Data Replication on the Cloud/Edge

David Mealha, Nuno Preguiça, Maria Cecília Gomes, João Leitão
NOVA LINCS & DI/FCT/UNL Portugal
d.mealha@campus.fct.unl.pt,{nuno.preguica,mcg,jc.leitao}@fct.unl.pt

ABSTRACT

This work presents a database replication system capitalising on hybrid cloud/edge infra-structures, which may be used in novel software architectures like microservices' applications. The system aims to reduce the latency perceived by clients performing read and update operations on a database, by locating replicas on edge nodes nearer end users. The replicas' convergence algorithm is based on eventual consistency and presents a novel solution that combines Operations Transformation and CRDTs techniques. The system follows the MongoDB data model and is validated on geographically disperse Amazon AWS data centers simulating edge nodes.

KEYWORDS

replication middleware, weak consistency, cloud/edge computing

ACM Reference Format:
David Mealha, Nuno Preguiça, Maria Cecília Gomes, João Leitão. 2019. Data Replication on the Cloud/Edge. In *6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '19), March 25, 2019, Dresden, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3301419.3323973

1 INTRODUCTION

The surge of generated data by current and emerging applications, accessed from an increasing number of fixed and mobile devices, demand adequate/novel solutions for data management that may support a timely data access and processing from where it may be needed [4, 8]. Examples of data intensive applications range from highly popular web applications prone to peak accesses (like Facebook or Twitter) or bandwidth intensive applications like video streaming and gaming, to emerging applications capitalising on fresh data generated by end devices in the domain of smart cities, autonomic cars, or smart health.

The current trend is hosting applications' services and databases in cloud infrastructures, where the resources are plentiful and can grow on-demand [1]. Yet these infrastructures may be distant from the end user or from where fresh data needs to be interpreted, leading to high response times (latency). Replication can mitigate this effect by placing the same data on different locations (geo-replication) closer to clients (e.g. web/mobile clients) and to applications operating over the data (e.g. smart buildings applications). Edge infrastructures [2, 14], located closer to the client, can also be used to replicate data, leading to hybrid cloud/edge infrastructures. This has the potential to further improve the latency and availability for users.

Orthogonally, new software architectures and patterns are being adopted to improve the applications' maintenance and scalability. *Microservices* [7] is one such example by defining a system/application as being composed of loosely coupled small services, each one managing its own database. Each service may hence scale independently taking advantage of the scalability and elasticity properties of current cloud platforms [1]. The loosely coupled nature and small size of microservices also make them better candidates to support the lighter/faster deployment of applications on the emerging hybrid cloud/edge infrastructures. Microservices may be migrated or replicated, eventually along with their databases, to such resource restricted machines according to the necessities [15].

Problem and Objectives. This paper addresses the problem of replicating a database (associated with a microservice) in a hybrid cloud/edge infrastructure, by keeping replicas in both cloud and edge nodes. The proposed database replication system considers the hierarchical structure of cloud and edge nodes, where some machines may be unable to store a large portion of the data. However, it adopts a Multi-Master replication model, where writes can be submitted and immediately processed by any replica, and are later asynchronously propagated to other replicas. The replication system is validated in the context of a particular microservices application within a hybrid cloud/edge infrastructure. The evolution of the system to a dynamic placement strategy of either full or partial databases is left for future work.

Contributions. Implementation of a database replication middleware supporting weak consistency in a large scale distributed scenario including geo-replicated and edge systems. The replication middleware is connected to the MongoDB database system, a popular NoSQL database, and it was evaluated on the Amazon Web Services (AWS) geo-distributed platform. A novel replicas' convergence algorithm is presented, which combines the techniques of Operational Transformation (OT) and CRDTs to implement an eventual consistency model.

