A Timestamp based Novel Caching Mechanism for Distributed Web Systems

Jay Parekh
Department of Information Technology
Sardar Patel Institute of Technology
Mumbai, India

Apurv Moroney
Department of Information Technology
Sardar Patel Institute of Technology
Mumbai, India

Lavina Golani
Department of Information Technology
Sardar Patel Institute of Technology
Mumbai, India

Radha Shankarmani
Department of Information Technology
Sardar Patel Institute of Technology
Mumbai, India

ABSTRACT
Microservices are a loosely coupled distributed systems architecture. With the uncertainty in prediction of the size of the application, microservices play an important role in development and scaling. As they are independently functioning applications, difficulties in caching data becomes manifold. Caching in microservices is achieved either by maintaining a local cache with peer to peer communication, or a global cache with single store communication. However, multiple local caches come with communication overheads and data consistency issues, while a global cache has data management issues. This project attempts to find a combination of both to reduce the communication overheads and data size, while solving the problem of data consistency. Focus is to create a mechanism which uses both a global cache and a local cache. The global cache would act as a verification cache and the local cache would act as a data cache. This will minimize the size of the global cache and the communication call size. In comparison to the existing cache management techniques, this system will act as a middle ground. It inherits the low communication overheads from the global caching systems and also manages to keep the global cache size minimum by storing only verification data. The impact of this research topic is multi-faceted, not only in scaling web applications to a global scale but also in maintaining modular, data-consistent caches in a cluster of microservices. Another advantage of the proposed solution is that it can ameliorate the problem of bandwidth always falling short in high load applications

General Terms
Distributed Systems, Cache Management, Web Applications, Microservices

Keywords
distributed systems, caching, microservices, web applications, web services

1. INTRODUCTION
1.1 Microservices
A microservice is a small, loosely coupled, independently deployed, independently scaled, and independently tested application that has a single responsibility in the scheme of a larger distributed web application [1]. Typically, these microservices are designated to handle certain granular functionalities of the larger application [2]. These features grant microservices based applications unparalleled levels of scalability, modularity, flexibility, and maintainability [3]. However, the adoption of the microservices architecture accompanies a few problems that are inherent to distributed systems. As the system grows to unexpected scale, the underlying communication systems begin to tumble because of the uncertain nature of large-scale networks [4]. This is relevant to this discussion, as will be seen in further sections.

1.2 Web Caching
In large scale web applications, peak network traffic usually far supersedes current network capabilities due to bottlenecks present at various junctures of a network route. This leads to traffic congestion in the network and significant performance drops. One of the most convenient solutions to this problem is the concept of caching. Caching is the storage of data at locations that are logically nearer to the consumer of said data [5]. Caches can be consumer-oriented or provider-oriented [6]. This discussion concerns provider-oriented caching mechanisms. These are systems where the data is cached (replicated) at a site near the provider of the data to the end user. In the case of microservices based applications, caching sites must be situated near the application servers that interface with the user through the HTTP protocol. “There are only two hard things in Computer Science: naming things and cache invalidation” - this is a notoriously popular quote by designer Phil Karlton. Cache invalidation is the practice of freeing a cache store of data that is old and does not correctly reflect the value of the original data that it is replicating. This is necessary so as to avoid inconsistency in the different data stores that simultaneously serve data. The problems in invalidating cache entries are further exacerbated in the case of caching in distributed systems. In large scale web applications, it is common for the database and the application servers functioning independently and connected only through a wide area network. This creates a motivation for application servers to cache database objects while in the process of serving them to the end user. In monolithic web applications (not distributed; composed of a

single application server), cache management is a simple matter because of the presence of only one single copy of the primary database. However, in a microservices based web application, with the presence of a large number of distributed points of replication, the level of complexity in cache management and invalidation rises exponentially.

1.3 Caching in distributed web systems - Status Quo

Presently, there are two parallel caching techniques commonly employed in microservices and other distributed web environments - Distributed (Global) Caching, and Localized Caching.

1.3.1 Distributed (Global) Caching

This design is characterized by a single primary data source as well as a single global cache store shared by all the application servers that access the database. This architecture essentially solves the data inconsistency issue since the cache is only composed of one single data store without any active replicas. It is important to note, however, that this global cache may be replicated into multiple copies, but these copies only serve as backup entities and do not actively interface with and serve the application servers. The single cache store also may be distributed across multiple nodes, but this distribution is transparent to the user and the application server, thus essentially acting as a singular unit. Since this single unit of cache is shared by multiple application servers, it must be connected to those servers over the network. This incurs a huge latency cost, when compared to caches that are connected to the application server over a local network, or caches residing on the same machine as the application server. Also, this single cache unit is also a single point of failure in the system. When this cache unit goes down, the application servers are left with no other means of caching their frequently accessed data. Ensuring high availability of this global cache unit is not straightforward and highly resource intensive. This is because it involves complicated distributed communication protocols to maintain data consistency between replicas, while enforcing a master slave architecture. Redis is a widely used distributed caching solution. Fig 1 shows a typical architecture of a distributed web application employing a distributed caching solution.

