ARCHITECTURAL PATTERNS AND FUNCTIONAL PROGRAMMING CONCEPTS SUGGESTED TO HANDLE ISSUES IN WEB SERVICES FOR BIG DATA

[1] M.Kalidas, [2] N.Rama Devi
[1] Assistant Professor, [2] Professor
[1] Department of MCA, CBIT(A), Hyderabad, India

Abstract : In today's world, there is a huge increase in the demand for web services and a corresponding need for ultramodern infrastructure and technology to implement them. Part of this demand stems from the explosion of users and their transactions across various devices and social networks connecting them. In this paper, we take a look at the role and importance of web services and their implementations across social networks and processes and examine some of the concerns and issues that arise when they are subjected to present-day and near-future usage scenarios and the suggestions and recommend architectural patterns to address them.

I. INTRODUCTION

The demand for web services to meet diverse needs of applications, processes, and users worldwide on a real time basis has increased tremendously in Modern day scenario.

. Part of this demand is due to :

a) Increase in the number of new users; b) Increase in the number of new and affordable devices with good 3G/4G connectivity; c) A variety of diverse new mobile applications (apps) intended for users on the move; d) Business processes are getting more integrated, device-enabled, and interdependent.

Another part of this demand arises due to the need to preserve metadata about users and their transactions in real time, including but not limited to profiles, history, and log data. This metadata accumulates in enormous quantities over time leading to a combinatorial explosion in terms of user activity, usage patterns and transaction history.

All this, presently known by the umbrella term Big Data, cannot be archived, or backed up easily, and cannot be deleted or erased securely even if it is outdated. It needs to be preserved mostly intact with valid identities and timestamps, and it needs to be available to web services and search engines for indexing, data mining, seeking patterns, history, and logs. Identification and authentication mechanisms are getting more standardized and commonly used (examples are OpenID, OAuth, Facebook/Gmail/LinkedIn single sign on). Existing web services are increasingly invoking other remote web services for certain operations and tasks that were earlier not needed or performed locally. Authentication, geolocation, email address verification are some examples. Membership in one or more social networks is either required or taken for granted (Facebook, Twitter, Gmail, Yahoo, MSN, LinkedIn).These trends indicate that in the near future the number of simultaneous connections made to popular web services at any given point in time is likely to increase beyond the capacity of the existing implementations of these web services.

For example, we can see that having to deal with a sudden and huge surge in simultaneous connections on specific time slots / days such as exam results dates, train / air travelling seasons causes the servers to crash, unable to withstand the heavy load. Since there is a practical limitation to the number of redundant/load-balancing servers that any given web service can use, these servers cannot handle such huge volumes of transactions and thus are prone to fail. Existing web service implementation technologies start revealing their age, weaknesses, and limitations when situations such as these begin to occur on a daily / hourly basis.

II. CHALLENGES AND ISSUES

Present day web services and their implementations have several issues which have not been adequately addressed.

Table 2.1: Web Services implementation services

Availability	Uptime - is the web service available at all times (24x7) ? What is the percentage of downtime if any?
Reliability	To what extent does the web service remain unaffected by adverse networking conditions and hardware/data integrity errors?
State, persistence and complexity	How is session state and/or transaction status preserved, transmitted, or retrieved in a stateless HTTP-based protocol? Is the data persistent on the client in the form of cookies, or on the server, or in some other way? How does this affect the complexity of implementing the business logic of the transaction?

Maintenance and cost-effectiveness	Does the web service require testing and maintenance on a regular basis? Is it expensive to test and maintain the web service?
Flexibility to add support for new features	How easy, difficult, or expensive is it to implement new features in addition to existing ones? Does the addition of new features require significant change management, impact analysis or regression testing? Can the changes and additions be added so as to minimize impact on complexity and maintenance?
Bandwidth requirements	Will the implementation of the web service require high bandwidth and high data transfer speeds? Will there be real-time transmission of large amounts of compressed and/or encrypted data (such as signatures, fingerprints or retinal scans for identity verification)?
Security	Will the web service authenticate its users using HTTPS? (SSL and encryption)? To what extent can it be implemented to detect / forestall /prevent attempts to hack it (by unauthorized users) or bring it down? (For example by DoS -- Denial of Service -- attacks)?