2 RELATED WORK

Geo-replication aims to provide low latency and high availability when accessing data in global services, where clients are spread around the world. A large number of geo-replicated storage systems have been designed in recent years [5, 6, 10]. Some systems [5] adopt a strong consistency model, where the system behaves as if a single replica exists, at the cost of requiring coordination among replicas to execute update operations. Other systems [6, 10] adopt a weak consistency model, where an update can be executed by contacting a single replica. This allows to provide low latency

PaPoC '19, March 25, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6276-4/19/03…$15.00
https://doi.org/10.1145/3301419.3323973

and high availability, but it requires the system to handle concurrent updates. Several approaches have been proposed to handle concurrent updates. The popular *last-writer-wins* policy handles concurrent updates by keeping for each data item, the value of the last update (or the update with the larger timestamp). This approach leads to *lost updates*, as a concurrent update with a smaller timestamp is overwritten by one with a larger timestamp. In contrast, operational transformation (OT) [13] and Conflict-free Replicated Data Types (CRDTs) [12] handle concurrent updates by combining the effects of concurrent updates for implementing a given concurrency semantics. In operation transformation, an operation is transformed in a way that the effect of applying it in a given replica preserves the original intent and guarantee that replicas converge. Operation-based CRDTs are defined in a way that all operation commute, thus guaranteeing that replicas converge independently of the order operations are executed. To this end, CRDTs internally store additional metadata.

In this work, unlike geo-replicated storage systems tat internally implement their replication mechanisms, we propose a middleware to geo-replicate a database. We adopt a weak consistency model and handle concurrent update using a mechanism based on OT. However, we build on ideas used to make CRDT operations commute for transforming the operations.

3 PROPOSED SOLUTION

The database replication system acknowledges the heterogeneous nature of an infrastructure composed of cloud and edge nodes, and their hierarchical structure in terms of capabilities. Cloud nodes are considered homogeneous and high capacity nodes whose communication is supported by high speed networks. Edge nodes are typically heterogeneous and globally spread at large numbers, at the border of the communication network near clients/end devices; they are underpowered and unable to store large volumes of data; and cannot/do not rely on high capacity communication networks.

The solution takes advantage of cloud computing properties in terms of scaling/infinite resources and, at the same time, explores the possibilities of edge computing, in terms of providing a large number of resources closer to clients and data sources with lower latency. The CAP theorem [3] proves that no distributed system can be available and consistent at the same time, in the presence of network partitions that are unavoidable in such hybrid infra-structure. This proposal elects availability in detriment of consistency and supports replica creation at edge nodes following an eventual consistency model. This allows better response times since the replicas do not have to wait for the result of coordination protocols. Concurrent updates may occur within this consistency level (weak consistency).

Figure 1 presents a generic architecture of the proposed solution. The database replication system presents a hierarchical architecture that maps onto the infrastructure with replicas at cloud and edge nodes. Every replica in the system is able to accept write operations, which potentially leads to a higher throughput. To guarantee that all replicas converge to a coherent state, the operations have to be disseminated to the other replicas, via *Message Queue* systems. However, considering the restricted capacity of edge nodes, the cloud node is the one responsible for disseminating a new operation

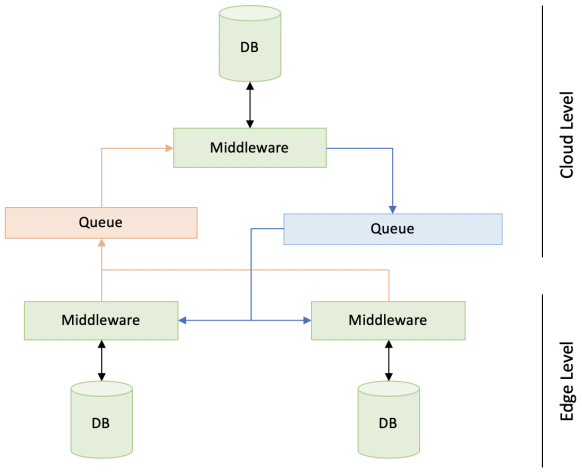

Figure 1: Multiple write replicas in cloud/edge platforms.

to all the replicas in the system. When an edge node receives an new operation locally, this one is persisted and sent to the cloud node to be disseminated to the rest of the replicas.

