iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 4 **Issue:** II **Month of publication:** February 2016

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

Volume 4 Issue II, February 2016
ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology (IJRASET)

A Novel Approach for Transaction Management in Heterogeneous Distributed Database Systems

Mr. Dheeraj Bhimrao Lokhande[1], Prof.Dr.R.C.Thool[2]
[1]*Research Scholar, Walchand Institute of Technology, Solapur University, Solapur, Maharashtra, India*
[2]*Shri Guru Govid Singhji, College of Engineering, Nanded, Maharashtra, India*

Abstract: ***RESTful APIs are widely adopted in designing components that are combined to form web information systems. The use of REST is growing with the inclusion of smart devices and the Internet of Things, within the scope of web information systems, along with large-scale distributed NoSQL data stores and other web-based and cloud-hosted services. There is an important subclass of web information systems and distributed applications which would benefit from stronger transactional support, as typically found in traditional enterprise systems. In this paper, we propose REST with Transactions, a transactional RESTful data access protocol and API that extends HTTP to provide multi-item transactional access to data and state information across heterogeneous systems. We describe a case study called Tora, where we provide access through REST+T to an existing key-value store (Wired Tiger) that was intended for embedded operation.***
Keywords: Transaction Management, Heterogeneous cloud databases, REST API.

I. INTRODUCTION

The Hypertext Transfer Protocol (HTTP) was originally designed to access static content Hypertext Markup Language (HTML) content over a network. It soon gained popularity as a general purpose data access and communication protocol. Today it is applied for streaming voice and video and in application data communication frameworks. Fielding's REST [15] took advantage of the popularity of HTTP, and showed how a wide variety of services could use it as a primary data communication protocol to benefit from its simplicity, ease of composition, and robustness – well recognised features of the Web. As a result, services in a variety of application domains offer access through RESTful APIs, many replacing proprietary APIs. This has fueled the Web 2.0 phenomenon triggering hybrid applications and mash-ups that compose services and data from heterogeneous systems. "The Internet of Things" include smart devices and widgets with network and wireless connectivity like household controls and monitors including the heating system, lights, and motion detectors. These devices allow sensing or changing the physical environment by allowing applications to interact with them via a simple HTTP interface. Similarly, the internet-scale NoSQL data stores that are popular for storing data in large distributed or cloud-hosted systems, are accessed through a RESTful API over HTTP. When composing these services, devices, and stores into complex applications, there is often a need to address the issues of ensuring consistency of activity, preventing interference between concurrent actions, and so on, that are solved in traditional enterprise information systems through transactions. For example, an application may wish to adjust together the lighting and the television in a house, or change the telecommunications plan and the banking instructions in a single action. While some of the NoSQL stores support single item consistent access, it is rare for RESTful components to offer multi-item transactions, and even harder to build transactions that include accesses to multiple services. Solving this challenge by supporting general transactions that involve items and components, is the focus of our paper, which proposes a protocol and extension of HTTP called REST+T. There are some alternative approaches that have been offered before, for transaction support in web information systems. When systems are build from component services that follow the Web Services standards, there is a specification WSTransaction [23], [30] that assists in transactional behaviour. However, like all the WS-_ specifications, this requires a lot of extra effort (and often, bloat in message size and processing costs) to fit into a whole ecosystem of standards and tools such as SOAP. The rise in REST-based systems in place of WS-based ones, suggests that transactional support within the REST approach is worthwhile. For cases where one needs concurrent authoring of documents within a single service, WebDAV [18] is a reasonable approach that extends HTTP (and variants like CalDAV deal with other specific data types). However, WebDAV uses a locking protocol that limits scalability, and it does not target situations where transactions must access data that is stored across multiple components.

