Fault Tolerant Central Saga Orchestrator in RESTful Architecture

Konstantin Malyuga, Olga Perl, Alexandr Slapoguzov, Ivan Perl
ITMO University
St.Petersburg, Russia
konstantin.malyuga@gmail.com, ovkalyonova@itmo.ru, slapoguzov@gmail.com, ivan.perl@itmo.ru

***Abstract*—Microservices RESTful architecture is almost a standard for e-commerce web applications today. It brings domain isolation, development and support independence. In the same time it increases complexity of cross-domain interactions. The case when distributed changes should not break consistency of microservice states is one of challenging task that might appear during development of such systems. Defining sagas in central orchestrator that performs changes in microservices one by one and controls compensations in case if failure occurs is one of the well-known approaches today. It solves the problem of consistency but creates new vulnerable area in fault tolerant environment. Usage of saga cluster and additional optimizations of its structure are modeled, evaluated and proposed in this paper. Provided fault tolerant solution with improved time and memory characteristics.**

I. Introduction

Microservices architecture brings data isolation in scope of concrete domain. Only domain service allowed interacting with domain data usually stored in database [1]. It is no longer possible to apply changes in multiple domains at once or to perform rollback of changes in database in case of failure. All changes should be performed through the service API as well as implementation calls should be implemented as part of service.

There are few widespread ways to organize data consistency in microservices environment:

- Two-phase commit protocols (2PC) to support distributed transactions. Implementation of such protocol should be additionally integrated to a system that will performs changes in services only when all services have prepared their data for changes and waiting for additional signal to perform commit[2]. Locked data that is going to be updated should be immutable during transaction that leads to either additional delays on commit wait or potentially data lost if no DB locks actually applied[3]. Also, changes in services should be made to support 2PC protocol that handles distributed transaction flow. Due to these reasons this approach is not so popular in the modern e-commerce applications [4].
- Eventual consistency approach implemented in saga patter. Saga is a described sequence of changes to be applied in microservices one by one and compensation algorithm for each request that should establish consistent state in services in case of changes failure[5]. Compensation actions does not differ from requests to apply direct changes so that services might not require additional changes in case of saga pattern introduction.
  Usage of this approach does not violate isolation advantage of microservices architecture and allows avoiding additional modification of endpoints so that RESTful architecture of services might still exists.

There are two types of saga implementation [6]:

- Choreography. In this case microservices are linked to the chain of sequential calls as depicted at Fig. 1. Each service will perform changes on event or request, emits successful event and awaits for the whole saga to proceed. In case of failure during local changes service should emit failure event to the previous service. This previous service should apply compensation logic and send failure event to a previous service of it.
  Expectation for successful or failed response for service does not imply introducing additional data states as 2PC does.
- Orchestration. This type of sagas depicted on Fig. 2. It requires implementation of saga orchestrator that controls sagas calls sequential. Orchestrator sends commands to services in the chain one by one, awaiting for successive response. In case of failure it sends compensation calls to the services where changes was already made to return such services to the previous state.

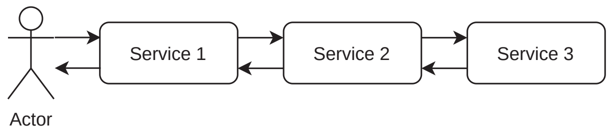

Fig. 1. Sequentially chained services in choreography saga

Implementation of sagas in choreography manner brings changes to microservices that takes part in chain of calls and couples services closely. Commands and compensations mechanism often designing for every pair of services but there are series of frameworks that simplifies this process [7]. Performance assessments shows [6] that choreography works faster due to no additional transport channels. This type of sagas recommended for begging sagas integration and for simple and small sagas [4].

On the other hand, orchestration allows using abstraction over services logic and useful in complex applications to loose coupling. Absence of necessity to update service API or it's behaviour during saga integration is also an advantage of orchestration based sagas. However, organization of external component increase complication and brings additional time expenses that appear because additional communication overhead becomes part of every command in saga flow.