The example in Figure 1 is composed of three database replicas, each one with its middleware instance to handle replication. There is a strict separation between cloud and edge replicas, following a tree-based architecture, where the former acts as the main replica. Message queues support the replicas' decoupling since they allow each replica to be unaware of the existence of other machines/replicas in the system by avoiding any direct communication among them. New replicas may be independently created near the users where they are needed the most, and locally accessed on read and write operations for faster responses. This supports requests with lower latency and a higher throughput on the number of operations resulting in the improvement of the clients' perceived quality of service (*QoS*) and quality of experience (*QoE*).

The middleware receives the operations from the database's clients and performs the following tasks: *a*) verifies the existence of conflicting operations; *b*) executes the write operation into the database; *c*) adds the operation metadata into the database; and *d*) sends the operation to a message queue for replication.

The message queue system provides a durable pipeline between nodes to push events from one replica to another. The operations are eventually consumed, depending on how fast the middleware can process operations and how many are waiting in the queue. In our system, new operations result from the following cases: *i*) a service/client connects to the edge replica and submits a write operation; *ii*) a service/client connects to the cloud replica and submits a write operation. In the first case, the write operation is processed locally and then sent to the upwards queue. Afterwards, the message is consumed and processed by the cloud replica and, finally, disseminated to all replicas through the downwards queue. The downwards flow is represented by the consumption of the new operation by all edge replicas. This means that all the replicas that have not executed the operation will persist it. The second case behaves exactly as the previous one but, since the operation is submitted at the cloud level, the middleware only needs to send the processed operations to the downwards queue. They will be

consumed at some point in time by all edge replicas. This approach provides causal consistency [10] if the client write operation is processed locally before returning to the client.

3.1 Replication Protocol

The replication protocol presented in this work aims to guarantee that all replicas eventually converge to the same state, while preserving and merging as many concurrent updates as possible. Each update is identified with an unique tags based on a realtime clocks, and when two concurrent updates are conflicting and it is impossible to preserve the effects of both, we adopt a *last write wins* (*LWW*) policy by keeping the effects of the last update.

To implement this approach we adopt a concurrency control mechanism that combines ideas from *Operational Transformation* (*OT*) [13] and *Conflict-free Replicated Data Types* (*CRDTs*) [12].

3.1.1 Data Model. The replication system is not database agnostic but depends on a particular data model, namely the MongoDB data model [11]. Due to its characteristics and common use on microservices applications, supporting the replication of MongoDB instances at edge nodes was selected as the first step towards a more generic replication middleware to be developed in the future.

MongoDB supports the following operations which are also typically available in relational and NoSQL databases:

INSERT addition of a new object/record into an entity/table/document;

UPDATE addition, removal and/or update of specific fields of an object/record;

REPLACE replacement of an object/record;

DELETE remove of an object/record.

In this work an UPDATE operation is defined as a dictionary of updated fields and a vector of deleted fields. For instance:

```
{  "updatedFields" : {
      "firstName" : "Johny",
      "userName" : "JohnyDoe71" },
   "removedFields" : [ "lastName" ] }
```

A REPLACE operation is a (full) object to be used to replace the current one in the collection:

```
{  "fullDocument" : {
      "firstName" : "Johny",
      "userName" : "JohnyDoe71" } }
```

3.1.2 Conflict Detection. The middleware at each node listens to two types of events, *localEvent* and *receivedEvent*. The reception of a *new local event* occurs when a service/client requests a write operation to a database. A *received event* happens when an operation is disseminated by another replica. All events are labeled using the *Hybrid Logical Clocks* (*HLC*) algorithm [9]. Thus, when a new label need to be generated, the current physical clock is used, unless the current clock is smaller or equal to a previously generated or received timestamp. In such case, the label will be equal to the largest timestamp generated or received plus one. HLC guarantees the happens-before property, making it also possible to establish a total order of operations even when physical clocks are not synchronised. This allows to combines the advantages of physical and logical clocks to find conflicting events in globally distributed databases.