II. BACKGROUND

In this age of online shopping, the web-application is the critical component that ensures that the seller charges for the correct items bought and the buyer receives what was promised. Behind the web-server most systems are backed by some form of a transactional database or data store the enables the seller to make the guarantees the buyer seeks. Without this, the seller would soon loose credibility causing his business to fail. HTTP [14] has emerged as the protocol of choice for making information available over the internet. Primarily in the form of HTML documents, this has gradually evolved into dynamic, application and system state information. For example, Wi-Fi network modems seen in nearly every household worldwide provide an HTTP interface to a web-based management console. This is implemented by a bare-bones HTTP server that enables any browser to manipulate settings on the router using the web-based UI . REST [15] has made it possible to implement services that enable applications to use HTTP as a means of communication over the network. This is done by allowing the application to communicate with the service using Remote Procedure Calls (RPC) implemented using HTTP methods. Leaving each REST end-point to maintain its own state is the essence of the simplicity of the architecture making it highly scalable and robust. Competing technologies like SOAP, on the other hand, have been less successful due to the inherent session oriented approach. Transactional updates over HTTP have so far been implemented using protocols like WebDAV [18] and its extensions like CalDAV [8]. These systems use a locking protocol that depends on an application session in the form of leases. In addition, due to the pessimistic locking protocol, there is an underlying expectation that the entire collection of documents (or objects) being accessed reside in one homogeneous system. This design is monolithic in nature and not suitable for scaleout applications.

A. Transaction Models

Wrapping operations on data within transactions comes at a significant performance cost. Over the years, industry and researchers alike, have sought techniques to reduce this cost and improve the throughout of database systems to allow applications, both web-based and others, to serve the customer better. Often many of these operations could span many minutes, or even hours before being completed. Any transaction would have to hold onto locks on individual records for the period with a significant impact on performance and concurrency. To handle this, a notion of Long Lived Transactions (LLTs) that could be broken up into Sagas were introduced [17]. This allowed each individual component of the Saga to be interleaved with other short running transactions to improve concurrency and throughput. In particular, web-based applications can be modeled as workflows and has been shown to be better handled by Workflow Management Systems (WFMS) than traditional transaction models [3]. These systems can be better modeled using Sagas and Flexible transactions [13]. Further, Transactional Intent built into an application framework [16] has been used to describe and implement an apology-oriented computing framework [21] and eventual consistency [28] based on ACID 2.0 [22] properties. Numerous other approaches to handling web-based application failure have also been described [20].

B. Transactions over HTTP

The REST interface to the graph database, Neo4j [2], allows the client to maintain a session with the server to perform transactional updates. Explicit API calls to start, update, commit and abort transactions is used by the client. This approach does not support heterogeneity of data stores and is suitable for accessing a single data store instance only. Numerous large distributed key-value stores implement transactions. For instance, Google Megastore [4] provides the ability to group multiple objects into static entity groups in order to commit them using a single call. A similar concept is employed by G-Store [9] which allows individual records to be included in an entity group. Entities can be migrated from one group to another using a migration protocol. JEST [25], a REST interface to OpenJPA, uses a notion of fetch groups to allow the application to define object closures in a traditional database that can be modified by the application and then updated in a single PUT or POST operation.

Large-scale, cloud-based, distributed key-value storage services like Windows Azure Storage (WAS) [5] and Google Cloud Storage (GCS) [1] support single item consistent updates and single item read-on-write consistency. This is enabled using conditional operations using If-None-Match headers that uses the object ETag. WAS returns an ETag that uniquely identifies the record version. GCS returns a generation number which is essentially a nanosecond granularity timestamp of the object.

C. State and state change

Traditionally, applications have used databases to store application data state. For instances, a banking application may maintain the state of each bank account in the form of its current balance and the last 10 transactions in a database. Similarly, the REST end-points used by an applications that use RESTful web services stores application state. As the application goes about performing its designated tasks, it interacts with the REST end-points to update the state with appropriate requests. While many operations may

involve interacting with a single end-point, there are situations when more than one end-point is updated, i.e. multiple individual state changes, as part single application-defined operation. An simple example of this is the deduction of a sum from one bank account and addition of the same amount to another bank account as part of a funds transfer from one account to the other. Data items stored in key-value stores are examples of endpoints that keep application state. Throughout the remaining document, we use the term data item interchangeably with end-points. It is crucial for many application to ensure that the changes to state across multiple data items occur in a transactional manner.

This is particularly challenging when there are multiple concurrent applications attempting to make simultaneous updates to the same set of data items. Transactions are a suitable programming abstraction that enables application developers to ensure that the applications do not step on each others toes while executing concurrently.

D. Consistency

Some systems [1], [5] provide the ability to enable the application to manage consistent change from one state to another when there is only one data item involved. This is made possible by providing a form of test-and-set operation using the items ETags or equivalent version identifier and a conditional update request. When more than one state change occurs as a result of a set of operations on multiple end-points, there is a need to perform these operations within bounds of a transaction. In order to maintain validity of the changes to the data items individually and as a whole, these transactions must exhibit the ACID [19] properties of transactions. This is an even more important when there are multiple clients that simultaneously access a common set of data items or end-points.

