SENRA Academic Publishers, British Columbia
Vol. 8, No. 2, pp. 2943-2954, June 2014
Online ISSN: 1920-3853; Print ISSN: 1715-9997

Canadian Journal of pure&applied sciences

TAIL TOLERANCE OF WEB SERVICES SOLUTION BUILT ON REPLICATION ORIENTED ARCHITECTURE (ROA)

*Godspower O Ekuobase[1] and Ifeanyichukwu E Anyaorah[2]
[1]Department of Computer Science, University of Benin, Benin City, Edo State
[2]Department of Computer Science, Auchi Polytechnic, Auchi, Edo State, Nigeria

ABSTRACT

Guaranteed responsiveness of Web Services solutions may not be possible on a large scale, if the solutions are not tail tolerant i.e. able to consistently keep latency within reasonable limit. Software techniques that tolerate latency variability and in particular, tail latency are vital to building responsive large-scale Web services solutions. Replication Oriented Architecture (ROA) though proposed to help application programmers build scalable Web Services solutions appears capable of mitigating latency variability and tail latency. Consequently, we investigated ROA for tail tolerance. To do this, we built two ATM Web Services solution using Java technology – the first was not built on ROA (conventional solution) but the other was built on ROA (ROA solution). These Web Services solutions were subjected to load performance test using Apache JMeter. The results showed that the tail tolerance of Web Services solution built on ROA is significantly better than its equivalent conventional solution. Specifically, we established that ROA is capable of improving the tail tolerance of Web Services solution by about 4.60% with 96% confidence. The results also affirm the scalability capability of ROA.

Keywords: ROA, latency variability, tail tolerance, java EE and web services.

INTRODUCTION

Scalability of Web Services is vital to its large scale deployment (Ekuobase and Onibere, 2011). However, guaranteed responsiveness of web services solutions may not be possible on a large scale, if the solutions are not tail tolerant (Dean and Barroso, 2013). Tail tolerant systems are systems that tolerate or mitigate latency variability including high tail-latency i.e. rare outrageous response times (Dean and Barroso, 2013). High tail-latency is therefore a serious threat to a responsive large scale Web Services solution.

Several techniques basically centered on replication have been proposed to curb high tail-latency in online service systems/applications (Dean and Barroso, 2013). These systems/applications are seriously prone to the problem of latency variability and tail latency. However, none of these techniques appears to target building responsive large scale Web Services solution (service applications built on the middle ware architecture called web services). Web Services solution has a unique nature of stateful/conversational asynchronous distributed orientation and use of TCP based technology such as SOAP (Baldoni *et al*., 2002; Ekuobase and Onibere, 2011, 2013; Ekuobase and Ebietomere, 2012). This drew our attention to the server-side software architecture – Replication Oriented Architecture (ROA) proposed by Ekuobase and Onibere (2011) which is aimed at helping application programmers build scalable Web Services solution. However, a critical examination of ROA in our domain of interest – latency variability and tail tolerance, exposed the need to also authenticate ROA's capability to mitigate latency variability including high tail-latency. The following observations encouraged this decision:

- The originators of ROA (Ekuobase and Onibere, 2011) only saw latency variability (guaranteed responsiveness) as an inherent scalability attribute (Ekuobase and Ebietomere, 2012) but never investigated it specifically for latency variability much less high tail-latency (Ekuobase and Onibere, 2013); thus creating an impression that scalable (web services) applications also guarantee responsiveness.
- Round the clock guaranteed responsiveness is a critical attribute of tail-tolerant systems but they could only give a 90% guarantee of 32% scalability assurance.

Consequently, we investigated whether or not ROA accommodates tail tolerance and if it does, by how much? This is the essence of this project, to determine the tail-tolerance capability of ROA.

The data collected, manipulated and interpreted were basically response times i.e. the time needed to process a query which is the time from sending a request until receiving the response (Yang *et al*., 2006; Repp *et al*., 2007). It is an important attribute of Web Services' Performance (Yang *et al*., 2006; Repp *et al*., 2007). According to the World Wide Web consortium (W3C), performance is defined in terms of throughput, response

*Corresponding author email: godspower.ekuobase@uniben.edu time, latency, execution time, and transaction time (www.w3.org). However, execution time and latency are sub-concepts of the W3Cs definition of response time (Repp *et al*., 2007).

A service's response time for a request, R, can be represented as shown below:

Response time(R) = Execution time(R) + Waiting time(R) (1)

The execution time is the duration of performing service functionality. The waiting time is the amount of time for all possible mediate events including message transmissions between service consumers and providers (Yang *et al*., 2006). From the service consumer perspective, we can see response time as the duration starting from the issue of a request to the end of the receipt of a service's response. On the other hand, service providers see response time as not being different from the execution time of a service, so it does not include all possible mediate events, which are seen as incontrollable variables during service execution (Yang *et al*., 2006).

Latency, which is an attribute of response time, is defined as the delay between the start of a message transmission from one process and the beginning of its receipt by another. It can be measured as the time required in transferring an empty message (Coulouris *et al*., 2012). In its general sense, it covers; (1) the time taken for the first bit of a string of bits transmitted in a distributed system to reach its destination. (2) The delay in accessing resources, which increases significantly when the system is heavily loaded, and (3) The time taken by the system's communication services at both the sending and the receiving processes, which varies according to the current load on the system (Coulouris *et al*., 2012).

The data transfer rate (speed at which data can be transferred between the two resources in distributed systems once transmission or message passing has begun, usually quoted in bits per second) is also a contributing factor to performance. It is determined primarily by physical characteristics, whereas latency is determined primarily by software overheads, routing delays and a load-dependent statistical element arising from conflicting demands for access to transmission channels. Considering that many of the messages transferred between processes in distributed systems are small in size; latency is therefore often of equal or greater significance than the transfer rate in determining performance (Coulouris *et al*., 2012). Thus, Web Services solution's architecture like ROA should be able to mitigate latency variability and present to the service consumers a consistent response time that is within an acceptable standard.