Algorithm 1 Replication Manager - Variables and Local Event

```
1:  Data:
2:  documents: last logical time for each object
3:  cache: cached operations for each object

4:  upon LocalEvent(operation) do
5:      objectId ← operation[objectId]
6:      newPhysicalTime ← getPhysicalTime()
7:      latestLogicalTime ← getLatestLogicalTime(objectId)
8:      newLogicalTime ← findMax(latestLogicalTime,
9:      newPhysicalTime)
10:     if newLogicalTime = latestLogicalTime then
11:         newLogicalTime ← newLogicalTime + 1
12:     end if
13:     call updateLocalStorage(objectId,newLogicalTime,operation)
14:     return { operation, newLogicalTime }
```

To detect conflicts, the middleware compares the HLC values of the new operation (either received from a local or external event) and the HLC value of the last persisted message into the system. An operation is defined as concurrent when the logical time value present in a new operation is inferior to the registered time of the last persisted operation. The protocol stores in memory the previously executed operations identified by the object id. This permits a cached snapshot of the database allowing the middleware to merge any new operation into a coherent state, as if the operations arrived in a total temporal order.

Algorithm 1 presents the relevant data structures and the *localEvent* procedure. The algorithm uses two data structures, *documents* and *cache*. The first one stores the *HLC* value of the last persisted operation for each object/record identified by its id. The second one stores all the operations received for each object/record in a temporal order (as a system snapshot). The *LocalEvent* procedure is executed every time a client/service requests an operation to a database replica. The algorithm retrieves the current physical time and the last persisted logical time for that object. Then, the maximum between these values is chosen to be the new logical value for the operation. In case the new logical value equals the previous one, the value is incremented so that it can be considered to be the most recent operation in that machine.

Algorithm 2 shows the pseudocode for the *receivedEvent* procedure. In this procedure, the operation is received from another replica, which means the operation logical time was already obtained in the other machine. Therefore, the last (most recent) logical time for that object id in the system is compared with the newly received one. In case the operation logical time is smaller, it means the operation has occurred in a previous point in time, triggering the merging process. The new operation object is mutated in order to preserve all the effects, as if the operations had been received in a total order. Otherwise, the new operation occurred in a posterior point in time (compared to the last persisted operation) and those are the only effects to be seen in the database. Also, the operation to be added into the cache structure must be the operation before the merge process, in order to maintain the snapshot that may be used in the next merging processes.

3.1.3 Conflict Resolution Protocol. The algorithm implementing the eventual consistency model is based on Operational Transformation (OT) and CRDTs, and it is applied whenever there is a conflict between two write operations, namely INSERT, UPDATE, REPLACE

and DELETE. Namely, the middleware performs a merge transformation in order to combine concurrent updates into a coherent state. The transformation may occur more than once, depending on how many operations are conflicting (the amount of operations with a higher logical value than the newly received). Fundamentally, the merge has to be executed one by one in an ascending order. As an example, consider the following two events over the same object that occur in the listed order:

(1) an Update operation is persisted at replica A (*T2*)
(2) replica B performs a new Update that has to be propagated to replica A (*T1*)

Each UPDATE operation is tagged using logical clocks, namely *T2* and *T1*. Operation identified as T1 has a logical clock with an inferior value than the one present at operation T2, i.e. in global logical terms, operation T1 occurred earlier in time than T2. This means that the first event at the object has a logical clock superior than the second event occurring at the same object.

A particular example of operation T2 may be:

```
{  "updatedFields" : {
      "firstName" : "Johny",
      "email" : "john.doe@gmail.com" },
   "removedFields" : [ "lastName" ] }
```

A particular example of operation T1 may be:

```
{  "updatedFields" : {
      "firstName" : "Johny",
      "userName" : "JohnyDoe71",
      "lastName" : "Doue" },
   "removedFields" : [ "email" ] }
```

In this case, operation T2 had been applied but in face of operation T1 it has to be merged considering that ideally T1 should have had occurred first. The result of merging the two operations applying the LWW policy is the following:

- The "firstName" field is updated by both T1 and T2, and T2 wins. This modification was already applied and so it can be ignored.
- The "userName" field is present in T1 for update but is nonexistent in T2, so it has to updated in the object.
- The "lastName" field is present in T1 for update but it is also present in T2 as a field to be removed in "removedField". Since this field was already removed from the object in the operation T2, and T2 wins wrt T1, this update operation in T1 is ignored.
- The "email" field is present in T1 for removal in "removedFields" but appeared has an update field in T2, and it has already been applied to the object. In this case, we select to preserve the operation of T2 (LWW) meaning that the "email" field is not removed from the object, and so no operation has to be done in this case.