E. Transactions across multi-items

There are various techniques developed to ensure transactions across multiple data items. One approach has been to implement transaction support in the service itself. To allow such a capability, WebDAV [18] provides a pessimistic locking protocol that allows multiple users to simultaneously edit a common set of files. This approach is effective in homogeneous environments with one set of authoring tools due to the underlying assumptions in the protocol. It is also suitable for implementation in a single host or small cluster. For instance, the technique is successfully extended in CalDAV [8] to provide define a calendar access, management, sharing and scheduling based on the iCalendar format. Similarly, NewSQL systems, distributed database systems with SQL interfaces, like Google Spanner [7] have implemented distributed transactions in very large, globally distributed cluster of machines using a transaction management service built into the data store. An alternate approach is to implement the transaction management in middleware which coordinates the transaction within a centrally managed system that resides between the client application and the data store or REST end-point. Systems like G-Store [9], MegaStore [4] and CloudTPS [31].

A third approach involves using the client to perform the transaction coordination task using transaction state stored within the data item and a access protocol that is based on the test-and-set operation to perform consistent updated across multiple items. This approach is described in systems like Google Percolator [24] and Cherry Garcia [11], [12].

F. REST with Transaction support

The REST way of implementing services provides the ability to keep state at the end-point and not at the client. This makes it possible for multiple application instances to simultaneously access the data item without compromising consistency. This is easily done using a test-and-set operation. Database systems have used the 2-phase commit protocol (2PC) [26] effectively in production systems for many years now.

We use this time-tested approach as a basis of our extensions to REST and optimize it remove some of its drawbacks. In essence, we treat each data item as a single record database and perform a coordinated transaction commit across multiple data items. To do this we extend the HTTP API with four new methods: PREPARE, COMMIT, ABORT and RECOVER. Using these extended HTTP methods, additional HTTP headers and test-and-set operations, we are able to support non-blocking, optimistic transactions with snapshot isolation semantics.

Volume 4 Issue II, February 2016
ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology (IJRASET)

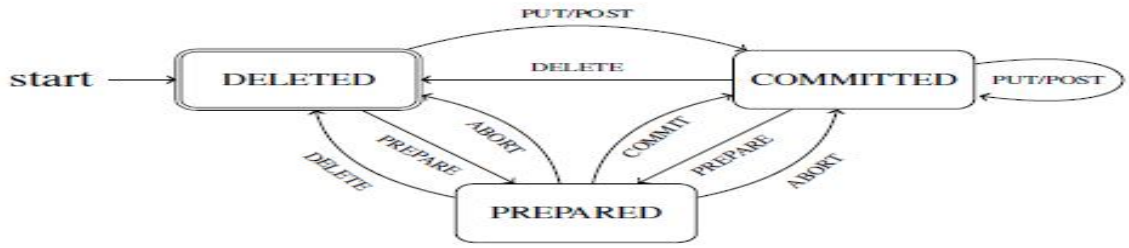


Fig. 1. REST+T object state transition diagram

III. THE REST+T PROPOSAL

In this section we define the REST+T protocol and the extensions to the set of standard HTTP protocol methods. Figure 1 describes the state transitions that occur during the process of committing a transaction on each data items as part of a multi-item transaction.

A. *REST+T data header meta-data*

In addition to the standard HTTP headers passed with each HTTP request and corresponding response, REST+T defines additional data state headers that are stored along with the data.

1) *ETag:* Every data item has a version tag associated with it. This is unique for each version of the object and can be implemented as a timestamp or a unique hash code for the object. This tag can be used to perform test-and-set operations on the data store using the If-Match HTTP request header.
2) *Transaction-Id:* This specifies a URI that can be used to identify the application defined transaction that updated the version object. The URI structure and content is application defined and is not mandated by the REST+T protocol. The intention of this header field is to enable other concurrent and subsequent application instances to determine the state of the transaction that created the version of the item.
3) *Transaction-State:* The specifies the state of the data item. It can be PREPARED, COMMITTED, and DELETED. An object is readable when it is in the COMMITTED state. An object in the PREPARED state must be recovered and then rolled back or rolled forward and committed.
4) **Valid-Time-Start:-** This is the timestamp from when the version of the data item is readable by a client. This is set by the PREPARE method.
5) **Valid-Time-End:**- This is the timestamp after which the version of the data item is not readable by any client. This is set by the DELETE method which is then followed by the COMMIT method.
6) **Lease-Time:-** This header specifies the period after the timestamp specified by the Valid-Time-Start header when the data item must be recovered and either rolled back to the previous COMMITTED state or rolled forward and set to a COMMITTED state from the PREPARED state.