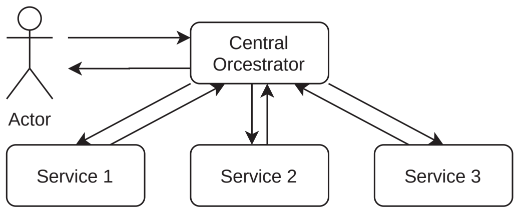

Fig. 2. Managed by central orchestrator services in orchestrational saga

New saga might be implemented as a new coordinator every time. In this case introduction of a new saga becomes complex process due to new service configuration and deployment should be prepared but not only saga flow declaration. That is why having all sagas of the system central orchestrator might become more satisfying approach. It means sagas are implemented as components of one service, so that development of a new saga logic requires no overhead for engineering.

II. Cluster organization

Creation of important central component that governs set of important system flows introduces new point of failure. In case of central orchestrator, failures on such component able to ruin data consistency in case of shutdown during saga.

Introduction of active-passive availability pattern to the structure of central orchestrator allows achieving high availability of a system and improves its tolerance to failures [8]. Such cluster solution should include active (master) and passive (slave) nodes. Each active node controls saga flow and replicates flow information to the passive nodes.

Researches often [4] [6] rely on event type of services communication using message brokers like RabbitMQ or Kafka when propose saga architecture. However, usage of HTTP endpoints to communicate with services to perform local changes does not contradict with initial idea of sagas [5]. It allows orchestrator to receive result of command immediately without intermediate components.

If saga relies on messaging mechanism, master node failure during local transaction process allows to continue saga after cluster startup due to one of the slaves may retrieve or receive response message of a microservice instead of master instance that went down. It is possible since messages persisted in message broker so that system might wait until slave instance will start up, retrieve message and continue current saga so that cluster will be restored.

However, additional struggles might be faced if saga relies on REST HTTP calls. As depicted in Fig. 3, if master node goes down during request-response procedure call to microservice before response received it is no longer possible to identify if local changes succeeded or failed even if slave node will replace failed master node. Due to specified reason additional restriction in fault tolerant environment should be applied: implementation of RESTful service methods for saga usage should be idempotent so that cluster failure during one of local transaction would not leave services in inconsistent state. Otherwise, repeat of request to non-idempotent endpoint might lead to changes duplication that can bring undesired changes accumulation. And skipping of endpoint call after cluster startup might lead to absent of required changes in local transaction so distributed data structure will become inconsistent.

Fig. 3 shows saga local transaction that takes less time than cluster restore. It is possible that local transaction will take more time than cluster restore so that second call that restored cluster performs will lead to parallel processing of it. Still, idempotency of affected endpoint allows avoiding conflicts or accumulation of changes.

It is important to mention that having only idempotent endpoints involved to saga becomes restriction that might not contradict with current system but take part during saga extension in future. In some cases endpoints change to make them idempotent can take less effort than integration of message broker for communication purposes during saga. But amount of required changes overall might overweight migration to messaging system in terms of complexity.

Replication of event based sagas does not require additional communication between nodes of saga cluster due to all nodes can be subscribed on message type that active node sends to service.

Meanwhile, HTTP requests to service from active node of saga cluster should be anticipated by replication calls in order to prepare for active node failure during operation. This is why replication in RESTful environment has a drawback in increase of total saga time. Evaluation of replication time will allow assessment of this drawback. Nevertheless, replication of current saga state in more than one nodes might be performed in parallel so that result delay before each local transaction call depends only on the maximum time all replication calls.

Execution of sagas in master-slave cluster structure allows applying different optimizations for sagas replication data storage and communication between replicas that can minimize replication time, coordinator recovery time and memory requirements.

A. Meaning data optimization

Request and progress data amount have effect on transfer and storage time and also defines required storage capacity of cluster. It might occur that saga introduced to