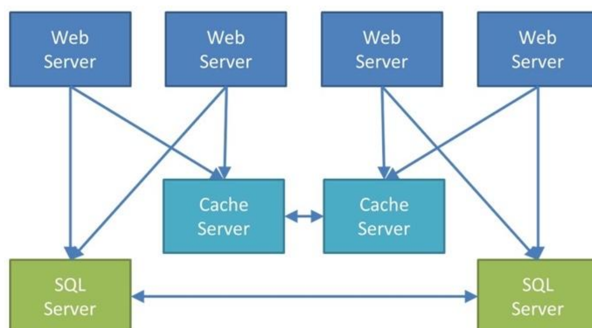

Fig 1 Distributed caching solution - Typical architecture

1.3.2 Localized Caching

In this approach, each application server maintains its own separate cache unit, which is stored either at the same machine as the server, or on a different machine in the same local network. This design enables modular caching as each application server can build and maintain its cache unit tailored to its requirements. However, as the original data source is still singular and shared by multiple such cache units, this design suffers from data consistency problems. Every update on a data object in the database must be propagated to every cache unit that has replicated that object. This also causes a massive communication overhead as well as a networking overhead, since it needs to maintain membership lists consisting of every 'node' or server that has cached data. This degrades the overall performance of the application from the perspective of the end user. Fig 2 shows a typical architecture of a distributed web application employing a local caching solution.

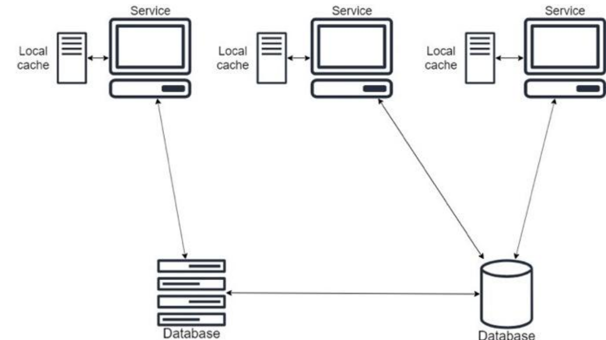

Fig 2 Localized Caching Solution - Typical Arthitecture

1.4 Objectives

As noted in the previous subsection, both the distributed caching and local caching approaches currently in fashion have certain advantages and certain disadvantages. The aim of this project is to design a system that offers the better features of both these approaches while overcoming the shortcomings of the same. The objectives of this article can be summarized by the following points:

- Create an efficient caching mechanism: Design a caching mechanism which uses the advantages of both global and local caching mechanisms i.e.: consistency and low latency.
- Create a caching mechanism which supports heavy reads: Design a system that can sustain high number of read request, while serving correct values of data to each of these requests.
- Design a scalable caching mechanism: The cache should be easily scalable and have the ability to function under a huge load without suffering any significant loss in performance.

2. LITERATURE REVIEW

In [7], a comparison is made between different caching solutions employed in distributed clusters. It elaborates on caching in distributed clusters. Caching solutions vary from application to application making a common solution very difficult. This paper first focuses on these different methods of caching in distributed systems. Later, it proposes a new method of caching which uses heuristics. These heuristics aid in reducing data transfer cost occurred while invalidation. In [8], the authors have addressed the problem of caching in a mobile environment. As in mobile environments, network is not consistent and disconnection is a problem, common cache strategies are not suited. Invalidation reports is a method suggested for such environments, but it has its own disadvantages. Long query latency and bandwidth wastage are the major drawbacks of the current IR based invalidation of caches. This paper proposes a modification in invalidation reports strategy by reducing the latency of queries. The proposed method also

uses the network bandwidth efficiently by reducing the number of uplink requests. In [9], the authors focus on the detailed analysis of cache invalidation patterns of applications. It also predicts the patterns of invalidation beyond 32 processors. The main idea proposed is a classification scheme for objects present in parallel programs. The results show that it is possible to write programs in parallel systems without the invalidation system fulminating. The classification scheme detects that the use of directory-based scheme is effective with a reduced number of pointers. Final result of the paper indicates that lowest data is generated, and least number of invalidations happen in the 32-byte range. In [10], they identify the cache replacement and invalidation mechanisms for data dependent on locations in mobile environments. A geometric location model is used to study the data. Apart from analysis of the data based on location, two new cache replacement algorithms are suggested. PA and PAID strategies take into consideration not only the location but also the distance from the client to the scope of data and scope of the data itself. If consideration of the valid scope, PA and PAID perform better than existing strategies.

3. METHODOLOGY

Firstly, an extensive survey on the common practices currently followed by the industry in developing scalable, distributed web applications is performed. A majority of these applications were found to be microservices based. Upon investigating about the problems commonly faced while building these solutions, it was realized that employing a reliable, economical caching system in these applications was a notoriously difficult undertaking. On delving deeper into the caching woes faced by these applications, it was found that presently there is no caching mechanism in practice that adequately surmounts the difficulties posed by the distributed nature of large-scale web applications. This led to the conclusion that presently there are two major different caching techniques employed by most web applications. These are described in detail in the Caching in Distributed Web Systems subsection of the Introduction section. Both these mechanisms suffer from certain shortcomings. From here on, the research was directed towards devising a caching mechanism for distributed web applications that can overcome the shortcomings of the two major existing solution mechanisms. Timestamps are a concept used to attain features like synchronicity, consistency, and sequence in various algorithmic and other problems in Computer Science. Timestamps have been proposed as a mechanism to perform cache invalidation in web search engines [11]. It was found that there is applicability for timestamps to be used as a device to blend the two caching mechanisms for distributed web systems discussed above. Once a rudimentary caching solution using timestamps, that was a combination of the distributed (global) and local caching mechanisms was devised, both logical and technical flaws in the system as it was, were discovered. This led to a road map of minor and major increments to the system, at the end of which a fully developed, robust system that satisfactorily addresses all the concerns regarding the two existing caching mechanisms, and the minor flaws found in the initial versions of the system, was ready. This novel caching mechanism essentially integrates the two existing mechanisms using timestamps. The rudimentary version of this mechanism had certain flaws. These flaws included issues with sequence and synchronicity, which in order to overcome, an algorithm was devised, for the correct execution of read and write requests on the system. To achieve this, a few state variables were introduced, and also a maneuver involving queuing of write requests on the global cache server. What is proposed and presented in this article is the final iteration of this system, and it adequately addresses all the flaws that cropped up during the various stages of the research and ideation

3.1 Proposed System

There are 2 operations that are done by the user - Read & Write.

- Read - Whenever a read is requested, the request goes to the global cache. Previous Write ID of this read is set as Latest Write ID and the read count is incremented by 1. If timestamp is not present in the global cache, the data is queried from the database and the timestamp is added to the local cache. Also, the data is set to the local cache and the data and timestamp are set in the local cache. This completes the read operation and increments the completed read count by 1. If timestamp is present in the global cache, we check if the status of the object is unlocked and previous Write ID of this read is equal to the completed read. – Until the above criteria is satisfied, polling is done every 500ms. – If this condition is satisfied, the timestamp from the global cache is returned to the local cache. If the timestamp of the local cache is greater than or equal to the timestamp of local cache, the data is fresh. But if the local cache timestamp is less than the global cache timestamp, it means that the global cache was updated after this server's last read. So, this local cache has stale data. a request is made to the database to fetch the latest data. The timestamp of the local cache is updated. This completes the read operation and increments the completed read count by 1. The flowchart for this operation is depicted as Fig 3.
- Write - When a write request is sent by the user, it goes to the global cache. This write ID is appended to the queue, the Latest Write ID is set as this Write ID and the previous read count for this write is made equal to the read count. Subsequently, the read count is set to 0. It is then checked whether the object satisfies the following conditions: its status is unlocked; the write ID of the request is at the head of the queue and the previous read count for this Write ID is equal to the complete read count. Until the above conditions are satisfied, the request keeps polling every 500ms. When all the above conditions are satisfied, the write request locks the object status and proceeds to update the database. After the write operation is completed- the global cache timestamp is updated, the completed Write ID of the object is set as this Write ID, the status of the object is unlocked and the completed read count is set to 0. The write response is given to the user. The flowchart for this operation is depicted as Fig 4.