B. *REST+T methods*

This section defines the methods on the data items:-

1) *GET:* The GET method returns the data item along with header information in the form of an ETag the uniquely identifies the version of the data item along with the standard HTTP headers. The actual contents of the data can be application or service specific and REST+T does not impose any restrictions on it. For all practical purposes, this method is identical to the standard HTTP GET method. In order to indicate whether the object returned is a valid version, the *Transaction-State* header is returned along with the response to indicate whether the object has been successfully committed to the data store. If the *Transaction-State* header is set to COMMITTED the object is consistently written. If set to PREPARED, it is not properly committed and needs to be recovered and either rolled back or forward depending on whether the transaction was successfully committed or not.
2) *PUT/POST:* The PUT and POST methods behave exactly as defined by the HTTP protocol. However, to maintain data

consistency, an application should not, in general, use these methods to update data; instead the PREPARE method must be used as described below. In the case where a transaction updates only a single item, the PUT or POST methods can be used, writing back with a conditional update header to ensure that the changes are done only if the item's ETag is unchanged.This is equivalent to the one-phase optimization in 2PC.

3) *PREPARE:* When there are more than one items to be updated, each item is first written to its respective data store using the PREPARE method and a conditional update header. If successful, the data item is moved to the PREPARED state and subsequent PUT, POST or PREPARE operations are permitted on the object. The application instance that prepared the data item is identified by the *Transaction-Id* HTTP header. This transaction identifier is in the form of URI that identifies the transaction. To ensure that an object does not remain in the PREPARED state indefinitely, the PREPARE method takes a *Lease-Timeout* header. This ensures that the upper bound on the time an object remains in the PREPARED state. GET operations on an object that is in the PREPARED state returns the last written version of the object when the lease time has not expired. If the lease time expires, the prepared version of the item is returned with the *Transaction-State* set to PREPARED. This is used by the client to recover the transaction based on the state of the transaction identified by the *Transaction-Id* herder. In addition, the *PREPARE* operation sets the object *Valid- Time-Start* header to indicate the timestamp from when the transaction can be considered to be committed. A *Valid-Time- End* header is used to indicate the time when the data item is no longer valid. If the object is already in the PREPARED state, the operation fails with the HTTP error code 412 (Precondition failed).
4) *COMMIT:* Once all the objects have been prepared to their respective data stores, their changes can be made permanent using the *COMMIT* method. Only, previously prepared objects can be committed using this method.
5) *ABORT:* If a client is not able to prepare all items involved in the transaction, it must call the *ABORT* method on the already prepared items in order to rollback the changes. This makes the item available for updates to other clients. A version tag and a conditional update header is used for this operation to ensure that *ABORT* operation is performed on the the version of the object intended by the client.
6) *RECOVER:* If a *PREPARE* operation fails because the object is already in the PREPARED state with error code 412, the client may choose to inspect the version of the object in PREPARED state to see if it needs to be recovered issuing a *RECOVER* request. This is used to perform lazy transaction recovery in the situation when a client dies without completing the *COMMIT* phase. The URI set in the *Transaction-Id* header is used to discover the fate of the transaction. If the URI points to a valid end-point with the *Transaction-State* header set to COMMITTED, the transaction must be rolled forward by calling the *COMMIT* method on the item, else it does not exist or if the *Transaction-State* header is set to DELETED then the *ABORT* operation must be issued.
7) *PREV:* If a GET operation returns an object whose *Valid- Time-Start* is later than the start time of the application transaction, it should read a previous version of the object. The PREV method is used to do this. A a conditional HTTP header is used using the object version tag to fetch a valid version of the data item with respect to the version previously read.
8) *DELETE:* A *DELETE* call removes a data item from the data store. Unlike the typical implementation of *DELETE* in standard HTTP, this operation performs a logical delete by setting the Valid-Time-End to the transaction commitment time and setting the item *Transaction-State* header to DELETED. Deleted items are garbage collected after the timeout set using the Delete-Timeout header expires.