Keeping latency consistent within reasonable limit, thereby keeping the tail of latency distribution short, is a challenge to Web Services solutions as the size and complexity of the system scales up or as overall use increases (Dean and Barroso, 2013). Latency variability results from occasional high-latency episodes that tends to overshadow the overall service performances of large scale systems/applications. Software techniques that tolerate latency variability are vital to building responsive large-scale Web services (Dean and Barroso, 2013). Factors that may encourage variability in latency include (Dean and Barroso, 2013):

- ➢ Sharing of systems resources such as CPU cores, processor caches, memory bandwidth etc. between and within applications.
- ➢ Usage of systems resources by background daemons.
- ➢ Global resource sharing by applications running on machines.
- ➢ Periodic maintenance activities.
- ➢ Multiple layers of queuing in intermediate resource such as servers and network switches.
- ➢ Garbage collection
- ➢ Energy management due to switching between inactive power saving modes and active modes

Dean and Barroso (2013) established that it is not feasible to eliminate latency variability completely and hence introduced two tail-tolerant techniques that mask or work around temporary latency deviations. The techniques are of two classes: the Within-Request Short-Term Adaptations and the Cross-Request Long-Term Adaptations.

Within-Request Short-Term Adaptations (WRSTA)

This class of techniques basically deploys multiple replicas of data items to provide additional throughput capacity and maintain availability in the presence of failures. One challenge posed by WRSTA is that it is basically suited for read-only and loosely consistent datasets, and is effective only when the phenomena that causes variability does not tend to simultaneously affect multiple request replicas. The techniques under this class include Hedged and Tied requests:

Hedged requests: Here a user send the same request to multiple replicas (e.g. servers) and use the results from whichever replica that responds first. The client first send the request to the replica believed to be the most appropriate but then falls back on sending a secondary request after a brief delay. Once a response is received, other requests are cancelled.

Tied requests: Here a request is simultaneously queued in multiple replicas. The replicas communicate with one another concerning the status of the resultant responses. An executing server sends a cancellation message to the other servers. Delay interval can be introduced to avoid sending the request at the same time.

Observe that these techniques will likely congest transmission channels further and result in wastage of computational resources. Besides, they defy a necessary property of replication - transparency (Coulouris *et al*., 2012; Ekuobase and Onibere, 2011).

Cross-Request Long-Term Adaptations (CRLTA)
These techniques are suited for reducing latency variability caused by coarse-grained phenomena such as service-time variations and load balancing. They include:

Micro-Partitions: The systems generate many more partitions than there are machines in the service, then do dynamic assignment and load balancing of these partitions to particular machines.

Selective Replication: This is an enhancement of the micro-partitioning scheme, it detects items that are likely to cause load imbalance and create additional replicas of these items. Load balancing systems can then use the additional replicas to spread the load of these hot micro-partitions across multiple machines without having to actually move the micro partitions.

Latency-Induced Probation: Here machines with high latency are placed on probation and reincorporated when its latency has improved.

Though CRLTA addresses the problem of transparency, resource wastage and congestion of transmission media, it is however difficult to implement. Besides, they are more oriented towards handling latency at the systems level and not at the application level. A situation we choose to refer to as macro and micro latency respectively. ROA appears to be more oriented towards handling micro latency.

MATERIALS AND METHODS

The following sub-section describes the hardware and software tools as well as the process used in this research.

Hardware Tools
A notebook computer (HP Pavilion dv6 Notebook PC, Intel® Core(TM) i3 CPU @ 2.13 GHz 2.13 GHz, 4.0GB of RAM and 300GB of Hard Disk) was used not only in development and testing of the Web Services solution but also to carry out performance test to check for latency variability and tail tolerance. It also served as host to the software used and developed in this research. A lower configuration may not conveniently cope with the huge size and nature of our development platform as well as the high computational resource requirements for executing our applications and testing them for tail tolerance.

Software Tools
We shall discuss software tools under Operating System, Development Platform, Language, Integrated Development Environment (IDE) and Packages.

- ❖ **Operating System:** We settled for Microsoft Windows 7 Home Premium edition which worked seamlessly with the other tools used in the research. The Operating System enabled our applications and other software tools to interact with the machine and tap its computational and peripheral resources.

- ❖ **Development Platform**: The choice of Java EE (Jendrock *et al*., 2006) as our development platform for building the Web Services solution in preference to the .NET platform is because Java EE is non-proprietary and it rivals with .NET platform as the dominant application developer's platform for enterprise applications in general and Web Services solution in particular (Vawter and Roman, 2001; Williams, 2003; Birman, 2005). Also our prior comfortable programming experience in Java boosted our choice of Java EE.

- ❖ **Language:** Language here covers programming language, modeling language and Database Management System (DBMS). Java 7.0 was the preferred programming language of choice since the application was built on Java EE platform which has support for only Java. Java Persistence Query Language (JPQL), a version of the Structured Query Language (SQL) was adopted because of its rich Application Programming Interfaces (APIs) for interacting with databases. Our choice of Objectdb as our DBMS for building and managing the databases was based on its very good performance and seamless compatibility with Java and Netbeans – our Integrated Development Environment of choice.

- ❖ **Integrated Development Environment (IDE):** We have several IDEs that support Java and these include JBuilder, JCreator, Eclipse, and Netbeans. We chose Netbeans (Netbeans 7.0) on the ground of familiarity though it is not in any way less powerful than the others. IDE makes application development easy, nimble and interesting.

- ❖ **Packages:** Argo UML (Ramirez *et al*., 2006; Tolke and Klink, 2006) was used for the UML design. Though there are many testing packages which include Apache JMeter, JUnit, Grinder, Siege, JProfiler, selenium, Tsung, and Load Runner; for request generation, load testing of Web Services application and capturing of response time which is vital to this research. We used Apache JMeter (http://jakarta.apache.org/) because it is open source, Java based and has rich and easy to use User Interface (UI). Besides, it is the testing tool also used by the originators of ROA (Ekuobase and Onibere, 2013). JMeter was added as plug-in to Netbeans to ease its use with IDE. Microsoft Excel was used for result computation. The Alentum

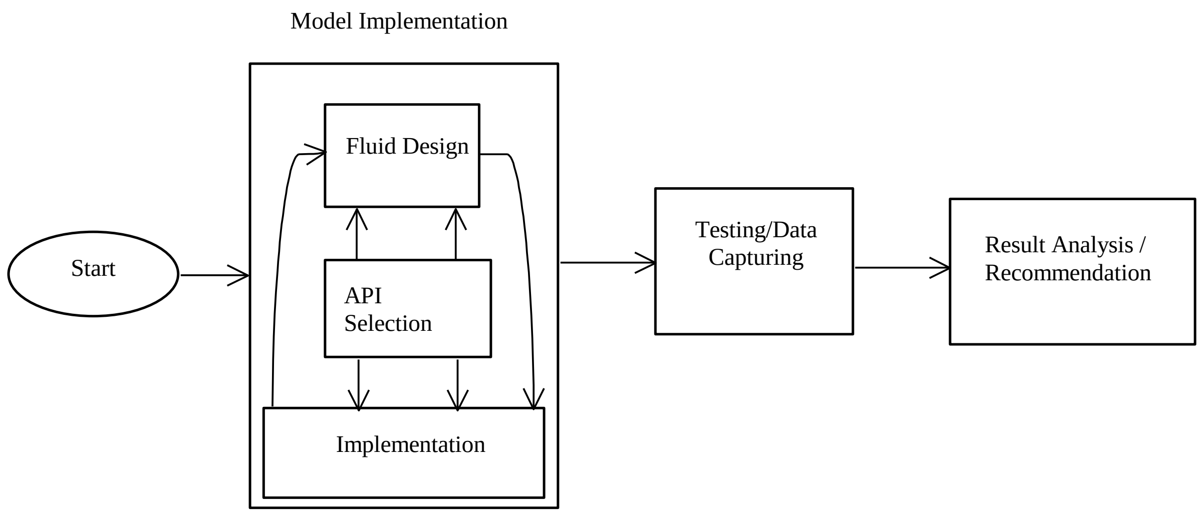

Fig.1. The Research Process.

Software Advanced Grapher was used for the graphical reporting of results.

Process

Figure 1 depicts a tailored systems analysis and design (SAD) research process used in this work.

We relied on the agile software development methodology established by Ekuobase and Onibere (2012, 2013) as the most appropriate for the project. Our test software development project of choice is the Automated Teller Machine (ATM) Fund Transfer System. This choice is predicate on the fears raised by Ekuobase and Onibere (2011) and the features of the ATM system also exposed by Ekuobase and Onibere (2012, 2013). The design of the ATM system as modelled by Ekuobase and Onibere (2012, 2013) was adopted. A Web Services solution was built using appropriate APIs of the Java platform using Netbeans 7.0 IDE. The solution consists of five endpoint replica built on ROA. We chose to implement only the five endpoint replica of ROA since it gave the optimum scalability result for our test problem – the ATM Fund Transfer Service (Ekuobase and Onibere, 2013). Besides, the five endpoint replica solution's computational strength data appears more regular (Ekuobase and Onibere, 2013). We also built a similar web services solution using the conventional approach void of ROA.

We, however used a different set of Java APIs for the implementation of the prototype ATM Fund Transfer system on ROA. The new set of implementation API's were Enterprise Java Bean (EJB), Java API for XML Web Services (JAX-WS), Java Message Service (JMS), Message Driven Bean (MDB) and Java Persistent API (JPA). These APIs were selected to design the implementation equivalence of ROA depicted in figure 2. The choice of these APIs is predicate on the drive not only to refine the implementation of ROA but also to demonstrate that ROA can be realised in several ways using diferrent technology and platforms. In particular, JPA a relatively new Java API handles how relational data is mapped to persistent entity objects, how these objects are stored in a relational database, and how an entity's state is persisted. In this realization, the JAX-WS receives a SOAP request, implicitly deserializes the request and, using the round robin mechanism, enqueues the resultant data in a JMS queue for some queues defined by the number of replicas; each replica has its own queue. A replica, implemented as MDB, listens and fetches the data in its JMS queue that it is statically bound to, using the MDB onMessage method. The replica then invokes the EJB that performs the business logic and with help of the JPA persist in memory (database) the computational state of the operation performed.

For the conventional implementation of the system i.e. implementation not based on ROA, we made use of JAX-WS, EJB and JPA as depicted in figure3. The EJB and JPA allow the web services to seamlessly communicate with the database. Observe that the selected APIs for the ROA implementation have JMS and MDB, in addition, that were used to realize the replicas in ROA. The use of JAX-WS for the ROA implementation as with the conventional implementation is particularly soothing because of the criticism that the ROA implementation equivalent used by Ekuobase and Onibere (2012, 2013) does not realize a Web Services solution but just a service solution. Besides, this ROA implementation is more coarsely grained than that of Ekuobase and Onibere (2012, 2013).

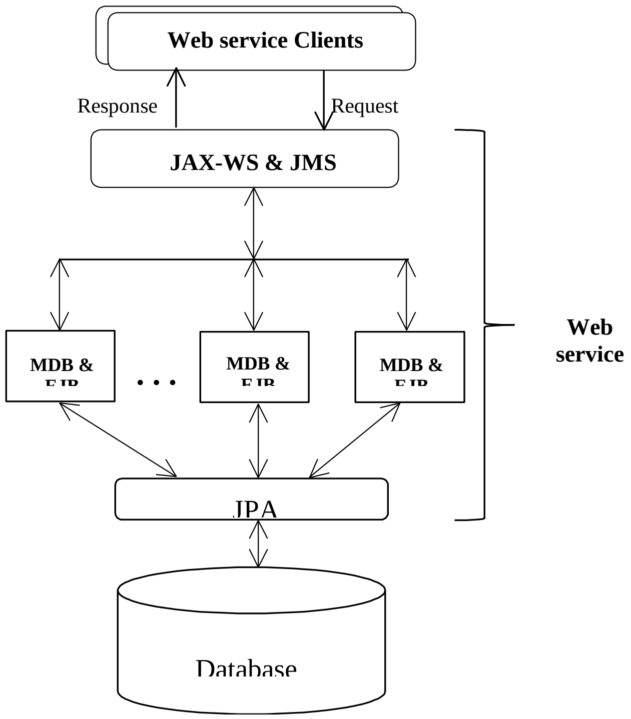

Fig. 2. A ROA Implementation Equivalence using Java Technology.

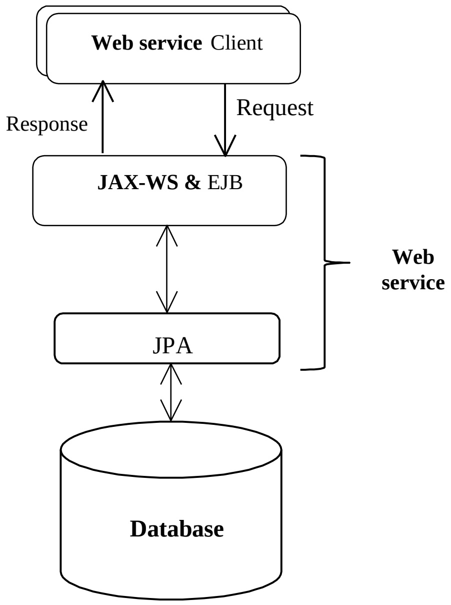

Fig. 3. A Conventional Implementation of Web Services using Java Technology.

We also made us of a small sized database built using ObjectDB an open source Object Database Management System (ODBMS) for about 30 account holders with a varying fictitious amounts in the accounts. The software development and deployment environment was Netbeans 7.0 with GlassFish 3.1 as the server.

Codes for the Conventional and ROA implementations will be supplied on request.

These systems are all server side applications and we therefore need a client to consume them. Apache JMeter played this role. Apache JMeter (Halili, 2008) is not only a load generator but a load and performance testing tool.

Table 1. Data Capture for Conventional Web Services Solution.

DATA CAPTURE FROM IMPLEMENTATION WITHOUT ROA					
NO OF SAMPLES	AVERAGE	MEDIAN	90TH PERCENTILE	MINIMUM	MAXIMUM
5	4	4	6	4	6
10	3	3	4	2	6
20	4	4	7	3	7
30	3	3	4	3	10
40	4	4	4	3	17
50	4	3	4	2	35
100	3	3	4	2	15
200	3	3	3	2	154
300	5	3	4	2	208
400	5	3	4	2	209
500	8	3	6	2	317
1000	27	4	16	2	2386
2000	42	14	92	2	2478
3000	291	321	514	2	4411
4000	563	569	944	3	5428
5000	791	790	1420	3	6299
10000	1930	1753	4230	3	10884

Table 2. Data Capture for the ROA Web Services Solution.

DATA CAPTURE FROM ROA IMPLEMENTATION					
NO OF SAMPLES	AVERAGE	MEDIAN	90TH PERCENTILE	MINIMUM	MAXIMUM
5	3	3	3	3	6
10	5	4	8	3	9
20	4	4	7	2	10
30	6	6	6	4	16
40	3	3	4	2	5
50	3	3	5	2	22
100	3	3	4	2	5
200	2	3	3	2	11
300	3	3	4	2	18
400	3	3	4	2	23
500	3	3	3	2	22
1000	3	3	4	2	68
2000	6	3	9	1	227
3000	8	4	19	1	163
4000	18	7	41	1	423
5000	30	9	80	1	577
10000	25	10	51	1	1045

It can handle variety of request from HTTP request to SOAP request depending on how its test plan is prepared.
We subjected the web services solutions to performance test under varying loads ranging from five to 10000 requests per 5 seconds using Apache JMeter. The resultant data samples' median, maximum, minimum, average and 90th percentile response times for each of the solutions were collected. We then entered this data into Alentum Grapher for appropriate graphical presentation. The maximum response times value were further

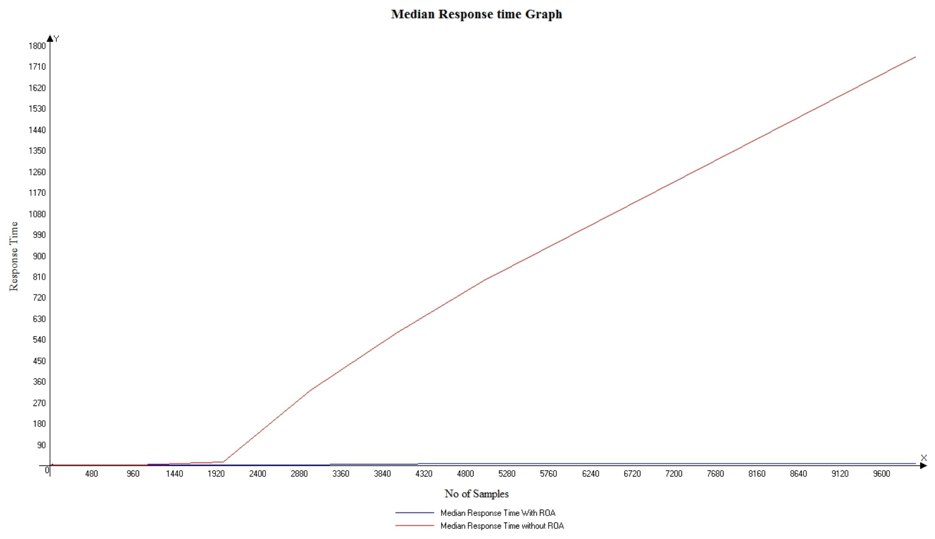

Fig. 4. The Mid-response Time of Conventional vs. ROA Web Services Solutions.

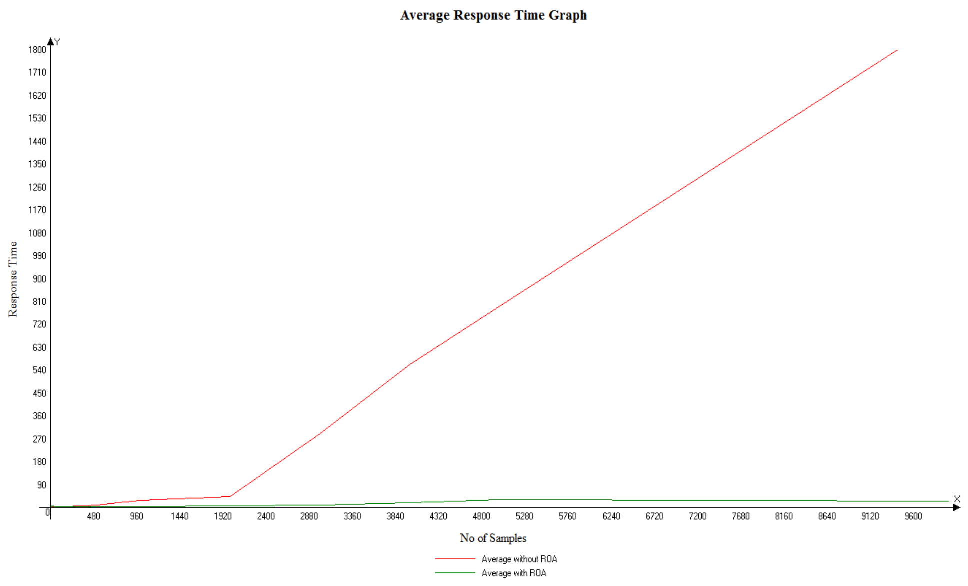

Fig. 5. The Average-response Time of Conventional vs. ROA Web Services Solutions.

subjected to statistical analysis and interpretation; since maximum response times best captures the highest tail latency of applications.

In particular, we ascertained the tail tolerance significance of the Web Services solution built on ROA over that built using the conventional approach. Since the two classes of Web Services solution were built on the same problem and platforms but with different development approaches, the student t-distribution for difference of two means was found most appropriate and adopted. The samples x and y are the maximum response times for the conventional and ROA solutions respectively. Let x and y be normally distributed with means μ_x and μ_y, and variance σ_x and σ_y respectively. The problem is to decide whether or not the use of ROA will mitigate the tail latency of Web Services solution.

Consequently, we tested the hypothesis H_0: $\mu_x = \mu_y$ (no tail tolerant significance between conventional and ROA systems), H_1: $\mu_x > \mu_y$ (ROA systems are significantly tail tolerant) and H2: $\mu_x < \mu_y$ (conventional system are significantly tail tolerant).

RESULTS AND DISCUSSION

After configuration the JMeter, we executed the package for varying number of threads (sample size) for each of

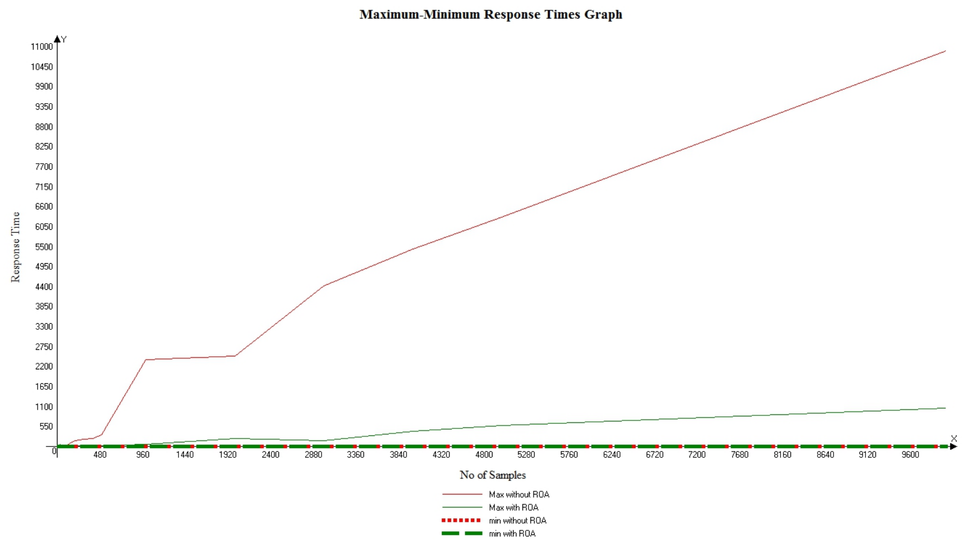

Fig. 6. The Maximum and Minimum Response Time of Conventional vs. ROA Web Services Solutions.

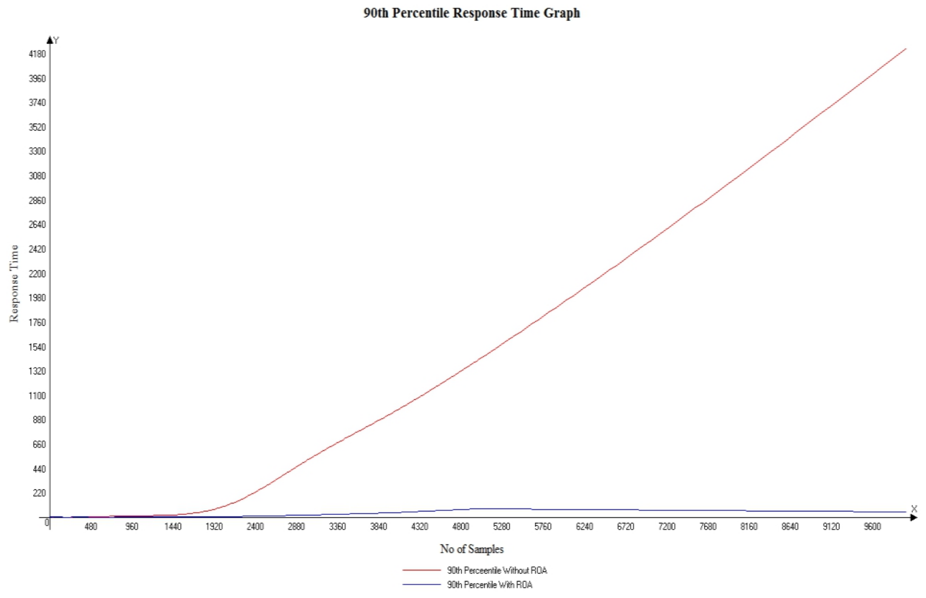

Fig. 7. The 90th Percentile Response Time of Conventional vs. ROA Web Services Solutions.

the two system implementations and the valuable data: average, median (mid), 90th percentile line, minimum and maximum response times; all in milliseconds were collected. Tables 1 and 2 contain these data for the conventional and ROA web services solution respectively.

For ease of appreciation, figures 4 to 8 depict graphically the relative behaviour of both applications with increasing number of request per unit time as explained underneath each of the figures.

Figure 4 captures the mid response times of both the conventional web services solution and those of the web services solution built on ROA with increasing number of request per unit time. Observe the near constant response time of the ROA solution even with increasing request per unit time as against that of the conventional solution which assumed a near exponential increase of response time with increasing request per unit time. The implication of this is that web services solutions built on ROA is far more stable and hence more scalable than its conventional counterparts. This affirms Ekuobase and Onibere (2013) scalability authentication of ROA. Also their maximum sample size was 2000 requests per 5 seconds against ours with a maximum sample size of 10000 requests per 5 seconds. Note that the behaviour of the conventional solution against the ROA solution

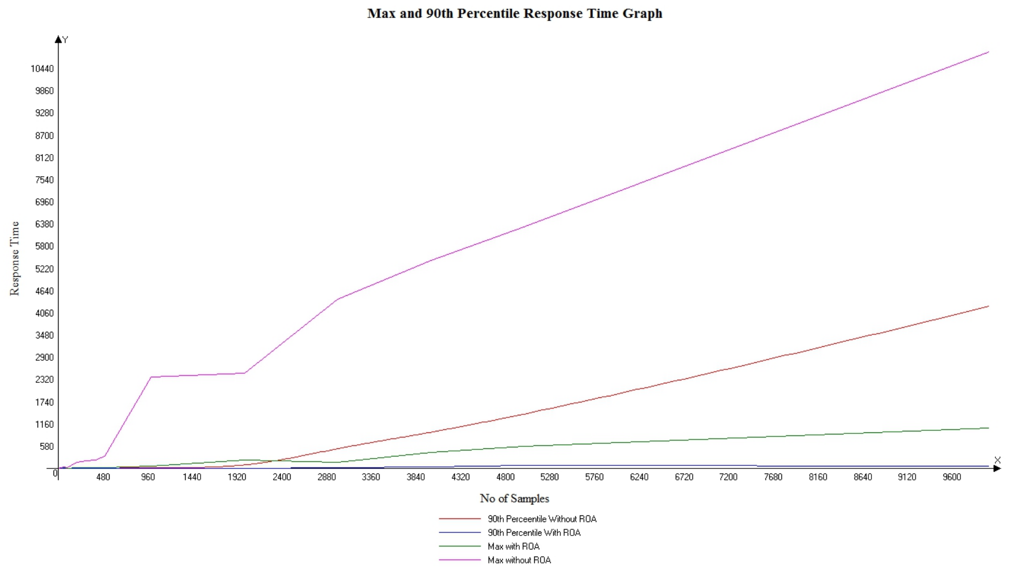

Fig. 8. The 90th Percentile and Maximum Response Time of Conventional vs. ROA Web Services Solutions

Table 3. Showing the computation of ns^2 for each web services solution.

S/n	Conventional Tail Latency (x)	$(x-\bar{x})^2$	ROA Tail Latency (y)	$(y-\bar{y})^2$
1	6	3715369.633	6	22464.71972
2	6	3715369.633	9	21574.42561
3	7	3711515.574	10	21281.6609
4	10	3699965.398	16	19567.07266
5	17	3673084.986	5	22765.48443
6	35	3604413.927	22	17924.48443
7	15	3680755.104	5	22765.48443
8	154	3166724.927	11	20990.89619
9	208	2977451.751	18	19011.54325
10	209	2974001.692	23	17657.71972
11	317	2613167.339	22	17924.48443
12	2386	204729.6332	68	7723.307958
13	2478	296448.2215	227	5057.719723
14	4411	6137860.516	163	50.66089965
15	5428	12211324.69	423	71351.83737
16	6299	19057333.46	577	177340.0727
17	10884	80110923.75	1045	790530.1903
	$\bar{x}$ =1933.529412	$n_x s_x^2$ =155550440.2	$\bar{y}$ =155.8823529	$n_y s_y^2$=1275981.765

skyrocketed particularly after the 2000 mark. An indication that the 32% scalability performance of web services solution built on ROA over the conventional solution is the least it could be. Mid response time is however not a useful indicator of latency variability or tail latency.

Figure 5 captures the average response time of the conventional and ROA web services solutions with increasing number of requests per unit time. It appears not to be different from the graph in figure 4 and therefore same analysis holds for both.

Figure 6 captures the maximum and minimum response time with increasing number of requests per unit time for both the conventional and ROA web services solutions. Note that the difference between the maximum and minimum for the conventional web services solution is far wider than that of the web services solution built on ROA. In particular, the response time for the ROA solution hardly exceeded 0.5seconds. This shows that web services solution built on ROA has tighter latency variability or guaranteed responsiveness against their conventional counterparts.

Figure 7 captures the 90th percentile response time of conventional vs. ROA web services solutions with increasing number of request per unit time. It appears not to be different from the graph in figures 4 and 5 and therefore same analysis holds here too. Besides, the graph also shows that web services solution built on ROA has tighter latency variability or guaranteed responsiveness against their conventional counterparts.

Figure 8 captures the 90th percentile and maximum response times of conventional vs. ROA web services solutions with increasing number of request per unit time. It obviously indicates that both applications are bedeviled with the problem of tail latency with the ROA solution however more tail tolerant. Whether this tail tolerant advantage of ROA web services solution over its conventional counterpart is significant and if it is, by what degree is however not obvious?

Statistical Analysis and Interpretation

It will be noted that the two sets of Web Services solutions were built on the same platform, using the same technology except that the ROA's solution was built on unique development architecture – ROA. It is also important to note that the performance load test and request generator was handled by the same package – Apache JMeter, and configured the same way.

The samples x and y are the maximum response time (tail latency) at varying but increasing request rates for the conventional and ROA solutions respectively. Let x and y be normally distributed with means μ_x and μ_y and variance σ_x and σ_y respectively. The problem is to decide whether or not the use of ROA will improve the tail tolerance of Web Services solution. Consequently, we tested the hypothesis H_0: $\mu_x = \mu_y$ (no tail tolerant significance between conventional and ROA systems), H_1: $\mu_x > \mu_y$ (ROA systems are significantly tail tolerant) and H2: $\mu_x < \mu_y$ (conventional system are significantly tail tolerant); since tail tolerance is about mitigating tail latency with increasing request per unit time.

It is safe to assume that $\sigma_x = \sigma_y$, and then apply the formula below (Hoel, 1966):

$$t = \frac{(\bar{x}-\bar{y})-(\mu_x-\mu_y)}{\sqrt{n_x s_x^2+n_y s_y^2}} \sqrt{\frac{n_x n_y (n_x+n_y-2)}{n_x+n_y}} \qquad (2)$$

Where, t is the student's t distribution for difference of two means and every other elements of equation (2) assume the conventional statistical use.

In our case, sample sizes are equal and equal to 17 i.e. $n_x = n_y = 17$; therefore equation (2) can be rewritten as:

$$t = \frac{(\bar{x}-\bar{y})-(\mu_x-\mu_y)}{\sqrt{n_x s_x^2+n_y s_y^2}} \sqrt{272} \qquad (3)$$

Adopting the null hypothesis H_0, we can rewrite equation (3) as equation (4) below:

$$t = \frac{(\bar{x}-\bar{y})}{\sqrt{n_x s_x^2+n_y s_y^2}} \sqrt{272} \qquad (4)$$

but, $ns^2 = \sum_{i=1}^{n}(x-\bar{x})^2$ (5)

Table 3 shows the computation of ns^2 for each solution: conventional solution followed by solutions built on ROA while table 4 shows the calculation details of t, for each Web Services solution.

From the student's t table (Hoel, 1966) the 0.02 critical value of t is 2.224. Observe that the calculated t value (2.3411022) is greater than 2.224, therefore, the hypothesis H_1: $\mu_x > \mu_y$ is valid and hereby accepted. Thus, tail tolerance of Web Services solution built on ROA is significantly better than its equivalent conventional solution.

Table 4. Showing the Computed t Value and Confidence Limit for Conventional vs. ROA's Solution.