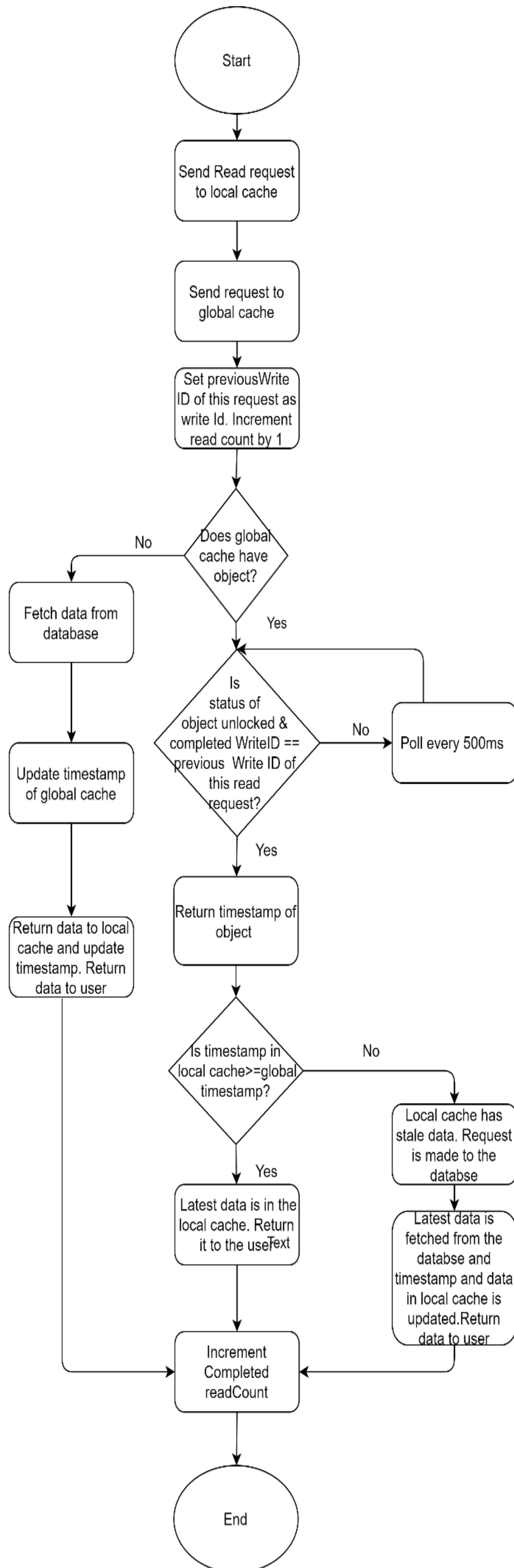

Fig 3 Read Request Flowchart

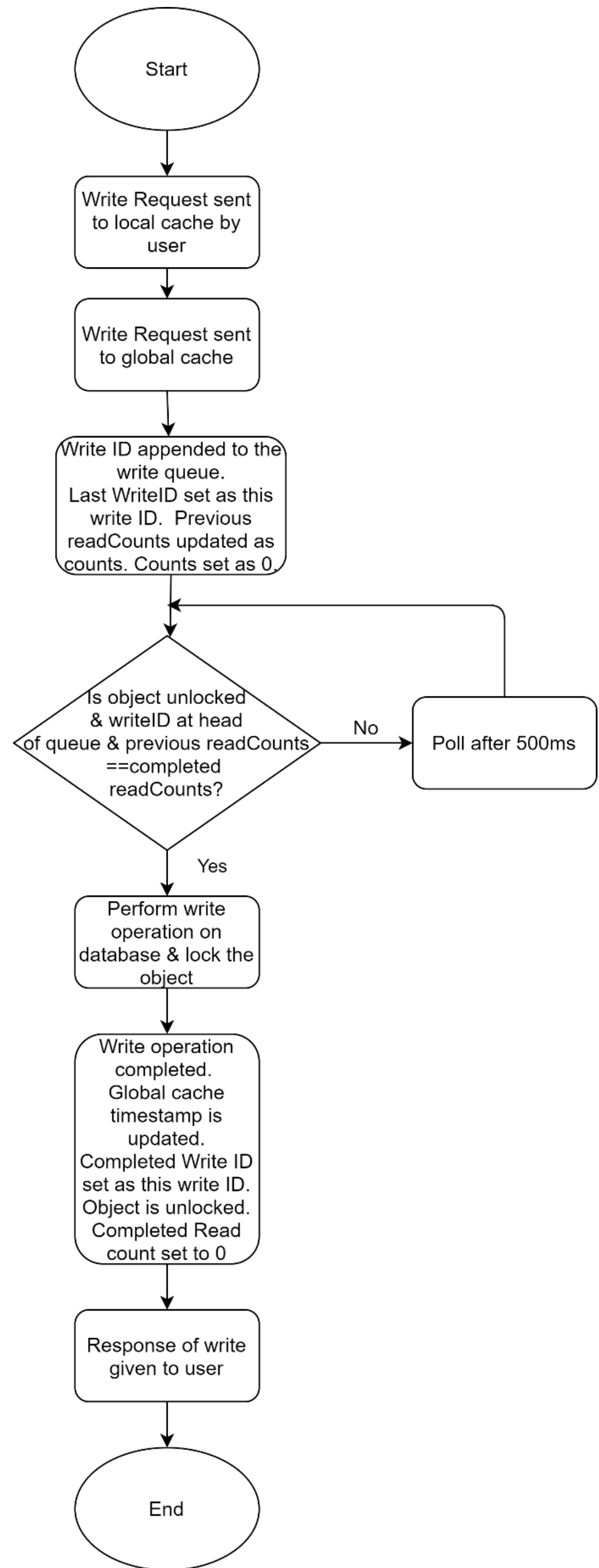

Fig 4 Write Request Flowchart

3.2 System Architecture

Fig 5. illustrates the proposed system architecture - Node JS and its framework Express JS is used to write the server-side code. A node library, called nodecache was used to build the in-process cache to cache the data. Mongo DB is used as the database which takes care of distributing and backing up the data. Redis is used to build the global cache. Mongo DB is a NOSQL Database which is deployed on Scale grid. Redis is

a key-value document store. This is also deployed on Scale Grid. The application servers are deployed to Microsoft Azure cloud machines. This deployment of all the servers and databases on the cloud is performed to see how the system works with the latency of the internet.

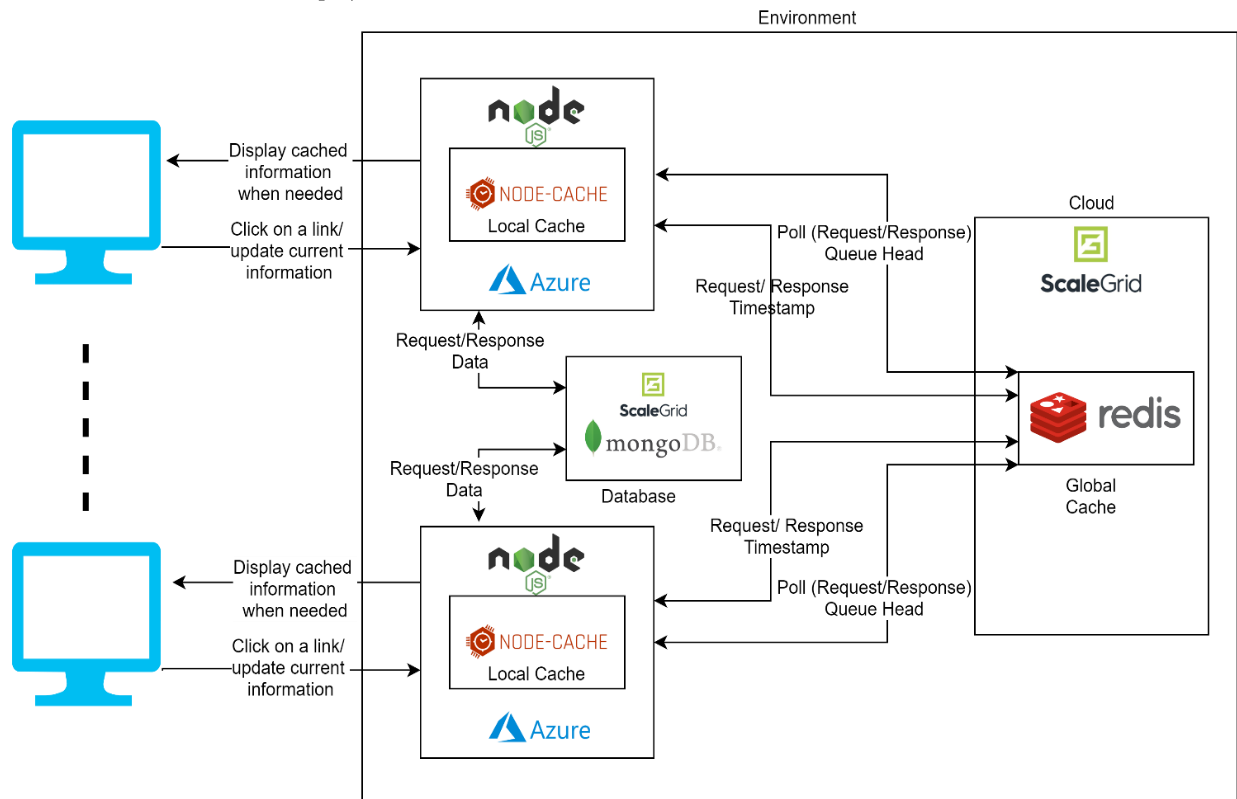

Fig 5 Proposed System Architecture

3.3 Process Flow

Fig 6. depicts the process flow. There are 3 servers in this setup. All operations take place on object X. Initially, the global cache has the following values for object key X: status-unlocked, timestamp-nil, latest Write Id-nil, latest Completed Write ID- nil, read count-0 & completed read count-0. A write operation(W1) for object X is sent from server 1. The write request is appended to the queue with previous read count as 0(which is the read count) the read count is set to 0 and the latest write ID is set as. Since W1 satisfies all conditions for a write operation to execute, the status of the object is locked, and the write operation starts. Meanwhile, a read request (R1) is sent from server 2. The read count is increased by 1 and now becomes 1. Since the read conditions for R1 are not satisfied, it keeps polling every 500ms. A write request (W2) is made from server 3. The write request W2 is appended to the queue with previous read as 1, read count is updated to 0 and the latest write ID is updated as W2. Since, all its conditions are not satisfied, W2 keeps polling every 500ms. The write Request W1 is completed, it is removed from the queue and the status of the object is set to unlocked and the latest completed Write ID is set as W1. Since the conditions of read R1 are satisfied, it executes and the read complete count increases by 1 and now becomes 1. After the read R1, conditions for W2 are satisfied and it locks the object and executes. After its execution is completed, status of the object is unlocked, the latest completed write ID becomes W2 and the completed reads is set as 0.

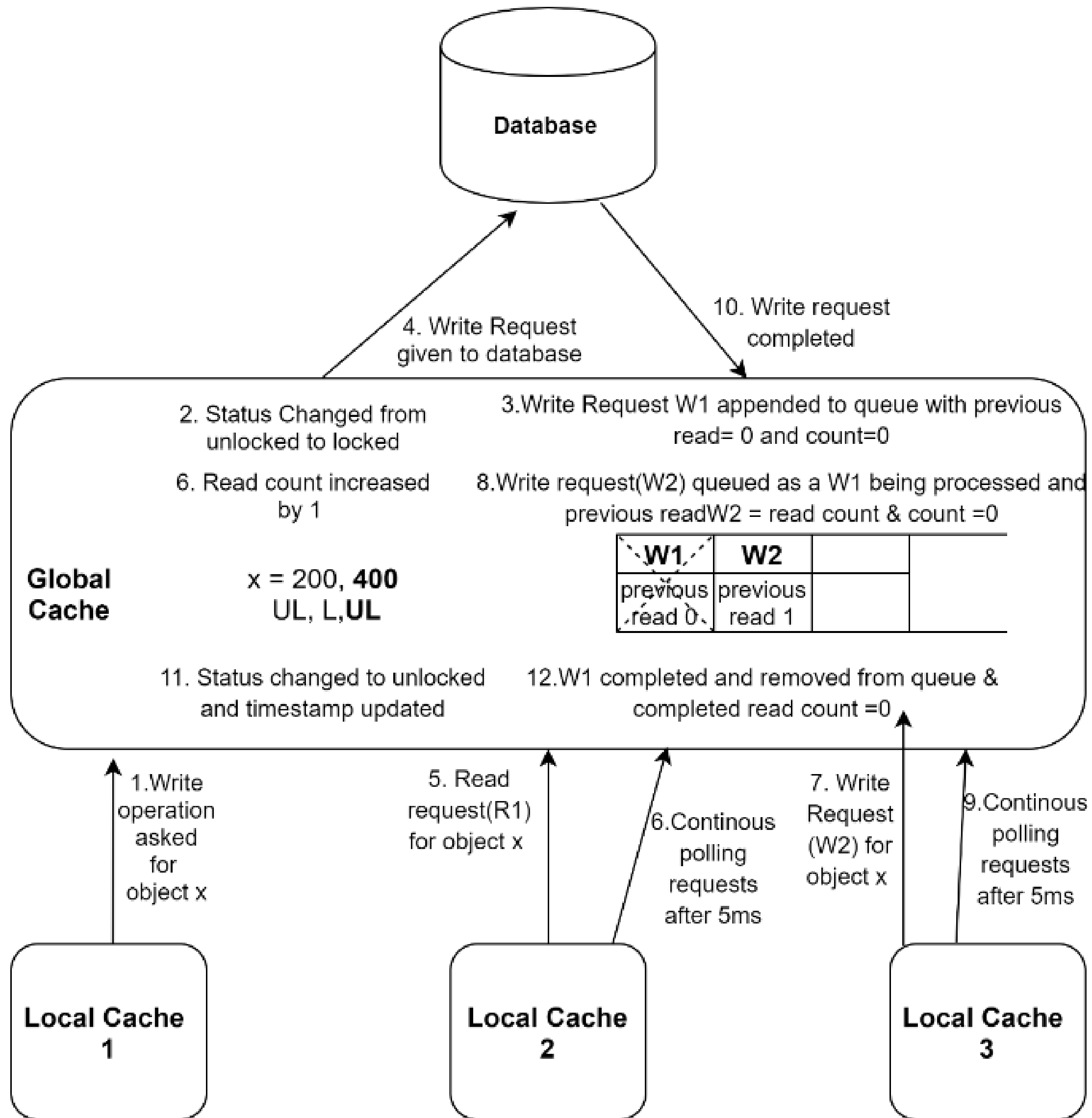

Fig 6 Process Flow Diagram

3.4 Status table

Let us assume that the operations are carried out on object X. There are 2 local servers- server 1 and server 2. The following is the sequence of requests made in increasing order of time. It is also mentioned that which request comes from which cache.

Order of Operations	R1	W1	R2	W2	R3	W3
Comes from which Cache	1	1	2	1	1	2

Legend for the table-

- TS- timestamp
- S- status of object
- QS- queue status
- LW- Latest Write ID(last write ID entered in the queue)
- LCW- Latest Completed Write ID(write ID of the latest completed write)
- RC- read count
- CRC- Completed Read count
- PRC- Previous read count(the reads that take place before write)

Table1 summarises the states of the variables for the given sequence of requests

Table 1 Variable status table for given sequence of operations

Operation	DB value	Global Cache (TS,S,QS), (LW,LCW) ,(RC,CRC)	Local Cache 1	Local Cache 2
Initial	A	No TS, UL, [ ] Null, Null, 0, 0	TS: No TS Data : No data	TS: No TS Data : No data
R1 arrives	A	No TS, UL, [ ] Null, Null, 1, 0	TS: No TS Data : No data	TS: No TS Data : No data
R1 completes	A	100, UL, [ ] Null, Null, 1,1	TS: 102 Data : A	TS: No TS Data : No data
W1 arrives	A	100, L, [W1/1] W1, Null, 0,1	TS: 102 Data : A	TS: No TS Data : No data
W1 completes	B	106, UL, [ ] W1, W1, 0, 0	TS: 102 Data : A	TS: No TS Data : No data
R2 arrives	B	106, UL, [ ] W1, W1, 1,0	TS: 102 Data : A	TS: No TS Data : No data
W2 arrives	B	106, UL, [W2/1] W2, W1, 1, 0	TS: 102 Data : A	TS: No TS Data : No data
R2 completes	B	106, UL, [W2/1] W2, W1, 1, 1	TS: 102 Data : A	TS: 108 Data : B
R3 arrives	B	106, L, [ ] W2, W1, 1,1	TS: 102 Data : A	TS: 108 Data : B
W3 arrives	B	106, L, [W3/1] W3, W1, 0, 1	TS: 102 Data : A	TS: 108 Data : B
W2 completes	C	112, UL, [W3/1] W3, W2, 0,0	TS: 102 Data : A	TS: 108 Data : B
R3 completes	C	112, UL, [W3/1] W3, W2, 0,1	TS: 116 Data : C	TS: 108 Data : B
W3 completes	D	118, UL, [ ] W3, W3, 0, 0	TS: 116 Data : C	TS: 108 Data : B

4. RESULTS

In this section, the prototype developed based on the proposed system is discussed. The functioning of the prototype is illustrated, and then the results obtained are examined, which indicates an appreciable improvement in

performance when compared to systems based on a global caching mechanism. The prototype application is deployed on Microsoft Azure cloud service, the global timestamp server is also deployed on Azure through Scalegrid in the form of a Redis instance. The central database is also deployed on Azure through Scalegrid as a MongoDB instance. The application servers are written in NodeJS using the Express framework. The in-process cache attached with every application server instance is set up using a NodeJS library called NodeCache. First, the functioning of the prototype is delineated through screen captures of live updates during execution of Read and Write requests on the server. Fig 7 illustrates this. Then, the raw request execution times for a read request on both the systems are displayed, one using the hybrid cache mechanism, and the other using the global cache mechanism. Figures 8 and 9 show the response times for read requests on the two systems. The requests are generated and sent using Postman, an open source tool for HTTP request generation. As seen in the figures, a significant improvement in response time can be observed when the application shifts from using the global cache system to using the hybrid cache system. A performance jump of around 400% is observed. This is illustrated in by the graph drawn in Fig 10.

```
py$ node index.js
(node:17282) DeprecationWarning: current URL string parser is deprecated, and wi
ll be removed in a future version. To use the new parser, pass option { useNewUr
lParser: true } to MongoClient.connect.
(node:17282) DeprecationWarning: current Server Discovery and Monitoring engine
is deprecated, and will be removed in a future version. To use the new Server Di
scover and Monitoring engine, pass option { useUnifiedTopology: true } to the Mo
ngoClient constructor.
Server is up and running on port numner 1236
successfully authenticated with redis cluster
Mongo connected

PUT Request:
WRITE: received hgetall response
WRITE: Redis entry for the object found
WRITE: previousReads value saved: 1
WRITE: readCount value set to 0
WRITE: current requestId pushed to queue
WRITE: Starting poll procedure...
WRITE: object found unlocked, previousReads: 1 obj.completedReads: 1
WRITE: object is locked before writing
WRITE: after actual update
WRITE: object is unlocked after update, completedReadCount set to 0
```

```
PUT Request:
WRITE: received hgetall response
WRITE: Redis entry for the object found
WRITE: previousReads value saved: 0
WRITE: readCount value set to 0
WRITE: current requestId pushed to queue
WRITE: Starting poll procedure...
WRITE: object found unlocked, previousReads: 0 obj.completedReads: 0
WRITE: object is locked before writing

GET Request:
READ: readCount updated to 1
READ: received hgetall response
READ: Object lock status: 1
READ: before starting polling
WRITE: after actual update
WRITE: object is unlocked after update, completedReadCount set to 0
READ: saved writeRequestId and latestCompletedWriteRequestIds match, ending poll
ing
READ: object found in local cache
READ: local value is stale (timestamps do not match)
READ: completedReadCount updated to 1
```

Fig 7 Screen capture of live updates during requests execution

58:1235/sports/Federer

leaders (7) Body Pre-request Script Tests Settings

VALUE DESCRIPTION

Value Description

Test Results Status: 200 OK Time: 27.86 s Size: 8.37

beProject20@BeProject20-redis: ~/nicecache/sports_news_redis

```
uccessfully authenticated with redis cluster
oc not found in redis
oc found in redis
oc found in redis
oc found in redis
K
oc not found in redis
oc found in redis
C
eProject20@BeProject20-redis:~/nicecache/sports_news_redis$ node index.js
node:30717) DeprecationWarning: current URL string parser is deprecated, and wil
 be removed in a future version. To use the new parser, pass option { useNewUrlP
rser: true } to MongoClient.connect.
node:30717) DeprecationWarning: current Server Discovery and Monitoring engine i
 deprecated, and will be removed in a future version. To use the new Server Disc
ver and Monitoring engine, pass option { useUnifiedTopology: true } to the Mongo
lient constructor.
erver is up and running on port numner 1235
uccessfully authenticated with redis cluster
K
oc not found in redis
oc found in redis
```

AAMAAAABA
vcCBDQyAy
ACAACIMgA
wADAAAAAC
AQABAACkA
6MTIAMjAx
AAAQAAAAA
JYTVAgQ29
zLzEuMC8i
htcE1NPSJ
G1sbnM6c3
b3Rvc2hvc
yczpMdW1p
FtZXRyaWN
0dyYXlzY2

Fig 8 Response time for a read request on the global cache system

```
13.78.151.28:1234/sports/Federer

beProject20@BeProject20: ~/nicecache/sports_news_timestamp
successfully authenticated with redis cluster
Mongo connected

GET Request:
READ: no entry found in global cache
READ: making entry to redis with initial values
READ: inserted object into NodeCache

GET Request:
READ: readCount updated to 1
READ: received hgetall response
READ: Object lock status: 0
READ: object found in local cache
READ: local value is fresh (timestamps match)
READ: completedReadCount updated to 1

GET Request:
READ: readCount updated to 2
READ: received hgetall response
READ: Object lock status: 0
READ: object found in local cache
READ: local value is fresh (timestamps match)
READ: completedReadCount updated to 2
```