Known issues about web services that need to be carefully addressed are:

Size of data: Web Services are very simple and easy to use in all respects. However, the large size of the request and response data packets is sometimes an issue .Since web services use plain text protocols, they utilize a fairly verbose method to identify data. This is especially more so in the case of XML structures used by WSDL and SOAP-based web services. The typical request and response can be quite large (tens of kilobytes). This extra size usually does not cause any performance issues. However, if bandwidth is low or network traffic is very heavy and/or the server's network infrastructure is heavily loaded this can turn out to be a genuine bottleneck. One way to solve this would involve compression. Compressing and decompressing on-the-fly can be used to shrink the size of textual data dramatically (this has no effect on the size of already-compressed binary data such as images, which are usually transmitted as Base64 encoded attachments).

But this needs to be done every time between interfaces and layers during marshaling. It will increase latency and reduce turnaround time in addition to placing higher runtime processing burden on CPUs.

Duration:

HTTP and HTTPS are simple and fundamental web protocols which are used frequently and extensively. They are not meant for long sessions or transactions of extended duration. A browser making a HTTP connection merely requests a HTML/XHTML page and its related images and then disconnects after downloading them. The transaction is therefore limited to this. Unlike HTTP, under older binary protocols such as CORBA, Comet or RMI, a client may connect to the server and stay connected for an extended period of time. To achieve this with multiple HTTP connections, the developer needs to do put in extra efforts in order to make up for the limitations of HTTP, such as AJAX and XML Http Request.

Sharing/preserving of state:

In fact, both HTTP and HTTPS protocols are meant to be stateless. In general, the time spent in the interaction between the server and the client is usually quite short. If no stateful data is being exchanged between the server and the client and preserved on either side, they will have absolutely no knowledge of each other's states.

Keeping track of each other's state:

In an ideal web service connection, a server will send some kind of session identification to the client when the client first accesses the server. The client then uses this identification when it makes further requests to the server. This is useful since the server then can reload the information about the specific client. To be more specific, if the client request to the server is in progress and the server is about the respond and in case the connection is lost due to a sudden power outage for instance, then the server will have no way of knowing that the client is no longer active. The only way the server can determine that the client is no longer active is if there is a timeout mechanism.

If there is no activity on the client's part for a specified (user-configurable) amount of time, the server will assume the client is no longer active and close the session data.

Session state:

Session data can be stored at the client side or at the server side. It is common for web services to store session data in cookies on their client (end-user). Cookies being files on a user's physical machine are subject to data corruption and loss. Users may disable cookies or delete them inadvertently. Cookies may be copied, modified, replaced, or misused leading to security loopholes. Storing the session data on the server instead of on the client's physical system alleviates disadvantages of cookies, but this is not considered practical due to the huge numbers of users and their session data that servers need to track all the time (it directly leads to the combinatorial explosion problem).

Distributed transactions:

If the environment requires distributed transactions with heterogeneous resources spread across servers, networks and operating systems, there are inadequate standardized protocols and architectures that can be used to implement them. [1] Their concerns and requirements need to be studied and existing workable solutions for them need to be tested to determine if they work. New ones should be evolved if necessary.

Quality of Service (QoS):

In case of a mission-critical solution, the service providers must examine the reliability and performance of the service under peak load and uncertain conditions for high availability. The system must be capable of withstanding a diverse spectrum of load and network traffic conditions and still maintain uptime and reasonable throughput. The exposed infrastructure must provide load balancing, and fail-over and fault tolerance, to resolve these scenarios.