Computed Value	**ROA web services solution**
$\bar{x}-\bar{y}$	1777.6471
$n_x s_x^2 + n_y s_y^2$	156826422
$\sqrt{n_x s_x^2 + n_y s_y^2}$	12523.036
$(\bar{x}-\bar{y})/\sqrt{n_x s_x^2 + n_y s_y^2}$	0.1419502
t	**2.3411022**
k = $(\sqrt{n_x s_x^2 + n_y s_y^2})/\sqrt{272}$	759.3206
2.224 * k	1688.729
α	88.91811
β	3466.376
$(\alpha/\bar{x}) * 100$	**4.5987**

It is also important we calculate the confidence limit for $\mu_x - \mu_y$ as ROA's solution show significantly better tail tolerance performance over its equivalent conventional solution. Here equation (3) comes handy and in our case 96 percent confidence limits is given by:

$|t| < 2.224$ (6)

Substituting (3) in (6), reduces (6) to:

$$\alpha < \mu_x - \mu_y < \beta \qquad (7)$$

Where α and β are the lower and upper limits respectively and are given by:

$$\alpha = (\bar{x} - \bar{y}) - 2.224\left(\frac{\sqrt{n_x s_x^2 + n_y s_y^2}}{\sqrt{272}}\right) \qquad (8)$$

$$\beta = (\bar{x} - \bar{y}) + 2.224\left(\frac{\sqrt{n_x s_x^2 + n_y s_y^2}}{\sqrt{272}}\right) \qquad (9)$$

Table 4 also show these calculated value for the ROA's solution.

Consequently, we can only guarantee α unit of increase i.e. $\alpha/\bar{x}$ percent in tail tolerance performance, if ROA is used to build Web Services solution. These results show that ROA can improve tail tolerance of Web Services solution by 4.60% with 96% confidence. This is the maximum degree of significance that can be guaranteed.

CONCLUSION

Guaranteed responsiveness of Web Services solutions may not be possible on a large scale, if the solutions are not tail tolerant i.e. able to consistently keep latency within reasonable limit. Software techniques that tolerate latency variability and in particular, tail latency are vital to building responsive large-scale Web services (Dean and Barroso, 2013). However, such efforts directed at making network applications tail tolerant are basically oriented towards handling latency at the systems or deployment level and not at the application or development level. ROA though proposed to help application programmers build scalable Web Services solutions (Ekuobase and Onibere, 2011), appears capable of mitigating latency variability and tail latency at the application or development level. Consequently, we investigated ROA for tail tolerance.

We realized a new ROA implementation equivalence also using Java technology but with a different set of Java APIs from that of Ekuobase and Onibere (2012, 2013). This implementation equivalence appears more of a Web Service implementation than theirs. We built two ATM Web Services solution using Java technology – the first was not built on ROA (conventional solution) but the other was built on ROA (ROA solution). The choice of the ATM system as the test problem was to investigate ROA in its worst case scenario (Ekuobase and Onibere, 2011, 2012, 2013).

The graphical and statistical analysis of the resultant data on subjecting the Web Services solution built on ROA to load performance test using Apache JMeter compared to the conventional solution showed that the tail tolerance of Web Services solution built on ROA is significantly better than its equivalent conventional solution. Besides, the statistical analysis of the results shows that ROA is capable of improving the tail tolerance of Web Services solution by about 4.60% with 96% confidence. The results also affirm the scalability capability of ROA (Ekuobase and Onibere, 2013).

REFERENCES

Baldoni, R., Marchetti, C. and Termini, A. 2002. Active Software Replication through a Three-tier Approach. Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS). IEEE Computer. pp10.

Birman, K. 2005. Can Web Services Scale Up? IEEE Computer. 38(10): 107-110.

Coulouris, G., Dollimore, J., Kindberg, T. and Blair, G. 2012. Distributed System: Concepts and Design. Addison-Wesley, USA. pp1047.

Dean, J. and Barroso, LA. 2013. The Tail at Scale. Communications of the ACM. 56(2):74-80.

Ekuobase, GO. and Ebietomere, EP. 2012. The Making of Replication Oriented Architecture. Journal of Computer Science. 23(2):71-78.

Ekuobase, G. and Onibere, E. 2011. Architecture For Scalable Web Services Solution. Canadian Journal of Pure and Applied Sciences. 5(1):1449-1453.

Ekuobase, GO. and Onibere, EA. 2012. Web Services Solution on Replication Oriented Architecture (ROA). Journal of the Ghana Science Association. 14(2):25-34.

Ekuobase, GO. and Onibere, EA. 2013. Scalability of Web Services Solution Built on ROA. Canadian Journal of Pure and Applied Sciences. 7(1):2251-2270.

Halili, E. 2008. Apache JMeter: a practical beginner's guide automated testing and performance measurement for your websites. Packt Publishing, United Kingdom. pp129.

Hoel, PG. 1966. Introduction to Mathematical Statistics, (3rd edi.). John Willey & Sons, USA. pp427.

Jendrock, E., Ball, J., Carson, D., Evans, I., Fordin, S. and Haase, K. 2006. The Java EE 5 Tutorial (3rd edi.). Sun Microsystems, USA. pp1304.

Ramirez, A., Vanpeperstraete, P., Rueckert, A., Odutola, K., Bennett, J., Tolke, L. and van der Wulp, M. 2006. Argo UML User Manual: A Tutorial and Reference Description. Open Content, USA. Also available online at http://www.opencontent.org/openpub/. pp386.

Repp, N., Berbner, R., Heckmann, O. and Steinmetz, R. 2007. A cross-Layer Approach to Performance

Monitoring of Web Services. Emerging Web Services Technology. Birkhauser Verlag, Switzerland. Whitestern Series:21-32.

Tolke, L. and Klink, M. 2006. Cook book for Developers of ArgoUML: An Introduction to Developing ArgoUML. University of California, USA. pp157.

Vawter, C. and Roman, E. 2001. J2EE vs. Microsoft.NET: A Comparison of Building XML-based Web Services. Sun Microsystems, USA. pp28.

Williams, J. 2003. The Web Services Debate: J2EE vs. NET. Communication of the ACM. 46(6):59-63.

Yang, S. J., Zhang, J. and lan, BC. 2006. Service Level Agreement-Based QoS Analysis for Web Services Discovery and Composition. International Journal of Internet and Enterprise Management. Inderscience. 1251-1271.

Received: Feb 15, 2014; Accepted: April 3, 2014