Algorithm 2 Replication Manager - External Event

```
1:  upon ReceivedEvent(fullObject) do
2:      objectId ← fullObject[objectId]
3:      operation ← fullObject[operation]
4:      opLogicalTime ← fullObject[createdAt]
5:      latestLogicalTime ← getLatestLogicalTime(objectId)
6:      if opLogicalTime > latestLogicalTime then
7:          callupdateLocalStorage(objectId,opLogicalTime,operation)
8:          return { operation, opLogicalTime }
9:      else
10:         result ← handlePastOperation(objectId, operation,
11:         opLogicalTime)
12:         callupdateLocalStorage(objectId,opLogicalTime,operation)
13:         return { result }
14:     end if
```

Algorithm 3 Replication Manager - Apply All Operations

```
1:  procedure applyAllOperations(listOperations, operation)
2:      newOperation ← operation
3:      listOperations ← sort(listOperations)
4:      index ← 0
5:      if newOperation.operationType = DELETE then
6:          hasInsert ← findInsert(listOperations)
7:          if hasInsert ≠ TRUE then
8:              return operation
9:          end if
10:     end if
11:     for op ← listOperations do
12:         if op.operationType = UPDATE then
13:             newOperation ← applyUpdate(newOperation, op)
14:         end if
15:         if op.operationType = REPLACE ∨ op.operationType =
    INSERT then
16:             newOperation ←
17:             applyReplace(newOperation, op)
18:         end if
19:         if op.operationType = DELETE then
20:             result ← applyDelete(newOperation,
21:             listOperations, index)
22:             newOperation ← result[newOperation]
23:             if result[hasInsert] then
24:                 index ← result[newIndex]
25:             else
26:                 break
27:             end if
28:         end if
29:         index ← index + 1
30:     end for
31:     return newOperation
32: end procedure
```

The result of the operation merge is that the effective transformation/update to be applied is simply:

```
{  "updatedFields" : {
      "userName" : "JohnyDoe71" } }
```

Conflict Resolution Pseudocode. Algorithm 3 presents a high-level pseudo code of the merging algorithm that is proposed to solve conflicts. The procedure *applyAllOperations* is executed every time a new operation is detected as conflicting. The procedure signature includes two arguments, the new operation (*newOperation*) and all the operations with a higher timestamp (*listOperations*).

The first assertion made by the procedure is to verify if the new operation is a DELETE. If affirmative and there is not a single INSERT in the *listOperations* variable, the merge is stopped and the DELETE operation is returned to be executed (the protocol only overrides a DELETE in the presence of later INSERT operations). Otherwise, a loop is made over all the operations inside *listOperations* and, for each operation, a merge transformation is performed. In the occurrence of a DELETE inside the *listOperations*, a verification is made to check if it contains any subsequent insertions. If true, the loop index is updated to the INSERT's index and the merge process restarts from that point.

The *applyUpdate* procedure merges an UPDATE operation (*listedOperation*) with the newly received operation (*newOperation*). For instance, considering two concurrent updates, the *updatedFields*

Algorithm 4 Replication Manager - Apply Update

```
1:  procedure applyUpdate(newOperation, listedOperation)
2:      newOperationType ← newOperation.operationType
3:      if isUpdate(newOperationType) then
4:          newOperation ← addUpdatedFieldsReplace(newOperation,
5:          listedOperation)
6:          newOperation ← addRemovedFieldsReplace(newOperation,
7:          listedOperation)
8:          newOperation.operationType ← UPDATE
9:      end if
10:     if isReplace(newOperationType) ∨
    isInsert(newOperationType) then
11:         newOperation ← addUpdatedFields(newOperation,
12:         listedOperation)
13:         newOperation ← addRemovedFields(newOperation,
14:         listedOperation)
15:         newOperation.operationType ← REPLACE
16:     end if
17:     return newOperation
18: end procedure
```

Algorithm 5 Replication Manager - Apply Replace

```
1:  procedure applyReplace(newOperation, listedOperation)
2:      newOperationType ← newOperation.operationType
3:      if isUpdate(newOperationType) then
4:          newOperation.object = listedOperation.object
5:          newOperation = addUpdatedFields(newOperation,
6:          listedOperation)
7:          newOperation.operationType ← REPLACE
8:          return newOperation
9:      end if
10:     if isReplace(newOperationType) ∨
    isInsert(newOperationType) then
11:         listedOperation.operationType ← REPLACE
12:         return listedOperation
13:     end if
14:     return newOperation
15: end procedure
```

from the *listOperation* variable is added to the *updatedFields* of *newOperation*. However, when these fields are added, the same fields may be present in the *removedFields* of *newOperation*, and they have to be removed from that data structure. The same behaviour happens when the *removedFields* values are added, the protocol must consider that these fields may be present in the *updatedFields* of the *newOperation*. This issue does not happen when merging with a REPLACE or an INSERT, since these operations do not have the *updatedFields* and *removedFields* structures. Instead, they just provide a full object that represents the new state. Therefore, the addition and removal of fields are done over the state object, resulting in a new object with the effects of the update operation.

The *applyReplace* procedure is transversal to the merge of REPLACE and INSERT operations, since both operations represent exactly the same, by providing a full object as a new state. When the newly received operation is an UPDATE, the transformation consists on adding the non-present *updatedFields* values to the object. On the other hand, when we have to merge REPLACE and INSERT operations, the decision is done by returning the most recent object (the one already persisted and stored in *listedOperation*).

At last, the execution of the *applyDelete* procedure always returns the DELETE operation, unless there is an INSERT operation with a higher timestamp. In the occurrence of an insertion, the procedure returns the *newOperation* variable with the state object and the new index from the *listOperations*. The return of the new index is necessary in order to force the restart of the merging process from the INSERT's operation index.

4 PROTOTYPE AND EVALUATION

This section presents a preliminary evaluation aiming to assess whether the proposed approach improves the experience of the client, by evaluating: the latency perceived by clients performing read and write operations, i.e. if there was an improvement on the perceived latency at a local edge node; the overhead of the implemented middleware in the overall system; how scalable is the proposed solution; and the delay of the write operations propagated to remote replicas.

4.1 Prototype

Our prototype was implemented as a middleware that replicates MongoDB databases. An independent MongoDB instance runs in each cloud and edge locations, and it is augmented with our middleware that controls the execution of local and remote updates. Clients connect to our middleware, which acts as a proxy of the database, implementing the MongoDB interface.

For reliably propagating operations among replicas, our middleware relies on Apache Kafka[1], which runs on the cloud infrastructure. Thus, each edge node send and receive updates by connecting to the cloud infrastructure. For each edge node, there is an ingoing queue with operations not yet processed by the edge node.

4.2 Test Infrastructure and Architecture

The evaluation considers a setting similar to the one of Figure 1. The system was deployed in AWS using three distinct locations, one for the cloud replica and two for the edge nodes: Cloud node – Dublin; Edge nodes and clients – Frankfurt and Paris.

There are two testing scenario. The first one, *Cloud*, is the representation of what happens nowadays. The application service and its database reside in cloud data centers. Clients perform accesses from remote locations. The second case represents the proposed hybrid *cloud/edge* solution, where clients are co-located with edge nodes. Two important factors were considered as pre-requisites for the evaluation's conditions. The EC2 machines for the edge nodes should have a much lower capacity, and this was achieved by launching a cloud's representative EC2 with 2vCPU and 4 GB RAM and the edge's representative EC2 with 1vCPU and 1GB RAM.

[1] https://kafka.apache.org

Algorithm 6 Replication Manager - Apply Delete

```
1:  procedure applyDelete(newOperation, listOperations, currentIndex)
2:      newOperation.operationType ← DELETE
3:      newIndex ← currentIndex
4:      result ← findInsert(listOperations)
5:      if result.hasInsert then
6:          newIndex ← result.index
7:          newOperation.object ← listOperations[newIndex].object
8:          newOperation.operationType ←
    listOperations[newIndex].operationType
9:      end if
10:     return newOperation, result.hasInsert, newIndex
11: end procedure
```

