal definition should not include "network". High-performance SOAs may not be viable, especially if deployed to distributed nodes on a network. Separate nodes for every (or most) services could become prohibitively expensive.
[38] Channabasavaiah, Holley and Tuggle, Migrating to a service-oriented architecture, IBM DeveloperWorks, 16 Dec 2003.
[39] Yvonne Balzer Improve your SOA project plans, IBM, 16 July 2004
[40] Thomas Erl Serviceorientation.org – About the Principles, 2005-2006
[42] Enterprise SOA. Prentice Hall, 2005
[44] SOA Reference Model definition
[46] Christopher Koch A New Blueprint For The Enterprise, CIO Magazine, Mar 1 2005
[48] Martin van den berg et al. SOA for Profit, A Manager's Guide to Success with Service-Oriented Architecture (Hardcover), 978-9075414141
[51] WS-I Basic Profile
[53] WS-MetadataExchange OWL-S
[54] The Overlapping Worlds of SaaS and SOA
[56] Index XML documents with VTD-XML
[57] The Performance Woe of Binary XML
[58] Manipulate XML Content the Simple Way
[64] Jason Bloomberg Mashups and SOBAs: Which is the Tail and Which is the Dog?, Zapthink
[72] Anne Thomas Manes, The Elephant Has Left The Building, 1 July 2005
[74] Gartner, Magic Quadrant for SOA Governance, 2007