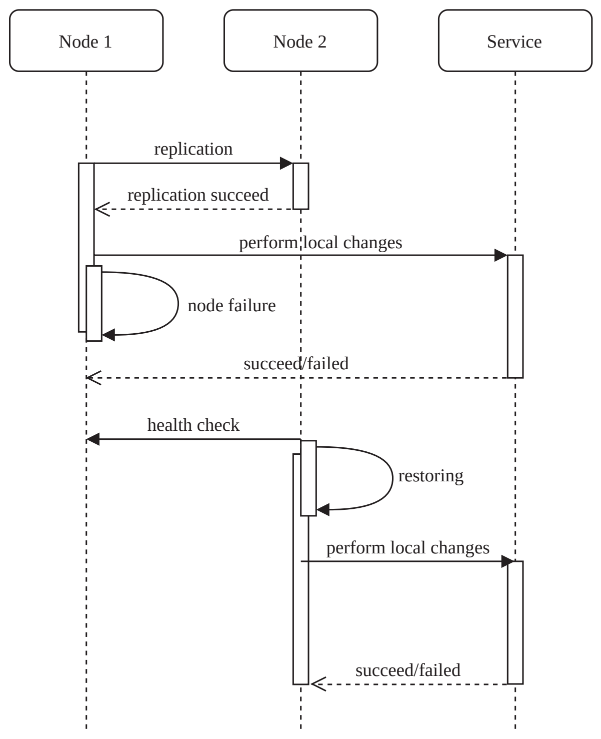

Fig. 3. One saga local transaction with master node failure

the system after it was defined so that endpoints and/or requester provides more data than it's required for saga operation.

Analysis of sagas might be performed and only meaning request and progress data might be replicated due to sagas are predefined flow with a priori known structure. Meaning data is request and progress information in minimal volume that is only required to continue saga progress and to perform compensation calls in case of failure during progress. Analysis of meaning data will not affect each saga execution because sagas have predefined structure. That means calculation of meaning data should be performed for saga only during initialization.

In specific cases meaning data can be only identifiers of resources to be changed during next saga calls so that rest of service information received in the response becomes redundant. Also, self-descriptive endpoints information (HATEOAS) could be excluded from replication due to its redundancy in context of automated saga flow.

Usage of RDF Mapping Language might be required considering specific of case when different endpoints included to saga could even have different data representation. It also allows to change system data representation to optimize preservation during replication. Having different format of replication could reduce replicated data size even more. For instance XML response payload can be replicated as JSON data.

Fig. 4 contains example of response from service. It contains HATEOAS links with autogenerated request paths and bunch of data fields. Part of that fields might be excluded from replication if analysis of saga during

```
<account>
  <account_number>123</account_number>
  <owner_id>456</owner_id>
  <country_code>FR</country_code>
  <balance currency="eur">100.00</balance>
  <restrictions>
       ...
  </restrictions>

  <links>
    <link rel="deposit" href="url/123/deposit"/>
    <link rel="withdraw" href="url/123/withdraw"/>
    <link rel="transfer" href="url/123/transfer"/>
    <link rel="close" href="url/123/close"/>
  </links>
</account>
```

Fig. 4. Service XML response with HATEOAS

```
{
    "account_number": 123,
    "owner_id": 456
}
```

Fig. 5. Replication data after optimization

initialization process shows that not all fields will be required during next commands and possible compensations. HATEOAS data could be excluded due to saga already contains information about commands and compensations to perform. Also change of data representation might reduce total size of replicated data. Fig. 5 shows the result of described changes application.

B. Optional data preservation optimization

Absence of data preservation on hard drive for replicas in case of small amount of progress data might improve preservation time and memory requirements due to saga progress might be not saved in cluster itself. Small amount of data mentioned because avoiding of hard drive usage brings additional restrictions: sagas that was optimized this way should store data of saga in memory until it ends.

Intermediate variation of this optimization is to store only failed sagas data due to such information may be used later to eliminate reason of failure.

Application of first described optimization might reduce memory load to improve benefit of current optimization. However, restriction will still, so this optimization could be applied optionally, considering current server load and saga meaning data size.

Worth mentioning that the history of sagas that was performed with this optimization will not be stored in cluster itself. It is still possible to track history by querying services that was involved to saga, but such process could be very complex due to data of different services is isolated in microservices architecture. Preservation of only failed sagas could reduce time and memory requirements so that data preservation could be part of saga final process in case of compensations invoked.

III. Modeling

Total time of replication T_R, defined as max replication time of concrete saga that depends on replication coefficient $r(1..)$. It consists of two main parts that creates noticeable delay: data transfer time T_T and preservation time T_P.

$$T_R = \max_{\{r\}}(T_R(r)) \tag{1}$$

$$= \max_{\{r\}}(T_T(r) + T_P(r)) \tag{2}$$

One saga time consumption T_{SAGA} includes replication on every node and local transaction time T_L. Both of them depend on properties of concrete local transaction n.

Replication variables T_T and T_P depends on meaning data coefficient $k(0..1)$, that specifies what part of actual payload D required either to proceed with next saga local transactions or to perform compensation of previous local transactions.

$$T_{SAGA}(r) = \sum_1^n (T_L(n) + T_R(n)) \tag{3}$$

$$T_R(n) = \max_{\{r\}}(T_T(k(n)D(n)) + [T_P(k(n)D(n))]) \tag{4}$$

Compensation should be send to every service that have performed local changes already during saga failure on node f. In this case $\overline{T_{SAGA}}$ should be calculated differently.

$$\overline{T_{SAGA}}(r) = \sum_1^f (T_L(n) + T_R(n)) + \sum_1^{f-1}(\overline{T_L}(n) + \overline{T_R}(n)) \tag{5}$$

Here $\overline{T_L}(n)$ is time to perform compensation call and $\overline{T_R}(n)$ is time to replicate compensation data. Absence of necessity in compensation for f service defines $f - 1$ as upper bound of second sum.

Presence of T_P depends on memory optimization attribute C_M. If it is enabled then T_P defined as optional operation, time consumption on data preservation might be reduced. On the other hand, preserving such data only in random access memory might lead to overflow if orchestration cluster is under the heavy load. C_F marks that data should be persisted after a saga failure if memory optimization used in a system. Algorithm 1 specifies how to define if T_P affects saga time consumption.

Algorithm 1 Preservation time impact on saga time consumption

```
if $C_M$ then
    if $C_F$ and saga failed then
        $T_P$
    else
        0
    end if
else
    $T_P$
end if
```

Memory threshold value M_{TH} for mentioned optimization should be specified for a system to determine is it safe to rely on memory optimization.

Local transaction time consumption T_L does not depend on meaning data coefficient due to request data of service endpoint is constant same as response payload value.

Calculation of meaning data payload and coefficient is not a part of T_{SAGA} due to saga algorithm is a priori determined that means it can be calculated on the saga initialization step.

Memory requirements M_{SAGA} of cluster also depends on meaning data coefficient k and replication coefficient r:

$$M_{SAGA} = r\sum_0^n [\begin{cases} k(n)D(n), & \text{if } !C_M \\ & \text{if } C_M \text{ and } M_{RAM} \leq M_{TH} \\ & \text{if } C_M \text{ and } C_F \\ 0, & \text{otherwise} \end{cases}] \tag{6}$$

Ability to avoid data preservation depends on currently available RAM value M_{RAM}.

In case of M_{TH} exceeds data should be stored to make cluster be able to continue saga with required data during saga iteration. However, such data might be deleted either right after saga finished successfully, or removed by background job. Deletion of stored data right after saga ends simplifies implementation but brings performance impact even if such process performed simultaneously with saga operationing. It can be critical during high load.

Having background job that clears redundant saga data could solve performance issue. Such job could be invoked if cluster load is not high, so having additional process won't affect system performance.

IV. Evaluation

Numeric values of parameters in model vary in a big range and depend on concrete saga implementation and cluster structure.

To determine how application of additional optimizations in saga cluster data flow affects saga time consumption and memory requirements, evaluation of modern cloud RESTful solutions was performed.

A. Source values

As specified in [9], median values of JSON and XML payloads in REST systems are 1545 and 2606 bytes respectively. Median data size $D = 2000$ bytes for services involved in saga was selected to not rely on concrete data representation that could be used in API.

Storage [10] and network [11] benchmarks for cloud systems provides an assessment of network and memory characteristics of services that are deployed in popular cloud provider systems such as Amazon Web Services, Microsoft Azure, Oracle Clouds and IBM Smart Cloud. AWS was picked to define network T_T and memory T_P properties in current research. Throughput for memory was defined as 1 GB/s and network throughput = 125MB/s so that goodput $\approx$ 110MB/s.

Coefficient of meaning data strongly depends on endpoint implementation and on semantic of saga. For instance, from many fields that will be returned in response from service 1 only ID will matter in next commands or compensations. And vice versa, all of the fields in response might be required for either next commands or possible compensations. Due to this reason coefficient of meaning data $k(n)$ for sagas in evaluation varies in range $[0.1, 1]$. Time of local transaction T_L that represents delay between request and response to one service in saga for commands and compensations also depends on service implementation and character of changes in the service. Range of $T_L = [100, 500]$ was defined based on recommendation [12] in 200ms for response delay.

Every command during saga might fail and trigger compensation calls chain that adds additional delay to T_{SAGA}. Probability of command failure empirically defined as $P_F = 0.5$. Minimal amount of commands in each saga $n = 3$ for the reason that having two calls in saga orchestrator is probably redundant complexity and should be transformed to direct call of services [4]. The upper bound of $n = 8$ chosen as one of common values for this property [6]. Increase of upper bound of local transactions in saga leads to accumulation of failure probability and compensation chain length.

Evaluation performed on cluster with replication factor $r = 3$ for 50000 sequentially invoked sagas N.

B. Time evaluation

Result of time evaluation with different optimization represented in Table IV-B. Calculated values shows that the impact of any optimizations related to data shrink that could affect either data transfer or storing time do not carry sufficient weight.

The reason of such low impact is relatively big value of T_L which might include heavy database interactions or even side services communications.

TABLE I. TIME OPTIMIZATIONS OF SAGA

Optimizations	T_{SAGA} reduction %
No saga data save	0.00070
No saga data save, meaning data transfer	0.00093
Failed saga data save	0.00066
Failed saga meaning data save and transfer	0.00089
All sagas meaning data and transfer	0.00026

However, optimization of replication time that takes place before each local transaction and during compensation calls chain significantly improved (fig. 6).

Two first bars ("Failed sagas data", "No save") shows that optimizations related to storage manipulations almost not affects replication time either it was skip of successfully ended sagas or all sagas data preservation. Next three bars brings bigger impact. Each of them implies data optimization. The reason for this is reduction of network load between nodes if meaning data optimization used.

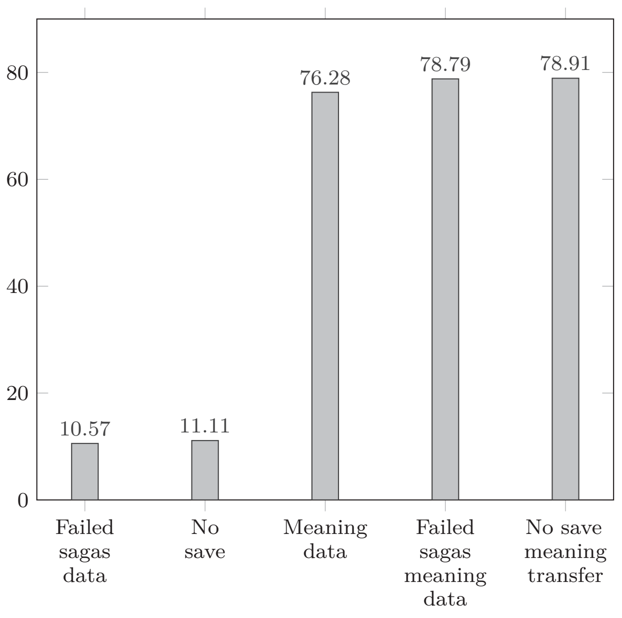

Fig. 6. Time optimizations of replication

Replication factor does not affects amount of optimization due to only bigger delay on replication defines total delay.

It is obvious that the biggest reduction of storage usage appears to be if all optimizations used. But difference in preservation of only failed sagas data and no preservation at all defined as 0.12%.

Overall time optimizations contribution appears to be insignificant against the saga duration. It also shows that drawback of consecutive replication before each command and compensation call can be assessed as insignificant too.

C. Memory evaluation

Relative optimizations of memory requirements depicted in Fig. 7.

Each optimization brings tangible memory requirements reduction. In systems with low saga failures rate, save of only failed sagas makes the biggest reduction relative to other values. This is why difference in current calculation appears to be the maximum difference between all optimization impact values (59.02%).

Application of only meaning data preservation of only failed sagas does not increase difference noticeably. Yet, in case of not only failed sagas data preservation, application of meaning data optimization creates a more noticeable effect.

Relative value of memory optimization in saga cluster does not depends on replication factor, unlike total memory requirements. In such cases usage of both optimizations could be more desired.

Absence of data replication should be considered in case if increase in memory requirements due to cluster structure use is undesirable solution.

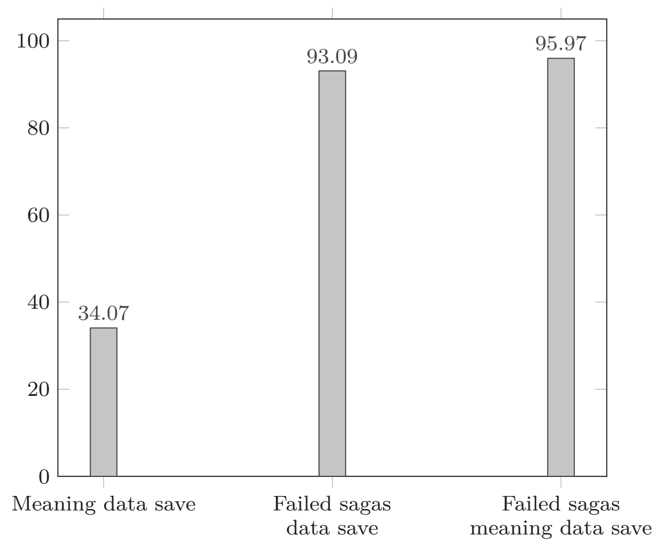

Fig. 7. Memory optimizations of saga

V. Conclusion

Usage of sagas to control microservices in REST fault tolerant system requires to choose one of the next approaches:

- choreography mechanism that couples services and supports both;
- message broker that allows to repeat message retrieve;
- orchestrator build as cluster that sends HTTP requests to idempotent endpoints.

Considering widespread of REST architecture, possibilities of having interchange payload with redundancy and cost of architectural changes, the last option might become preferable. Different optimizations of saga central orchestrator cluster was shown in this paper:

- analysis of response payload from service to replicate and preserve only meaning data;
- optionality of data preservation.

Optimizations can be used together in different degree. Assessment of time and memory parameters based on modern cloud benchmarks values and data structure statistics was performed. It shows relatively low time consumption before every local transactions for replication might be reduced even more. And memory requirement of cluster might be reduced significantly. Thereby application of optimizations during cluster structure implementation in central saga orchestrator can lead to minor losses comparing to orchestrator without fault tolerant structure.

However, having idempotent endpoints might be crucial constraint to use proposed architecture. If REST system requires integration of distributed data consistency support and some of endpoints involved to data modification flow are not idempotent so that modification of them required in any way, we propose to use original event choreography or orchestration to simplify flow organization. Such solution will eliminate possible risks of having difficulties with idempotent endpoints later.

References

[1] *F.* Li, J. Fröhlich, D. Schall, M. Lachenmayr, C. Stückjürgen, S. Meixner, F. Buschmann “Microservice Patterns for the Life Cycle of Industrial Edge Software”, *in Proc. EuroPLoP’18 Conf.*, 2018, art. 4, pp. 111

[2] *G.* Samaras, K. Britton, A. Citron, C. Mohan, “Two-phase commit optimizations and tradeoffs in the commercial evironment”, *in Proc. IEEE Conf.*, April 1993, pp. 17-22.

[3] *J.* Nimis, P.C. Lockemann, K. Krempels, E. Buchmann, K. Böhm. *Towards Dependable Agent Systems.* Springer, Berlin, Heidelberg, pp. 465-501

[4] *C.* Richardson, *Microservices Patterns.* Manning Publications, 2017.

[5] *H.* Garcia-Molina, K Salem, “Sagas”, *in Proc. SIGMOD Conf.*, Dec. 1987, pp. 249-259.

[6] *C.* K. Rudrabhatla, “Comparison of Event Choreography and Orchestration Techniques in Microservice Architecture”, *IJACSA*, vol.9, Issue 8, 2018.

[7] *X.* Limon, A. Guerra-Hernandez, A.J. Sanchez-Garc, J.C.P. Arriaga, “SagaMAS: a software framework for distributed transactions in the microservice architecture”, *CONISOFT*, Oct. 2018

[8] *K.* S. Ahluwalia, A. Jain , “High availability design patterns”, *in Proc. PLoP Conf.*, Oct. 2006, pp. 1-9.

[9] *C.* Rodriguez, M. Baez, F. Daniel, “REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices”, *ICWE*, June 2016, pp. 21-39

[10] *H.* Lee, G.C. Fox, “Fair Benchmarking for Cloud Computing systems”, *IEEE CLOUD*, May 2019

[11] *L.* Gillam, B. Li, J. OLoughlin, A.P.S. Toma, “Fair Benchmarking for Cloud Computing systems”, *Journal of Cloud Computing*, Feb. 2013

[12] *P.*G. Talaga, S.J. Chapin, “Reducing Latency and Network Load Using Location-Aware Memcache Architectures”, *WEBIST*, April 2012, pp 53-69