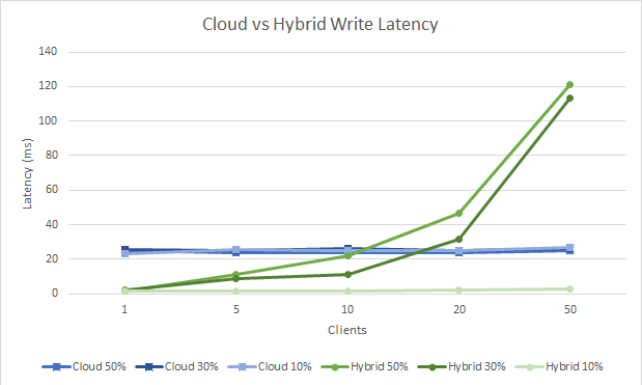

Figure 2: Cloud vs Hybrid write latency.

A loosely coupled synchronisation of clocks between replicas is guaranteed through *Amazon Time Sync Service*.[2].

The client application sends parallel requests to a microservice, in order to mock concurrent accesses from different users. The write operations are either INSERT, UPDATE, REPLACE or DELETE operations. Each type of write is executed 1/4 of the times of the write rate. For each read request, the client retrieves a single record from the database based on a random identifier.

4.3 Results

The following sections compare the results of throughput and latency between the cloud only solution and the hybrid cloud/edge proposal. The dissemination average time is also presented, which represents the amount of time it takes for a write operation to be visible in another replica.

4.3.1 Latency. The first collected metric was the response time per request, in order to acknowledge if the location of a replica closer to the end user has any significant improvements.

Our solution shows a decrease of the write latency, when operations are performed in tests that have up to a maximum of ten (10) concurrent clients. As shown in Figure 2, the results do not exceed a single-digit value, in case of few concurrent users. However, when the machine is overloaded with a high amount of clients, the middleware does not guarantee acceptable values.

The values for the read operations are shown in Figure 3. In this type of operations the overhead of the write operations' propagation is absent and, consequently, the result is lower/better latency values perceived by clients. Namely, a replica at an edge node, with a much lower capacity than the cloud node, can serve requests faster in the presence of up to twenty (20) concurrent clients.

4.3.2 Throughput. A replica's throughput, presented in Figure 4, is an additional metric in our evaluation that shows how many operations a node can process per time unit, while varying the write rate. The results highlight how quickly the hybrid solution reaches its peak performance. Since the latency between the clients and the server at an edge node is so low, the machine quickly reaches its limit in terms of its processing capacity. In a nutshell, the edge node is capable of serving a maximum of one thousand and two hundred (1200) requests per second. The reason for reaching

[2]https://aws.amazon.com/pt/blogs/aws/keeping-time-with-amazon-time-sync-service/

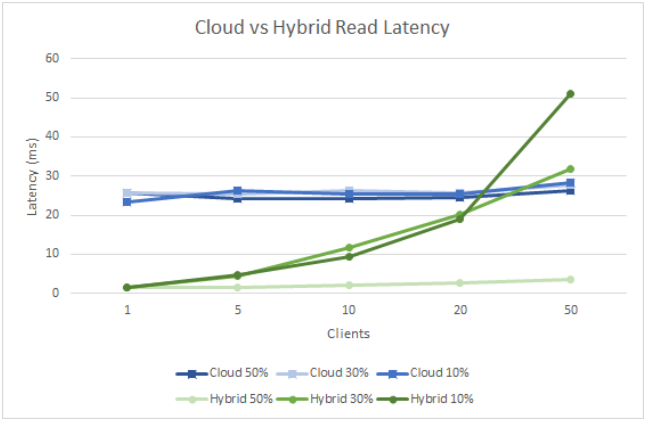

Figure 3: Cloud vs Hybrid read latency.

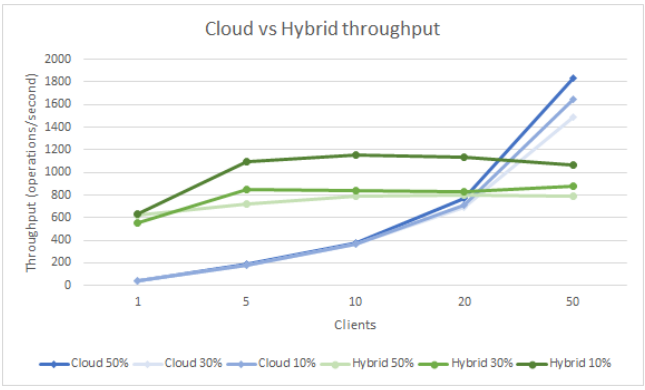

Figure 4: Cloud vs Hybrid throughput.

this limit so quickly is that the operations arrive faster than the necessary time to process them totally, since it is only able to satisfy one request at a time (only one thread per edge node).

4.3.3 Dissemination times. The last metric is the delay between performing an operation at a node and the time it becomes visible at the remote nodes. It was defined by measuring how long it takes for a message originated in Frankfurt (Edge) to arrive at Dublin (Cloud) and at Paris (Edge). On average, the operations from Frankfurt to Dublin took over two hundred (200) ms and an additional thirty (30) ms from Dublin to Paris. This shows that the latency introduced by our solution is reasonable.

5 CONCLUSIONS AND FUTURE WORK

This work presents a database replication system capitalising hybrid cloud/edge infra-structures that may be used in novel software architectures like applications based on microservices. The system is similar to a multi-master replication strategy implementing an eventual consistency model with all replicas accepting read and write operations. The architecture follows a hierarchical organisation with a master replica responsible for disseminating the updates received from a particular replica. The replication system is associated with the MongoDB data model and implements a novel consistency replication protocol based on Operation Transformation (OT) and Conflict-free Replicated Data Types (CRDTs).

The proposal was validated in the context of a particular microservices application use case, a sock shop, and the evaluation presents promising results towards building an automatic database replication system in the future. To this extent, the next steps consist on supporting database sharding on edge nodes, defining/implementing a generic middleware capable of supporting diverse databases and their data models, and supporting dynamic replication to both full and sharded databases, both at cloud and edge nodes.

Acknowledgments

This work was partially supported by the European Union H2020 LightKone project under grant 732505 (https://www. lightkone.eu/) and FCT/MCTES grants UID/CEC/04516/2013 and Lisboa-01-0145-FEDER-032662 /PTDC/CCI-INF/32662/2017.

REFERENCES

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58. https://doi.org/10.1145/1721654.1721672

[2] Kashif Bilal, Osman Khalid, Aiman Erbad, and Samee U. Khan. 2018. Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers. *Computer Networks* 130 (2018), 94 – 120. https://doi.org/10.1016/j.comnet.2017.10.002

[3] E. Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (Feb 2012), 23–29. https://doi.org/10.1109/MC.2012.37

[4] C.L. Philip Chen and Chun-Yang Zhang. 2014. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. *Information Sciences* 275 (2014), 314 – 347. https://doi.org/10.1016/j.ins.2014.01.015

[5] James C. Corbett and et. al. 2013. Spanner: Google&Rsquo;s Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[7] Martin Fowler and James Lewis. 2014. Microservices, a definition of this new architectural term. martinfowler.com/articles/microservices.html

[8] Martin Kleppmann. 2017. *Designing Data-Intensive Applications The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly.

[9] Sandeep Kulkarni, Murat Demirbas, Deepak Madeppa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases (under submission). (2014). https://cse.buffalo.edu/tech-reports/2014-04.pdf

[10] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don'T Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 401–416. https://doi.org/10.1145/2043556.2043593

[11] MongoDB. 2017. MongoDB Architecture. https://www.mongodb.com/mongodb-architecture

[12] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

[13] Chengzheng Sun and Clarence Ellis. 1998. Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work (CSCW '98)*. ACM, New York, NY, USA, 59–68. https://doi.org/10.1145/289444.289469

[14] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. 2016. Challenges and Opportunities in Edge Computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. 20–26. https://doi.org/10.1109/SmartCloud.2016.18

[15] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. 2016. Osmotic Computing: A New Paradigm for Edge/Cloud Integration. *IEEE Cloud Computing* 3, 6 (Nov 2016), 76–83. https://doi.org/10.1109/MCC.2016.124