IV. A CASE STUDY

In this section we illustrate the REST+T approach by describing the implementation of Tora1, a high-performance distributed key-value store that provides a REST+T interface. Tora has three primary components, the REST+T interface, the transactional data storage system and the distribution and fault tolerance layer. Here, we focus on the REST+T interface. The transactional data store is an existing system called Wired Tiger. The distribution and fault-tolerance of the storage system is beyond the scope of this paper.

A. *REST+T interface*

The REST+T implementation requires just 2407 lines of C++ code in 16 source files using the Boost library. This included all the REST+T request and response handling and network I/O. The current version does not support any security measures to perform authentication and authorization of request. However, with a bit of code restructuring, it should be fairly easy to plug in any third-party mechanism. The implementation obtains the system time at microsecond granularity using the *gettimeofday(2)*function. The obtained timestamp is used to generate the ETag and Last-Modified response headers.

Volume 4 Issue II, February 2016
ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology (IJRASET)

B. Storage layer

We use Wired Tiger [29] to store the data items on disk. Wired Tiger is a high-performance key-value store providing transactional access to data stored on disk with support for snap-shot isolation semantics. It uses a log-structured storage layout taking advantage of lock-free data structures to enable high transaction rates even with higher degrees of concurrency.

V. EVALUATION

In this section, we evaluate the performance and correctness of the REST+T protocol. Further, we measure the correctness using a simple micro benchmark application that performs simple operations over standard HTTP using test-and-set operations versus an implementation that uses the REST+T protocol. Performance evaluation of our implementation of the REST+T API is done using a cluster of Tora servers and YCSB+T clients in a cluster of machines deployed using Amazon EC2.

A. Evaluating correctness

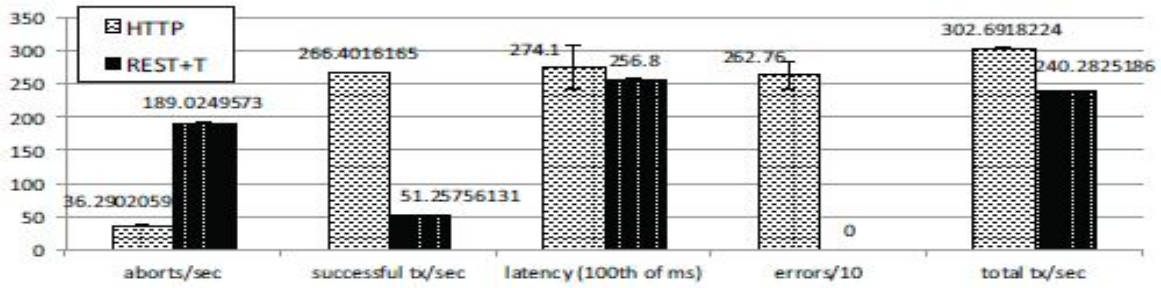


Fig. 2. Correctness behavior of HTTP using If-Match vs. REST+T

We implement a simple application that reads 2 records which simulate 2 bank accounts. Initially each account has a fixed sum of $100000 in them. The application reads one randomly selected record from the two and subtracts an amount of $10 from it and then reads the other account record and adds the amount of $10 to it. Both records are then written back to the data store in the order they were read. We ran our benchmark application with two threads and a Tora server on a single laptop running Mac OS 10.9.2 with 8GB 1600 MHz DDR3 RAM, 128GB SSD, and a dual-core 1.8GHz Intel Core i5. The application is run 5 times with 10000 transactions on each application thread. We measure the number of successfully committed transactions versus the number of aborted transactions achieved per second. We also measure the throughput and number of anomalies detected after the applications exit. If no inconsistencies are introduced, the net sum of both records will be 200000. Each error will create a difference of 0, 10 or 20 in the sum. We use this to measure the approximate number of errors introduced.

The graph in Figure 2 shows the results of this experiment. Despite using the test-and-set PUT calls using the If-Match header and the ETag, the HTTP approach was not able to detect all erroneous PUTs. The number of errors detected after the applications averaged 2627.6 for the HTTP application and 0 for the application using REST+T as expected. As a result, the number of aborts using HTTP averaged 36.29 aborted transactions per second versus an average of 189.02 per second using REST+T. The successful transaction using HTTP were 266.40 per second against 51.25 for REST+T. The total number transactions run against the Tora servers were 302.69 per second for HTTP versus 240.28 per second for REST+T highlighting the overhead of the REST+T protocol of approximately 25%, which is a small cost compared to the cost of the number of anomalies detected when using HTTP.