Security:

Web services are exposed to the general public using HTTP-based protocols. As web services are used by the general public, they must preferably be implemented using authentication and authorization mechanisms and using SSL-enabling encryption of the messages for preventing fraud, authenticating the user(s) and securing the usage. Adopting open security standards like OpenID, SAML, XML Encryption, XML Signature, or XACML or even biometric images such as fingerprint scan or retinal scan images may be a solution. To address the issues we have covered, the technology, infrastructure and the architecture of web services need to be upgraded.

This is already in progress at various levels and has been partially completed to some extent. However, there are still a vast number of web services that simply have not been able to keep up with the need for change. This is especially true when the cost of change in terms of time and availability of skilled developers is high, sometimes prohibitively so. Earlier these included HTTP/HTTPS, XML/RPC, AJAX, SOAP and WSDL which are primarily based on XML. Alternatively in the present, REST (Representational State Transfer) architecture with JSON (JavaScript Object Notation) as the data interchange format has become very popular, with the ability for any web resource located anywhere on the internet being addressable by REST-like URLs (also known as endpoints) that serve and receive JSON data. One critical aspect of the above mechanisms and approaches is the management of application state across HTTP/HTTPS, which is a stateless protocol by design.

III. ARCHITECTURAL PATTERNS SUGGESTED [2] & [6]

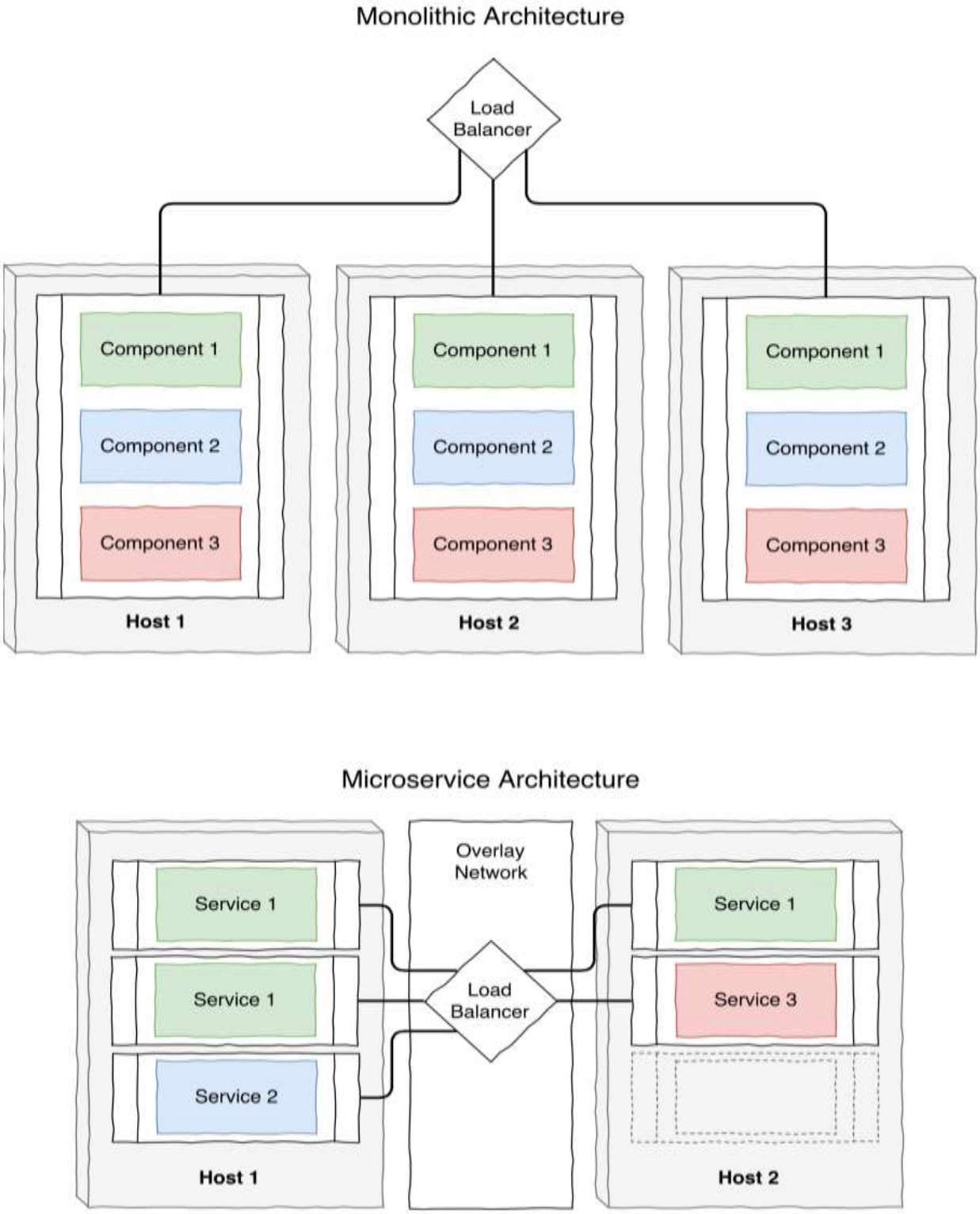

Figure 3.1: Monolithic and Microservice Architecture

BENEFITS:

The benefits of decomposing an application into different smaller services are numerous: [2] &[3]

- Modularity: This makes the application easier to understand, develop, test, and become more resilient to architecture erosion. This benefit is often argued in comparison to the complexity of monolithic architectures.
- Scalability: Since microservices are implemented and deployed independently of each other, i.e. they run within independent processes, they can be monitored and scaled independently.