Fig 9 Response time for a read request on the hybrid cache system

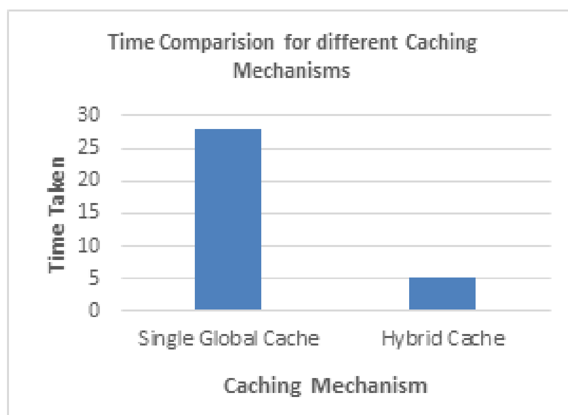

Fig 10 Performance Gain through Hybrid Caching

5. LIMITATIONS

The authors do not intend to advertise this system as a cure-all mechanism that can provide efficient, fast caching to every kind of distributed web application. In fact, when it comes to caching on networked systems, it is believed no 'one size fits all' solution exists. As such, this proposed system has certain features that may prove to be drawbacks in the following forms: This system is not expected to perform well when integrated with applications that suffer from a heavy load of write requests, since in such a scenario, the server will have to frequently fetch fresh data from the database.

The global timestamp server may turn out to be a single point of failure if adequate backup mechanisms are not in place. It is important to note however, that setting up backup plans for this timestamps server through replicas is much easier than doing the same for the global cache server that stores the actual data.

Table 2 Comparison of the three caching approaches in various respects

Approach/Characteristic	Global Cache	Local Cache	Proposed Solution
Suitable in	Write-heavy applications	Read-heavy applications	High reads, low writes
Network Transfer Size	Entire Data Object	None	Only Timestamp

Data Transferred Over	WAN	LAN	LAN
Data Consistency	Immediate	Eventual	Immediate
Network Latency	High	Low	Low
Cache Modularity	No	Yes	Yes
Single Point of Failure	Yes	No	No
Amount of Data Stored on Cache Service	High (Entire Data Objects)	None	Low (Only Timestamp)

6. CONCLUSION

Through this article, the authors have demonstrated that it is possible to integrate the two commonly employed caching mechanisms in distributed web environments in order to reap the benefits of both, using timestamps as the crux of the solution. Table 2 depicts a comparison of the three caching approaches and thus illustrates the benefits of the proposed system. The proposed system, in essence, is a hybrid combination of both the existing solutions, the global cache system and the local cache system. Development of a prototype for the proposed system was undertaken, and a significant performance gain in the response times for read requests was observed, when compared to the existing global cache systems. Having addressed the limitations of this system, it is concluded that this caching mechanism can provide excellent performance gains for distributed web application systems that function under the constraints mentioned in the previous section.

6.1 Future Scope

Here, the authors acknowledge that there are certain areas in the system with potential for enhancement that may further improve the efficiency and performance of the system. Discussion on them is omitted either because they are highly tangential to the primary objectives as put forth at the start, or because they involve domains that the authors do not wish to touch upon within the scope of this article. These areas include, but are not limited to, setting up cache eviction policies based on Machine Learning on the local caches, and devising failure control mechanisms for both the global timestamp server and the local caches.

7. REFERENCES

[1] J. Thones. “Microservices”. In: IEEE Software 32.1 (2015), pp. 116–116.

[2] Claus Pahl and Pooyan Jamshidi. “Microservices: A Systematic Mapping Study.” In: CLOSER (1). 2016, pp. 137–146.

[3] Nicola Dragoni et al. “Microservices: Yesterday, Today, and Tomorrow”. In: Present and Ulterior Software Engineering. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. ISBN: 978-3-319-67425-4. DOI: 10.1007/ 978-3-319-67425-4 12. URL: https://doi.org/10.1007/ 978-3-319-67425-4 12.

[4] Werner Vogels, Robbert van Renesse, and Ken Birman. “The Power of Epidemics: Robust Communication for Large-Scale Distributed Systems”. In: SIGCOMM Comput. Commun. Rev. 33.1 (Jan. 2003), pp. 131–135. ISSN: 0146-4833. DOI: 10.1145/774763.774784. URL: https: //doi.org/10.1145/774763.774784.

[5] Michael Rabinovich and Oliver Spatscheck. Web caching and replication. Vol. 67. Addison-Wesley Boston, USA, 2002.

[6] G. Barish and K. Obraczke. “World Wide Web caching: trends and techniques”. In: IEEE Communications Magazine 38.5 (2000), pp. 178–184.

[7] Pooja Kohli and Rada Chirkova. “Cache Invalidation and Update Propagation in Distributed Caches (extended abstract)”. In: (June 2010).

[8] Guohong Cao. “A scalable low-latency cache invalidation strategy for mobile environments”. In: IEEE Transactions on Knowledge and Data Engineering 15.5 (2003), pp. 1251–1265.

[9] A. Gupta and W. -. Weber. “Cache invalidation patterns in shared-memory multiprocessors”. In: IEEE Transactions on Computers 41.7 (1992), pp. 794–810.

[10] Baihua Zheng, Jianliang Xu, and D. L. Lee. “Cache invalidation and replacement strategies for locationdependent data in mobile environments”. In: IEEE Transactions on Computers 51.10 (2002), pp. 1141– 1153.

[11] Sadiye Alici et al. “Timestamp-Based Result Cache Invalidation for Web Search Engines”. In: Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval. SIGIR ’11. Beijing, China: Association for Computing Machinery, 2011, pp. 973–982. ISBN: 9781450307574. DOI: 10.1145/2009916.2010046. URL: https://doi.org/ 10.1145/2009916.2010046.