B. YCSB+T transactional NoSQL benchmark

In our tests, we use YCSB+T [10], an extension of YCSB [6], to evaluate our Java library implementation against Tora. YCSB+T provides an ability to group data store operations into transactions like traditional database benchmarks like the TPC-C [27] that are designed to measure the transaction performance of RDBMS implementations while maintaining the flexibility of the cloud services benchmark to measure performance. It measures the transaction throughput of web services in a web-based application context while ensuring data consistency. We use a YCSB+T workload with the Closed Economy Workload (CEW) [10] for this experiment. It simulates a closed economy with a fixed set of 10,000 accounts which add up to a fixed initial total of $1,000,000. The workload

consists of read operations reads one account record and read-modify write operation involves reading 2 records from a data store each and subtracts $1 from one account and adds it to the balance of the other. If there are inconsistencies, they can be measured by the difference between the expected total balance and the actual total balance. We thus present a Simple Anomaly Score (SAS) to measure the degree to which ACID transaction semantics are violated.

C. Measuring scaleout performance using YCSB+T

In this experiment, we ran 1 through 8 instance of YCSB+T client on 1 through 8 m1-xlarge EC2 hosts with 8 cores 15.1*GB* RAM standard 10*GB* EBS volume against 4 Tora data stores server with the same configuration. The OS installed is RedHat/CentOS Linux running Java JDK 7. To make our evaluations more realistic, we did not take advantage of any special network optimizations made possible by AWS in recent times. The graph in figure 3 shows that it scales linearly across the 4-node Tora cluster as we add 1, 2, 3, 4, up to 8 client nodes reaching a peak transaction throughput of 23,288 transactions/sec.

VI. CONCLUSIONS AND FUTURE WORK

It is becoming increasingly challenging to built effective web information systems and distributed applications, as the variety of the components composed within them increases. For example, there is a rapid rise in capabilities of smart devices and mobile phones, from new storage technologies, faster networks, and multi-core processors. REST has been very successful in enabling the ability to compose services in order to make complex web systems. However, newer applications will require multi-item transactions to keep data consistent across components. This makes it essential to offer transaction support within the RESTful approach built on HTTP. In this paper, we have described the REST+T protocol and API, an extension of the HTTP protocol. We have shown that it can be used to enable transactions across multiple items in an elegant and scalable manner without compromising REST's characteristics of interoperability and heterogeneity. We have illustrated and evaluated our approach using the Tora key value store. In other work [11], [12], we have shown that Tora can interoperate with other stores supporting REST APIs, such as Windows Azure Storage and Google Cloud Storage. In future work, we will apply this approach to the "Internet of Things" and smart device domains and foresee wide application of this protocol in this space.