- Integration of heterogeneous and legacy systems: Microservices are considered as a viable mean for modernizing existing monolithic software application. There are experience reports of several companies who have successfully replaced (parts of) their existing software by microservices or are in the process of doing so. The process for software modernization of legacy applications is done using an incremental or piecemeal approach.

- Distributed development: It parallelizes development by enabling small autonomous teams to develop, deploy and scale their respective services independently. It also allows the architecture of an individual service to emerge through continuous refactoring. Microservice-based architectures facilitate CI (continuous integration), continuous delivery and prompt deployment.

RESOURCE DESCRIPTION FRAMEWORK (RDF):

The Resource Description Framework (RDF) is a general method for conceptual description or modeling of information that is implemented in web resources. It is also used in Knowledge Management applications that work with semantic definitions and business knowledge. The purpose of RDF is to separate the knowledge or descriptive information content of a web resource (such as a website, web page or URL, or API endpoint), independent of the way the web resource is stored, retrieved and/or accessed. RDF denotes the semantic interpretation of the content of a web resource in a specific syntax among several such representations. RDF has been adopted as a W3C (World Wide Web Consortium) as an industry standard form of content representation, particularly so for knowledge-based applications and the Semantic Web. The RDF data model is similar to classical conceptual modeling approaches used in relational database (RDBMS) design (such as entity-relationship or class diagrams). It is based on the idea of making factual statements about resources (web resources in particular) in assertive and declarative expressions of the form:

<subject> - <predicate> - <object> which is known as an RDF triple. In this form,
<subject> denotes an entity (web resource) and is mapped to its URL,
<predicate> denotes a trait, an attribute, or an aspect of the web resource, and expresses a relationship. between subject and the object. It is also precisely defined at a commonly accepted URL.
<object> denotes another (different) entity (web resource) which is also mapped to its URL.

As an example, one way to represent the notion.
"The forest has the color green" in RDF syntax notation is as the RDF triple:
<forest> <has_color> <green>
This triple represents a subject denoting "the forest", a predicate denoting "has the color", and an object denoting the specific color "green". This mechanism for describing web resources is a major component in the W3C's Semantic Web architecture.
The Semantic Web is an evolutionary stage of the World Wide Web in which automated software can represent, store, exchange, and use machine-readable information distributed throughout the Web. In turn it also enables users to query and retrieve the information with greater efficiency and certainty. RDF's simple structure has the capacity to model disparate, abstract concepts and so this has also led to its increasing use in other areas, even in knowledge management applications unrelated to Semantic Web activity. Using RDF, the entities, objects, classes, instances and their respective attributes, qualities, aspects and metadata are asserted as facts or axioms in the OWL 2 functional syntax.

FUNCTIONAL PROGRAMMING AND WEB SERVICES [4] [5]:

In the paradigm of functional programming, all data is considered immutable, i.e., no value can be overwritten or updated or replaced by another value in the same memory where it is stored. New values may be created at any time by extracting parts of existing values and combining them with other values and manipulating the results. However once created, these new values will again be immutable, i.e., read-only. Applying this to the present scenario, data values are RDF triples and axioms, which are represented in the OWL 2 functional syntax. These can be processed as immutable data using functional programming techniques. That means data in the form of OWL 2 RDF triples can be read, stored, retrieved, extracted, combined and manipulated using functional programming in such a way that the desired results can be achieved in the same form without directly updating, modifying, overwriting or replacing parts of these immutable data values. This can be of immense benefit in simplifying the complexity in the structure, storage, parsing, extraction, business logic and application-level access and information retrieval.

Some issues are currently being remedied or mitigated to a good extent by :-

a) Switching from dedicated hardware to scalable virtual servers and virtual environments
b) Implementing web services on Platform-As-A-Sevice offerings (cloud-based web services)
c) Breaking down web services into several smaller components and implementing each of these components as a RESTful micro web service on its own, using server-side techniques such as URL rewriting, AJAX, and XML Http Request with JSON to exchange data between them
d) Requiring OpenID or OAuth access (user needs to login first, using an email address and a password, and after verifying the email address, a unique authorization key is generated for that user and reused every time that user logs in)

It has become common to allow logging in using Facebook, Gmail, Yahoo or LinkedIn login credentials, so that the authorization key can be shared across social networks.

IV. CONCLUSION

Apart from the recommendations as suggested above, in certain fundamental issues, we must do more with the design, programming style, modularity and loose coupling of components than anything else. These involve the issues of state, persistence, complexity, and flexibility as discussed earlier.

To address these, it may be necessary to take a deep and further look at how we implement the web services. What we conclude thus, the design and architecture and implementing them along with functional programming paradigm the

implementational frame work can drastically improve the effective usage and operations of the implementations of Web services in modern day scenario.

REFERENCES

[1] Armstrong, Joe (2003). "Making reliable distributed systems in the presence of software errors"(PDF). PhD Dissertation. The Royal Institute of Technology, Stockholm, Sweden.

[2] Fielding, Roy Thomas (2000). "Representational State Transfer (REST)" https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm) "*Architectural Styles and the Design of Network-based Software Architectures*" (https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm) PhD Dissertation. University of California, Irvine.

[3] Fielding, Roy T. (20 Oct 2008). "*REST APIs must be hypertext-driven*" (https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven).

[4] Vinoski, Steve. "*RESTful Services with Erlang and Yaws*", article in InfoQ online magazine.

[5] (https://www.infoq.com/articles/vinoski-erlang-rest/). Steve Vinoski's blog:). http://steve.vinoski.net/blog/

[6] Nicola Dragoni, Ivan Lanese, Stephan Larsen, Manuel Mazzara, Ruslan Mustafin, et al. "*Microservices: How To Make Your Application Scale*". .P. Ershov Informatics Conference (the PSI Conference Series, 11th edition), Jun 2017, Moscow, Russia. HAL Id: hal-01636132 (https://hal.inria.fr/hal-01636132).