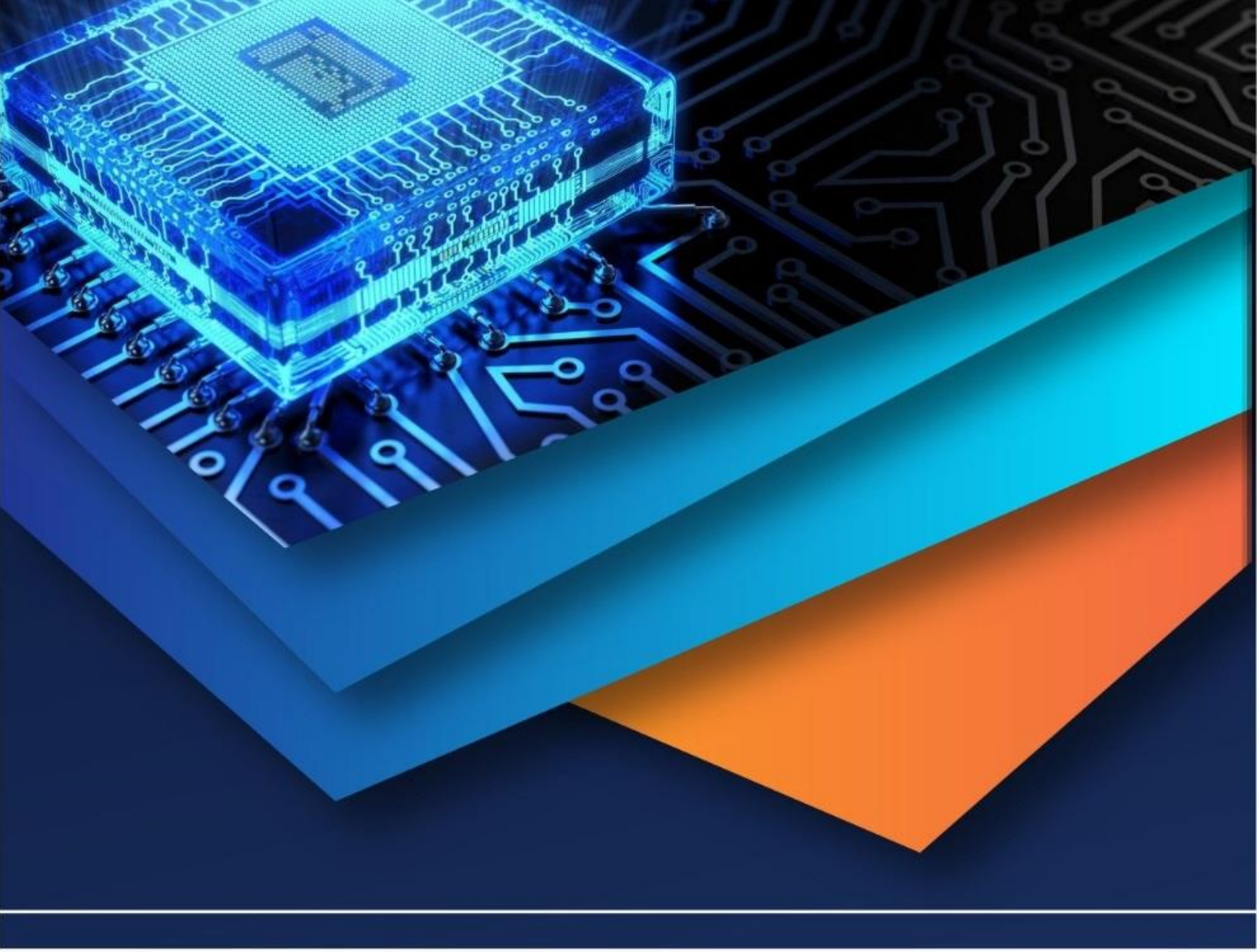
REFERENCES

[1] Google Cloud Storage: API Reference Guide, 2013.
[2] The Neo4j Manual v2.0.3, 2014.
[3] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. G¨unth¨or, and C. Mohan. Advanced transaction models in workflow contexts. In Proceedings of IEEE ICDE, pages 574–581, 1996.
[4] J. Baker, C. Bondc¸, et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In CIDR, pages 223–234, Jan. 2011.
[5] B. Calder et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In SOSP'11, pages 143–157, 2011.
[6] B. F. Cooper, A. Silberstein, et al. Benchmarking cloud serving systems with YCSB. In SoCC '10, pages 143–154, 2010.
[7] J. C. Corbett, J. Dean, et al. Spanner: Google's globally-distributed database. In OSDI '12, pages 251–264, 2012.
[8] C. Daboo, B. Desruisseaux, and L. Dusseault. Calendaring Extensions to WebDAV (CalDAV). RFC 4791 (Proposed Standard), March 2007.
[9] S. Das, D. Agrawal, et al. G-Store: a scalable data store for transactional multi key access in the cloud. In SoCC '10, pages 163–174, 2010.
[10] A. Dey, A. Fekete, R. Nambiar, and U. R¨ohm. YCSB+T: Benchmarking Web-scale Transactional Databases. In Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on, pages 223–230, March 2014.
[11] A. Dey, A. Fekete, and U. R¨ohm. Scalable Transactions Across Heterogeneous NoSQL Key-value Data Stores. Proc. VLDB Endow., 6(12):1434–1439, Aug. 2013.

AUTHOR PROFILE

Mr. Dheeraj Bhimrao Lokhande

He received his BE degree in 2008 from walchand Institute Technology from Solapur University,Solapur with first class with Distinction. He Qualified GATE Examination in 2008. He Completed his ME(CSE) postgraduate degree in 2012 from Solapur University, Solapur with First Class. In 2014,He Registered for Ph.D(CSE) from Solapur University,Solapur, Maharshtra, INDIA.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY