Effective Replica Management in Car Navigation System using Edge Cloud Environment

Lavish Gulati **Vakul Gupta**
Indian Institute of Technology Guwahati
{gulat170123030, vakul170101076}@iitg.ac.in

Abstract

Car navigation system is nowadays based on centralized cloud architectures. However, edge cloud computing provides robust computational and storage resources improving response times, system scalability and data reliability. The multi-replica strategy used in edge cloud computing architecture can create multiple data replicas and store them in different edge nodes, which improves data availability and data service quality. Due to time-varying user demand, the number of data replicas need to be dynamically adjusted. Load balancing is also required while placing the newly added replicas. We also consider the problem of data replica synchronization and data recovery in case of failures. We propose an optimization problem based on the performance of the edge cloud computing architecture in car navigation system, and improve upon the existing replication algorithms. We also simulate the algorithms on a sample edge cloud network to obtain the results. The simulation code is given in this GitHub repository.

1 Introduction

A car navigation system (1) uses a satellite navigation device to collect various information such as origin-destination stations, occupancy of vehicles, pedestrian activity trends, and more. The on fly traffic information collected from various such navigation systems can be used to plan the optimal path to some destination and reduce traffic congestion.

Edge computing is a distributed computing paradigm that brings computation and data storage closer to the location where it is needed (in our case the car users), to improve response times and save bandwidth.

The architecture of the car navigation system is divided into three layers: centralized cloud layer, edge cloud layer and client layer. The client layer includes the cars equipped with navigation devices and an application interface. The edge cloud layer includes the computing nodes and data centers located near the clients. The centralized cloud layer includes the central cloud computing server and the nodes monitor.

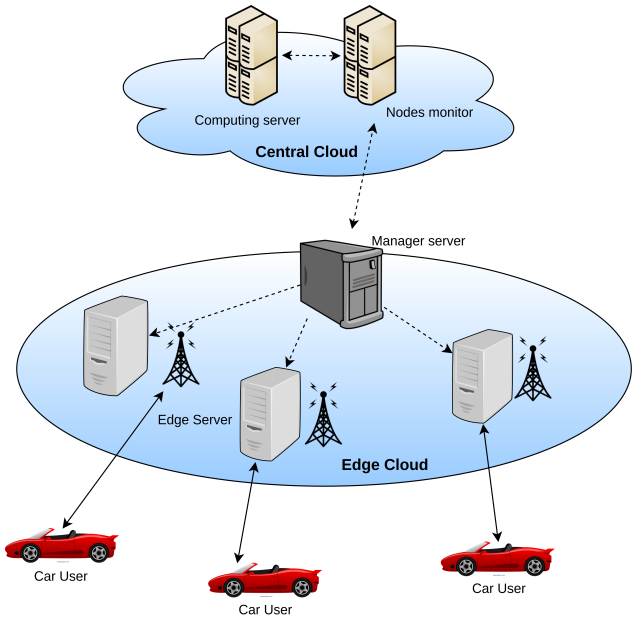

Figure 1: Navigation system in edge cloud architecture

Edge computing devices provide computational and storage resources to the clients. The data is first transmitted to the edge node for local processing, which reduces network traffic and response times (2; 3). Since the computing nodes are located over a large geographical region, failures or outages might interrupt the service to the clients which can be solved by data replication approach (4). Data replication is the process in which the data is copied at multiple locations (different edge servers) to improve data availability and data service quality.

The number of data replicas needs to be dynamically adjusted because of the time-varying data and computation requests. The continuously changing number of replicas leads to the requirement of placing the newly added replicas on edge nodes. Replica placement (5) involves identifying the best possible node to duplicate data based on network latency and user request.

We also need to ensure data synchronization between two or more edge nodes and update changes automatically between them to maintain consistency. This assures congruence between each source of data and

its different endpoints. So, in order to ensure accurate, secure, compliant data and successful end-user experience, data synchronization is required among different edge nodes.

To protect volume data from unrecoverable failure, we need to replicate a volume to a replica node. Data loss occurs due to crashing, correlated failure, logical failure, power outages and security threats. We can recover the data from the replica node once the issue has been rectified. In such scenario, it is important to resume data availability as soon as possible to prevent or limit application downtime.

The rest of the paper is organized as follows: The objective and problem formulation are discussed in Sections 2 and 3 respectively. The experimental setup is discussed in Section 4 which comprises of structure of the edge cloud network (4.1), socket programming (4.2), multi-threading in edge node (4.3) and synchronization using threading lock (4.4). The implementation is discussed in Section 5 which comprises of dynamic replica creation strategy (5.1), replica placement strategy (5.2), replica synchronization strategy based on replica delayed update (5.3) and replica recovery strategy based on load-balancing (5.4). The observations and results are discussed in Section 6. The conclusion and future work are discussed in Section 7.

2 Objective

To study the performance of edge cloud computing architecture using effective replica management in car navigation system based on different parameters like response time and load balancing capabilities, and in the process maintaining data consistency using synchronization and the capability of the system to recover from failures.

3 Problem Formulation

Suppose we have an edge cloud architecture A with m central nodes given by $\{C_1, C_2, \ldots, C_m\}$ and n edge nodes given by $\{E_1, E_2, \ldots, E_n\}$. The car navigation system data contains information about road network and traffic intensity of different localities. The network is represented as a directed graph where each edge denotes a unidirectional road and weight of the edge denotes the traffic intensity on that road.

The data of each locality is considered as a single data block. Thus, we get a set of d data blocks containing the entire information for all the localities denoted by $(D_1, D_2, \ldots, D_d)$, where D_i represents the data block of the ith locality. The car user queries for the optimal path between the current location and the desired destination in a specific locality via a client-end application interface. The query for the specific locality is received by the nearest load balancer, which is then re-routed to one of the edge nodes containing data block of that locality. The edge node solves the shortest path query using Dijkstra's algorithm (10).

The dynamic replica creation strategy finds the optimal number of replicas $H_{i,t}$ of the ith data block at time period t.

For replica placement, we assign a binary encoding B_j (Fig. 2) to each edge node E_j whose length is same as the number of data blocks d and the ith bit in the encoding takes a value of 1 or 0, which means whether a replica of ith data block is stored in the edge node or not. The set of n such encodings for all the edge nodes determine where the replicas are placed in the edge cloud system.

The encodings should optimize the multi-objective replica placement performance indicator function $S(d)$ under the constraint that the optimal number of replicas of ith data block is $H_{i,t}$ which can be modelled as

$$\begin{aligned} \sum_{j=1}^{n} B_{j,1} &= H_{1,t} \\ &\vdots \\ \sum_{j=1}^{n} B_{j,i} &= H_{i,t} \\ &\vdots \\ \sum_{j=1}^{n} B_{j,d} &= H_{d,t} \end{aligned} \tag{1}$$

where $B_{j,i}$ represents the ith bit of the binary encoding B_j.

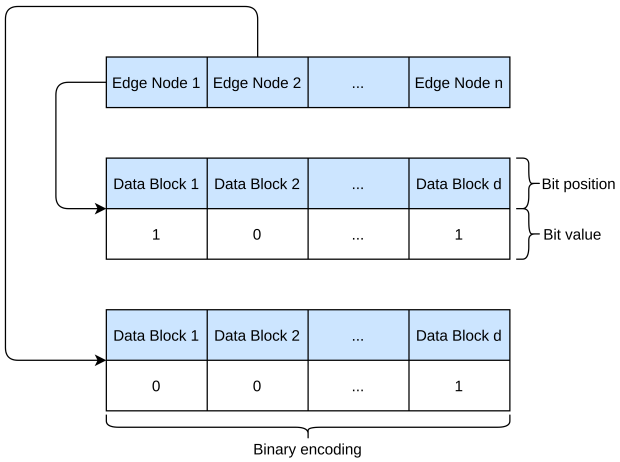

Figure 2: Binary Encoding Scheme

4 Experimental Setup

4.1 Structure of the edge cloud network

The experimental setup consists of two personal laptops and two lab computers which act as "servers".

Due to hardware limitations, we only deploy four edge nodes; one on each server with the corresponding car user on the same server. We also use another lab computer on which Central node and Collector node are deployed. All components are deployed as Docker containers which provides an isolated environment for the execution of the script. The sample network is shown in Figure 3.

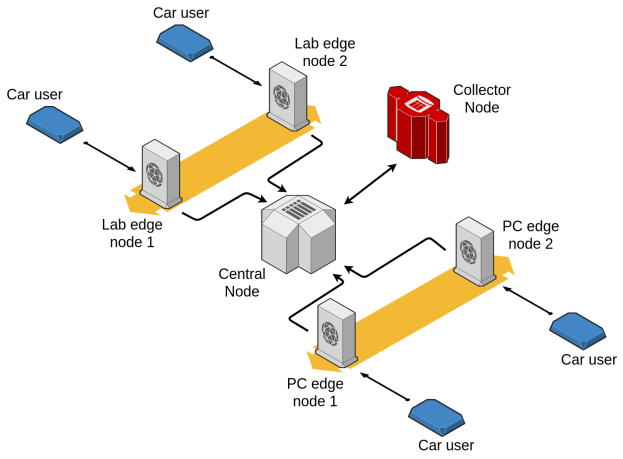

Figure 3: Experimental setup of edge cloud network

4.2 Socket Programming

We extensively use socket programming for inter-container communication. Inter-container communication is required for sending queries to the edge node, returning output of the queries, sending statistics to the central server, and sending heartbeat packets to the Collector node. TCP connection is established between the sending server and receiving server. The data is defined as a JSON object, which is then encoded into bytes before initiating the transfer.

There are two properties which can be associated with each server:

1. Sender
2. Receiver

and both of them are handled differently.

4.2.1 Sender

The sender establishes a single TCP connection each time a JSON object is to be sent. This ensures better network utilization, but also takes more time in establishing each TCP connection. However, since in a practical setting where each car user sends queries spatially in time, this approach performs better. We use the Python library **socket** for the implementation.

4.2.2 Receiver

Since there can be multiple incoming connection requests on the receiving server, we need to queue the requests to avoid connection failure. For this, we use a Python library **selectors** which does not block multiple incoming requests. The TCP connections are established and served sequentially till there are no more pending requests.

4.3 Multi-threading in edge node

We also use multi-threading in edge node to facilitate the sending of heartbeat packets to the Collector node. One thread is used to send heartbeat packets periodically every $HEARTBEAT\_PERIOD$ seconds. The second thread is used to handle the other tasks assigned to the edge node. We use Python library **threading** for this.

```
Thread(target = connect_to_clients).start()
Thread(target = send_heartbeat_packet).start()
```

4.4 Synchronization using threading Lock

Due to multi-threading and presence of global variables, we require locks to ensure exclusive reads and writes to the variables. This is achieved using the Python synchronization primitive **threading Lock**. Once a thread has acquired the lock, subsequent attempts to acquire it are blocked, until the lock is released.

5 Effective Replica Management in Edge Cloud Environment

For simulation, the data is assumed to be an array of random integers represented by $DATA$, with $DATA\_BLOCK\_LENGTH$ number of cells acting as a data block. This is because we can encode any type of data (like graphs in this case), to JSON objects which can be further converted to byte array as explained previously. The byte array is itself an array of integer-represented binary values which is analogous to an integer array and hence, we need not dwelve deep into the exact value of data being considered.

The car user sends queries to the edge node. There are two types of queries: read and update. The queries are sent in the form of IDs of the data blocks requested or to be updated.

The read query asks for the values of the data blocks, given the IDs of the data blocks, from the corresponding edge node. If a specific data block is not present in the edge node, it returns a value of -1. Read query is indicated by $type = 0$.

The update query instructs the edge node to update the data blocks given the IDs of the data blocks and the updation values. Update query is indicated by $type = 1$.

```
data_blocks = list of randomly generated
  IDs of data blocks
type = random choice between 0 or 1
```

```
data = {}
if type == 0:
    data = {"data_blocks": data_blocks,
      "type": 0}
else:
    values = randomly generated updation
      values
    data = {"data_blocks": data_blocks,
      "type": 1, "values": values}
```

5.1 Dynamic replica creation strategy

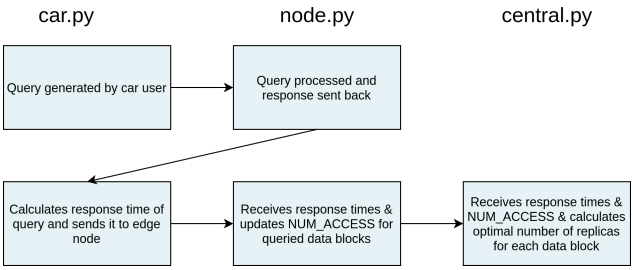

Figure 4: Dynamic replica creation strategy

The car user calculates the response time of the q^{th} query as RT_q.

```
queryStartTime = time.time()
# execution of qth query
queryEndTime = time.time()
RT_q = queryEndTime - queryStartTime
```

The car user then sends the response time RT_q to the edge node. Hence, for each query, there are two TCP connections established: one for sending the query and receiving the results of the query, and other for sending the response time of the query.

The edge node receives the response time RT_q and also updates the number of accesses of data blocks specified in the query q.

```
lock.acquire()
for d in requested data blocks of query q:
    RT[d] = RT_q
    NUM_ACCESS[d] += 1
lock.release()
```

The edge node sends the data (RT and NUM_ACCESS) to the central server after every T/K seconds, where T is the total time period of the clock and K is the number of intervals in the time period.

The central server collects the RT and NUM_ACCESS data from all the edge nodes.

```
for d from 0 to NUM_DATA_BLOCKS - 1:
    if at least 1 access of data block d:
        F[k][d] += NUM_ACCESS[d]
        ART[k][d] += RT[d]
```

which it uses to calculate the optimal number of replicas for each data block d. The optimal number of replicas of a data block, OPT_d, is directly proportional to the heat of the data block $HEAT_d$. In the original algorithm, $HEAT_{d,t}$ depended only on the immediate history $h_{d,t-1}$. However, sudden spike in queries could lead to unnecessary increase in heat which could alter the performance of the system. So we take history upto k intervals of time, giving more weightage to the nearer history. Let $\beta_1, \beta_2, ..., \beta_k$ be positive real numbers such that $\beta_1 \geq \beta_2 \geq ... \geq \beta_k$ and $\sum_{i=1}^{k} \beta_k = 1$. Then the history is defined as

$$hist_{d,t} = \beta_1 h_{d,t-1} + \beta_2 h_{d,t-2} + ... + \beta_k h_{d,t-k} \quad (2)$$

The original algorithm also fails to capture how the number of accesses of the data block h_d is changing over time. If h_d increases sharply, the increase in the predicted heat should be more because there is a sharp increase in the demand of the data block. Similarly, if h_d increases gradually, the predicted increase in heat should be less. The change in heat at time $t-1$ is defined as

$$\begin{aligned} ph_{d,t+1} &= \frac{dh_{d,t-1}}{dt} * \Delta t \\ &\approx \frac{h_{d,t-1} - h_{d,t-2}}{\Delta t} * \Delta t \\ &= h_{d,t-1} - h_{d,t-2} \end{aligned} \quad (3)$$

To avoid noisy data, we take average of changes over k time intervals.

$$\begin{aligned} ph_{d,t+1} &= \frac{h_{d,t-1} - h_{d,t-2} + ... + h_{d,t-k+1} - h_{d,t-k}}{k-1} \\ &= \frac{h_{d,t-1} - h_{d,t-k}}{k-1} \end{aligned} \quad (4)$$

Finally, we take $k = K$ so that it does not affect the overall time and space complexity of the algorithm. Therefore, $HEAT_{d,t}$ is defined as

$$HEAT_{d,t} = h_{d,t} + hist_{d,t} + ph_{d,t+1} \quad (5)$$

The implementation of the improved algorithm is given below.

```
for d from 0 to NUM_DATA_BLOCKS - 1:
    for k from 0 to K - 1:
        if ART observed in kth interval for
          data block d:
            F[d] += 1
            ART[k][d] /= F[k][d]
            ART[d] += ART[k][d]
    ART[d] /= F[d]

for d from 0 to NUM_DATA_BLOCKS - 1:
    for k from 0 to K - 1:
        H[d] += F[k][d]
    H[d] /= K

for d from 0 to NUM_DATA_BLOCKS - 1:
    HEAT[d] = H[d] +
        (H_PREV[d][1] - H_PREV[d][k]) / k-1
    for k from 1 to K:
        HEAT[d] += BETA[k] * H_PREV[d][k]
```

ART[k][d]	Average response time for data block d in interval k
F[k][d]	Total number of accesses of data block d in interval k
ART[d]	Average response time of data block d
F[d]	Number of time intervals in which data block d is accessed
H[d]	Average access frequency of data block d
PH[d]	Predicted value of average access frequency of data block d at time period $t+1$
HEAT[d]	Heat of data block d
DF[d]	Dynamic factor of data block d
OPT[d]	Optimal number of replicas of data block d

Table 1: Dynamic replica creation strategy

```
for d from 0 to NUM_DATA_BLOCKS - 1:
    DF[d] = HEAT[d]*ART[d]

for d from 0 to NUM_DATA_BLOCKS - 1:
    OPT[d] = ALPHA * OPT_PREV[d] +
    ((1 - ALPHA) * DF[d]) / STATIC_FACTOR
```

The overall time and space complexity of the algorithm is $O(NUM\_DATA\_BLOCKS * K)$.

5.2 Replica placement strategy

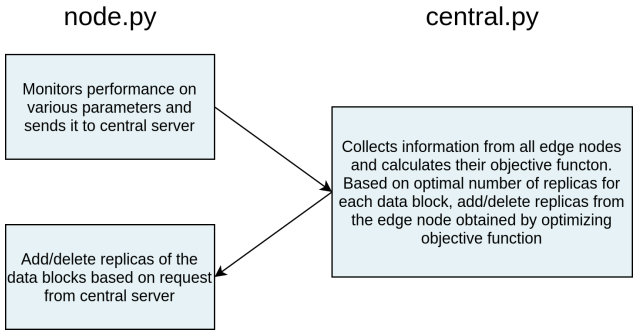

Figure 5: Replica placement strategy

The edge node monitors its load performance on various parameters and sends it to the central server every time period T seconds. We use the Python library **psutil** which retrieves information about system utilization (CPU, memory, disks and network).

Apart from the first three sub-objective functions, we add a new sub-objective function which captures the demand of all data blocks in an edge node. Higher demand in an edge node occurs when there are more number of clients in the locality of that node. In this case, we need to place replicas of more data blocks in that edge node to serve the increased demand. We use the summation of $HEAT$ values of all data blocks in the node as the demand factor. The sub-objective function 4 is formalized as:

$$sub\_objective\_4 = \sum_{d} HEAT_d \tag{6}$$

The implementation of the improved algorithm is given below.

```
# Sub objective 1
uf = psutil.cpu_percent()/100
nc = psutil.cpu_count(logical = False)
fr = psutil.cpu_freq().max
cpu_capacity = fr * nc * (1 - uf)

read_speed = psutil.disk_io_counters().
                read_time/1000
write_speed = psutil.disk_io_counters().
                 write_time/1000
disk_performance = ALPHA * read_speed
         + (1 - ALPHA) * write_speed

mem_usage = psutil.virtual_memory().
            percent/100
mem_size = psutil.virtual_memory().
           total/10**6
load_capacity_memory = mem_size *
                       (1 - mem_usage)

load_capacity_disk = psutil.
                     disk_usage('/').
                     free/10**6

sub_objective_1 = cpu_capacity +
                  disk_performance +
                  load_capacity_memory +
                  load_capacity_disk

# Sub objective 2

f = total number of data blocks stored in the
    edge node

total_disk_space = psutil.disk_usage('/').
                   total/10**6
bsize = uniform size of data block in file
        system
beta = (f * bsize) / total_disk_space

# Sub objective 3
net_dis_coeff = nd / nd_max
Ctr = transmission cost per bit of data per unit
      time
sub_objective_3 = 1 / (net_dis_coeff *
                  data_block_size * Ctr)

# Sub objective 4
sub_objective_4 = 0
for d from 0 to NUM_DATA_BLOCKS - 1:
    sub_objective_4 += HEAT[d]
```

The central server collects the $SUB\_OBJECTIVE$ data from all edge nodes and computes the $OBJECTIVE$ values for each edge node. The IDs of the edge nodes are then sorted according to increasing $OBJECTIVE$ values and stored in $SORTED\_OBJ$.

We consider the weighted average of all sub-objective functions instead of maximizing one of the sub-objective function. Consider the situation where, for edge node 3, its network distance is the smallest ($sub\_objective\_3$ is the largest) compared to other edge nodes but file system cluster load ($sub\_objective\_2$) is comparatively low. In this case, edge node 3 will be selected for placement according to the Pareto-optimal strategy, but this will further increase the file system cluster load on edge node 3 which will result in a decrease in performance. Weighted average ensures that those edge nodes with substantially good sub-objective function values are chosen over Pareto-optimal function values.

```
# Sub objective 2
for j from 0 to NUM_EDGE_NODES - 1:
    BETA_MEAN += BETA[j]
BETA_MEAN /= NUM_EDGE_NODES

for j from 0 to NUM_EDGE_NODES - 1:
    SUB_OBJECTIVE_2[j] = NUM_EDGE_NODES /
                         (BETA[j] - BETA_MEAN)

# Objective function
for j from 0 to NUM_EDGE_NODES - 1:
    OBJECTIVE[j] = WEIGHT_1 *
                   SUB_OBJECTIVE_1[j]
              + WEIGHT_2 * SUB_OBJECTIVE_2[j]
              + WEIGHT_3 * SUB_OBJECTIVE_3[j]
              + WEIGHT_4 * SUB_OBJECTIVE_4[j]

SORTED_OBJ = Sort all edge nodes on the basis
             of increasing OBJECTIVE values
```

Based on $SORTED\_OBJ$ and optimum number of replicas OPT, central server computes which data blocks need to be added or removed from the edge nodes.

```
num_replicas = [0 for d from
                0 to NUM_DATA_BLOCKS - 1]
for d from 0 to NUM_DATA_BLOCKS - 1:
    num = 0
    for j from 0 to NUM_EDGE_NODES - 1:
        num_replicas[d] +=
        BIN_ENCODING[j][d]

rep_added = [-1 for d from
                0 to NUM_DATA_BLOCKS - 1]

for j from NUM_EDGE_NODES - 1 to 0:
    edge_id = SORTED_OBJ[j]
    for d from 0 to NUM_DATA_BLOCKS - 1:
        if (rep_added[d] == -1 and
            OPT_NUM_REPLICA[d] >
            num_replicas[d] and
            BIN_ENCODING[edge_id][d] == 0):
             BIN_ENCODING[edge_id][d] = 1
             rep_added[d] = edge_id

rep_removed = [-1 for d from
                  0 to NUM_DATA_BLOCKS - 1]

for j from 0 to NUM_EDGE_NODES - 1:
    edge_id = SORTED_OBJ[j]
    for d from 0 to NUM_DATA_BLOCKS - 1:
        if (rep_removed[d] == -1 and
            OPT_NUM_REPLICA[d] <
            num_replicas[d] and
            BIN_ENCODING[edge_id][d] == 1):
             BIN_ENCODING[edge_id][d] = 0
             rep_removed[d] = edge_id
```

uf	Usage of CPU
nc	Number of CPU cores
fr	Frequency of CPU
nd	Distance between central server and edge node
nd_max	Maximum distance between central server and any edge node
num_replicas[d]	Current number of replicas of data block d
rep_added[d]	Stores edge node ID j in which replica of data block d is to be placed
rep_removed	Stores edge node ID j in which replica of data block d is to be removed

Table 2: Replica placement strategy

The central server then sends the updation requests to each edge node j.

```
for j from 0 to NUM_EDGE_NODES - 1:
    new_data_blocks = {}
    for i from 0 to len(rep_added) - 1:
        if rep_added[i] == j:
            new_data_blocks[i] = DATA[i]

    for i from 0 to len(rep_removed) - 1:
        if rep_removed[i] == j:
            new_data_blocks[i] = -1
```

The overall time and space complexity of the algorithm is $O(NUM\_DATA\_BLOCKS * NUM\_EDGE\_NODES)$.

5.3 Replica synchronization strategy

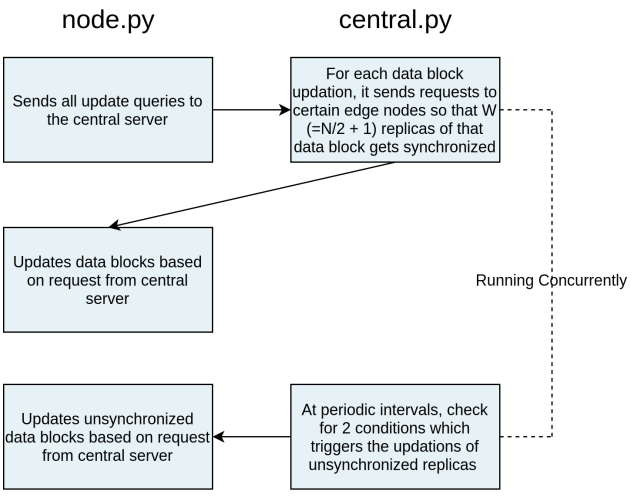

Figure 6: Replica synchronization strategy

Replica synchronization is needed in case of update queries. The edge node forwards the update queries to

the central server, because the main repository of the data is maintained in the central server, while the edge nodes only contain the replicas of the main repository.

The central server updates the main repository with the update requests and triggers the synchronized updates of $W = NUM\_EDGE\_NODES/2 + 1$ replicas. The original strategy selected W edge nodes holding the replica of the requested data block which are closest to the client in terms of network distance. — (i)

However, this strategy does not account for the demand of the data block d in consideration whose update request is observed. The synchronous updates should also be sent to the edge nodes with the highest number of accesses of data block d. ———————————- (ii)

We also send the synchronized updates to the edge nodes closest to the central server to reduce the time for data writing thereby improving the response time of the update query. ——————————— (iii)

Considering the priorities of the improvements required in the edge cloud network, we assign $W/2$ edge nodes for (i), $W/3$ edge nodes for (ii) and $W/6$ edge nodes for (iii) from the total W edge nodes for sending synchronized updates.

```
updates = [[0 for d from
  0 to NUM_DATA_BLOCKS - 1] for j from
  0 to NUM_EDGE_NODES - 1]
for j from 0 to NUM_EDGE_NODES - 1:
    for d from 0 to NUM_DATA_BLOCKS - 1:
        if WRITE_REQUESTS[j][d] == 1:
            for k from 0 to NUM_EDGE_NODES - 1:
                if k < max(0, i - SYNC_W // 2)
                  or k >= min(NUM_EDGE_NODES,
                        j + SYNC_W // 2 + 1):
                    RST[j][d] = 1
                else:
                    updates[k][d] = 1
                    RST[j][d] = 0
Send synchronized update of data block d to
  edge node j wherever updates[j][d] = 1
```

The overall time complexity of the above algorithm is $O(NUM\_EDGE\_NODES^2 * NUM\_DATA\_BLOCKS)$ and the space complexity is $O(NUM\_EDGE\_NODES * NUM\_DATA\_BLOCKS)$.

The asynchronous updates of the $N-W$ replicas are handled after every time period T. The two conditions are checked of whether a data block is hot and whether load performance of an edge node is strong. If any of the two conditions hold, async updates are sent for that data block or edge node, and RST table is updated.

```
hah = 0
for d from 0 to NUM_DATA_BLOCKS - 1:
    hah += H[d]
hah /= NUM_DATA_BLOCKS

alc = 0
for j from 0 to NUM_EDGE_NODES - 1:
    alc += SUB_OBJECTIVE_1[j]
alc /= NUM_EDGE_NODES
```

RST[j][d]	Replica status of d^{th} data block in j^{th} edge node
WRITE_REQUESTS[j][d]	Indicates whether update request is observed for d^{th} data block in j^{th} edge node
hah	Average access frequency at time period t
alc	Average load capacity of all edge nodes at time period t

Table 3: Replica synchronization strategy

```
# Case 1
for d in from 0 to NUM_DATA_BLOCKS - 1:
    if H[d] > hah:
        for j from 0 to NUM_EDGE_NODES - 1:
            if RST[j][d] == 1:
                update_data_blocks =
                  {d: DATA[d]}
                send asynchronous updates
                RST[j][d] = 0

# Case 2
for j from 0 to NUM_EDGE_NODES - 1:
    if SUB_OBJECTIVE_1[j] > alc:
        update_data_blocks = {}
        for d from 0 to NUM_DATA_BLOCKS - 1:
            if RST[j][d] == 1:
                update_data_blocks[d] =
                  DATA[d]
                RST[j][d] = 0
        send asynchronous updates
```

The overall time and space complexity of the above algorithm is $O(NUM\_EDGE\_NODES * NUM\_DATA\_BLOCKS)$.

5.4 Replica recovery strategy

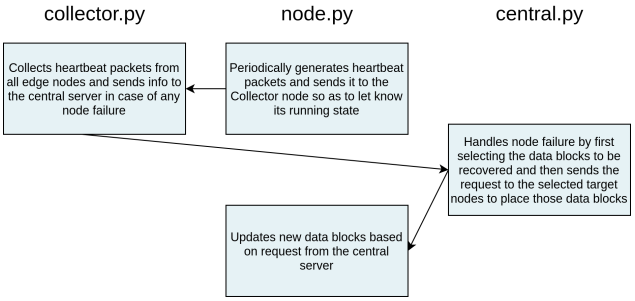

Figure 7: Replica recovery strategy

The edge node sends heartbeat packets containing the ID of the node regularly every $HEARTBEAT\_PERIOD$ seconds. The Collector node collects the heartbeat packets from all edge nodes, and periodically checks for the failure of any edge node. The two tasks of the Collector node are run parallelly in separate threads.

Suppose at a certain point of time, the Collector node receives the heartbeat packets of edge nodes $\{1, 3, 4\}$. There are two indecisive cases:

1. Heartbeat packet of edge node 2 is yet to be received. This might be due to TCP connection failure, network congestion or packet loss. In this case, edge node 2 will try to resend its packet.

2. Edge node 2 has failed and cannot not send its heartbeat packet.

We cannot distinguish between both of the cases unless we wait for a certain amount of time to check whether edge node 2 resends its heartbeat packet. If edge node 2 does not send its heartbeat packet, within say, $HEARTBEAT\_PERIOD/2$ time, then we perceive edge node 2 as failed.

Therefore, the heartbeat packets are received at time $t = \{T, 2T, 3T, ...\}$ and the failure of the edge nodes is checked at time $t = \{3T/2, 5T/2, 7T/2, ...\}$.

If failure is detected for a set of edge nodes, the Collector node reports the IDs of the failed edge nodes to the central server for replica recovery.

```
while True:
    sleep(HEARTBEAT_PERIOD)
    failed_nodes = {}
    for j from 0 to NUM_EDGE_NODES - 1:
        if NODE_FAILURE[j] == 0:
            failed_nodes.add(j)
    send failed_nodes to central server
    for i from 0 to NUM_EDGE_NODES - 1:
        NODE_FAILURE[i] = 0
```

The central server receives the failed edge node IDs, and selects the data blocks from the failed node to be recovered. In the original algorithm, the data blocks with the least recovery costs were chosen to be recovered. We introduce another factor of access probability which accounts for the demand of the data block. The data blocks with high demand are chosen to be recovered which improves the read response time.

Finally, the central server sends the recoverable data blocks to the target edge node. We reuse the OBJECTIVE function given in Subsection 5.2 instead of the given function. This is because the OBJECTIVE function already captures all the load capacity parameters of an edge node, and is more versatile.

```
potential_data_blocks = {}
for j from 0 to NUM_EDGE_NODES - 1:
    if NODE_FAILURE[j] == 1:
        for d from 0 to NUM_DATA_BLOCKS - 1:
            if BIN_ENCODING[j][d] == 1:
                potential_data_blocks.add(d)

if len(potential_data_blocks) == 0:
    reset NODE_FAILURE
    return

sco = 0
for d from 0 to NUM_DATA_BLOCKS - 1:
    sco += H[d]

cost = [0 for d from 0 to NUM_DATA_BLOCKS - 1]
for d from 0 to NUM_DATA_BLOCKS - 1:
    cost[d] = (size[d] + Tseek[d]) / BW[d]
```

NODE_FAILURE[j]	Indicates failure of j^{th} edge node
sco	Total count of accesses for all data blocks
cost[d]	Block recovery cost for data block d
size[d]	Size of data block d
Tseek[d]	Time required to locate other replicas of the data block d
BW[d]	Represents maximum available network bandwidth
sel[d]	Replica selection factor for the data block d
asel	Average value of replica selection factor for all data blocks in the failed edge node

Table 4: Replica recovery strategy

```
asel = 0
sel = [0 for d from 0 to NUM_DATA_BLOCKS - 1]
for id in potential_data_blocks:
    sel[id] = H[id] / sco
    sel[id] /= cost[id]
    asel += sel[id]

asel /= len(potential_data_blocks)

recoverable_data_blocks = {}
for id in potential_data_blocks:
    if sel[id] > asel:
        recoverable_data_blocks.add(id)

for j from NUM_EDGE_NODES - 1 to 0:
    edge_id = OBJECTIVE[j]
    new_data_blocks = {}
    for d in recoverable_data_blocks:
        if BIN_ENCODING[edge_id][d] == 0:
            new_data_blocks[d] = DATA[d]
    for d in new_data_blocks.keys():
        recoverable_data_blocks.remove(d)
    send the new_data_blocks to node
      edge_id

reset NODE_FAILURE
```

The overall time complexity of the above algorithm is $O(NUM\_EDGE\_NODES * NUM\_DATA\_BLOCKS)$ and space complexity is $O(NUM\_EDGE\_NODES + NUM\_DATA\_BLOCKS)$.

6 Observations and Results

The system response time (SRT) performance of the original and improved dynamic replica creation algorithms under the changes in number of data block accesses is summarized in Figure 8. For both the algorithms, SRT increases with increase in number of data block accesses. This is because more data block accesses leads to higher system stress, which results in

more queuing of jobs, thereby increasing the overall SRT for all jobs. Also, at lower frequencies, both algorithms perform similarly. But as frequency increases, the improved algorithm provides lesser response times thereby performing slightly better than the existing algorithm . This is because at higher frequencies, the impact of considering more history in calculation of heat is profound, thereby providing better results.

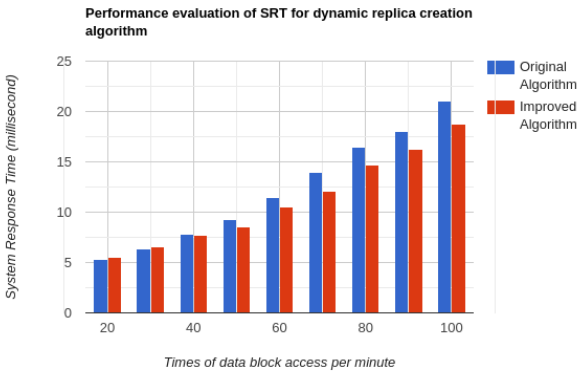

Figure 8: SRT for dynamic replica creation algorithm

The storage space utilization (SSU) performance of the original and improved replica placement algorithms under the changes in number of submitted jobs is summarized in Figure 9. For both the algorithms, SSU increases with increase in number of submitted jobs. The improved algorithm performs better by having lower SSU than the existing algorithm. This is because as demand is also taken into consideration in the improved placement algorithm, hence fewer replicas are needed to suffice the need, resulting in availability of more free space.

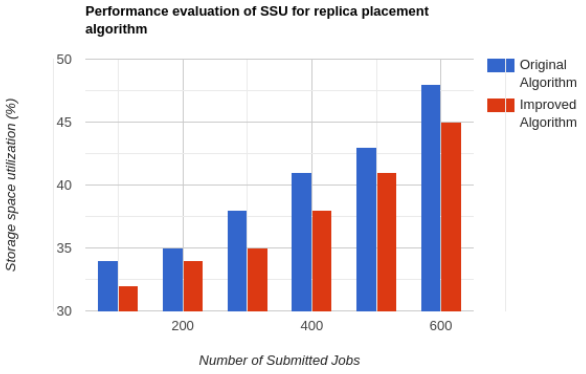

Figure 9: SSU for replica placement algorithm

The average read throughput (ART) performance of the original and improved replica placement algorithms under the changes in number of data block accesses is summarized in Figure 10. For both the algorithms, ART increases with increase in number of data block accesses as the probability of reading data blocks gets increased. The improved algorithm performs better by having higher ART than the existing algorithm. This is because as demand is also taken into consideration in the improved placement algorithm, hence replicas are placed at more appropriate positions resulting in greater number of reads, thereby increasing the ART.

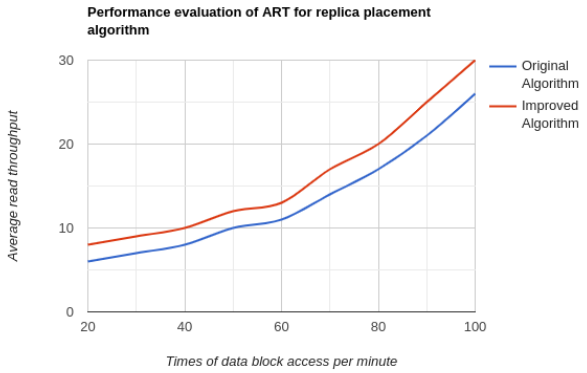

Figure 10: ART for replica placement algorithm

The time for data writing (TDW) performance of the original and improved replica synchronization algorithms under the changes in data block size is summarized in Figure 11. For both the algorithms, TDW increases with increase in data block size. This is because for a fixed hardware configuration, write to a larger data block takes more time. The improved algorithm performs better by having lower TDW than the existing algorithm. This is because in the improved algorithm while choosing W edge nodes for synchronization, W/6 edge nodes closest to the central server are chosen which ensures lower network latency, thereby reducing the time for data writing.

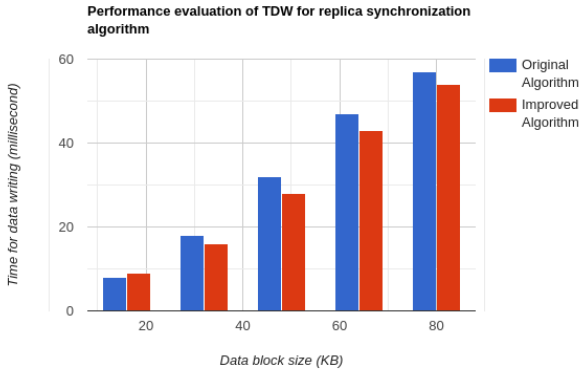

Figure 11: TDW for replica synchronization algorithm

The average read throughput (ART) performance of the original and improved replica synchronization algorithms under the changes in number of concurrent read is summarized in Figure 12. For both the algorithms, ART increases with increase in number of concurrent read as more data is read in that case. The

improved algorithm performs better by having higher ART than the existing algorithm. This is because in the improved algorithm while choosing W edge nodes for synchronization, W/3 edge nodes are chosen having the highest demand resulting in success of more read requests from those edge nodes, thereby increasing the read throughput.

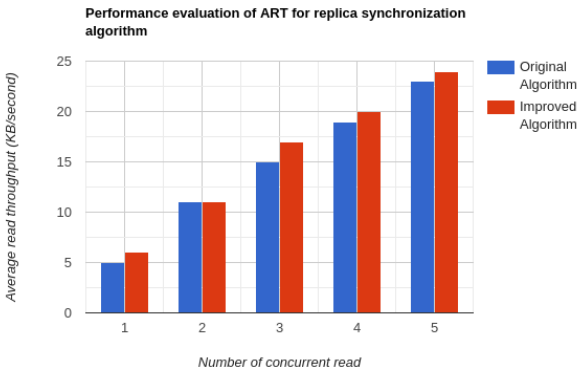

Figure 12: ART for replica synchronization algorithm

The read response time (RRT) performance of the original and improved replica reconstruction algorithms when a data node failure occurs is summarized in Figure 13. For both the algorithms, RRT suddenly increases when the data node failure occurs. The improved reconstruction algorithm performs better by reducing the data read response time faster than the existing algorithm. This is because, after a data node fails, in the selection of data blocks to be recovered, access probability is also taken into consideration in the improved algorithm resulting in recovering of data blocks having more read requests, thereby reducing the read response time.

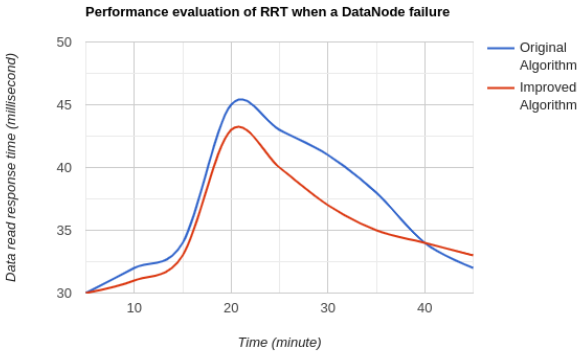

Figure 13: RRT when a DataNode failure

The average read throughput (ART) performance of the original and improved replica reconstruction algorithms when a data node failure occurs is summarized in Figure 14. For both the algorithms, ART suddenly decreases when the data node failure occurs. The improved reconstruction algorithm performs better by increasing ART faster than the existing algorithm. This is because, after a data node fails, in the selection of data blocks to be recovered, access probability is also taken into consideration in the improved algorithm resulting in recovering of data blocks having more read requests, thereby increasing the read throughput.

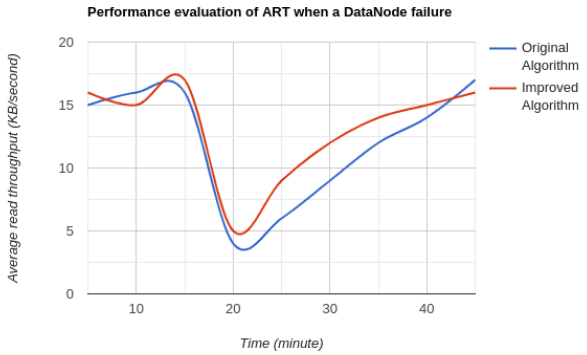

Figure 14: ART when a DataNode failure

The system response time (SRT) performance of the original and improved replica reconstruction algorithms under the changes in number of submitted jobs is summarized in Figure 15. For both the algorithms, SRT increases with increase in number of submitted jobs. The improved algorithm performs slightly better by having lower SRT than the existing algorithm. This is because, after a data node fails, in the selection of target node, demand factor of edge node is also taken into consideration in the improved algorithm. This results in placing data blocks at edge node having more requests, thereby reducing the system response time.

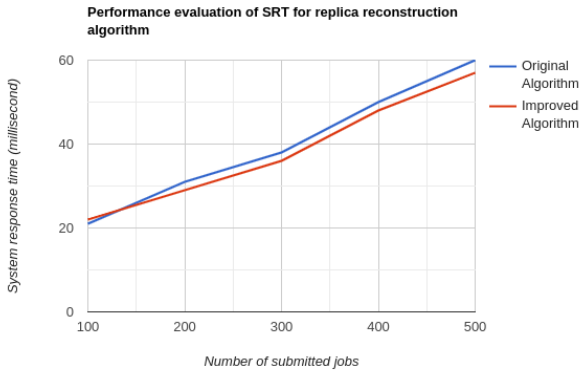

Figure 15: SRT for replica reconstruction algorithm

7 Conclusion and Future Work

We analyse how effective replica management improves the performance of an edge cloud environment in a car navigation system. Our major contribution

is combining the four existing algorithms in a unified car navigation system framework and improving upon them to increase the edge cloud performance. We simulate the improved algorithms in a Python and Docker environment using socket programming for inter-container communication. The simulation verifies the results of the improved algorithms which are better than the former ones.

We first study the problem of dynamic replica creation in the edge cloud system and this paper proposes an improved algorithm which is able to effectively reduce system response time. Then, we study the problem of replica placement, and the improved algorithm is able to reduce system response time, improve data read throughput and system storage space utilization while ensuring load balancing. At last, we study the problem of data replica synchronization and the data recovery of failed edge nodes, and the improved algorithms produce better data read response times, system response time, time for data writing and average read throughput.

The future work includes simulating the improved algorithms in a larger and practical edge cloud environment setting, and examining the results of the simulation to check whether the improved algorithms can be adopted in an existing car navigation system. We can also deliberate upon how the network architecture of the edge cloud environment is placed in an exemplary locality, and examine how the architecture can affect performance while minimizing the hardware costs.

References

[1] B. Qi, L. Kang, S. Banerjee, A vehicle-based edge computing platform for transit and human mobility analytics, in: ACM/IEEE Symposium, ACM, 2017, pp. 1–14.

[2] Nasir Abbas, Yan Zhang, et al., Mobile edge computing: A survey, IEEE Internet Things J. 5 (1) (2018) 450–465.

[3] Rodrigo Romana, Javier Lopez, et al., Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges, Future Gener. Comp. Syst. 78 (2) (2018) 680–698.

[4] W.C. Chang, P.C. Wang, An adaptable replication scheme in mobile online system for mobile-edge cloud computing, in: IEEE, 2017, pp. 109–114.

[5] R. Kingsy Grace, R. Manimegalai, et al., Dynamic replica placement and selection strategies in data grids — A comprehensive survey

[6] Charles J. Geyer, Practical Markov chain Monte Carlo, Statist. Sci. 7 (4) (1992) 473–483.

[7] R. Chandakanna, Veerabhadra, REHDFS: A random read/write enhanced HDFS, J. Netw. Comput. Appl. 103 (2018) 85–100.

[8] Bashari Rad, Babak Bhatti, Harrison Ahmadi, Mohammad. (2017). An Introduction to Docker and Analysis of its Performance. IJCSNS International Journal of Computer Science and Network Security. 173. 8.

[9] Chunlin Li, Mingyang Songa, Min Zhang, Youlong Luo: Effective replica management for improving reliability and availability in edge-cloud computing environment, Journal of Parallel and Distributed Computing, Volume 143, September 2020, Pages 107-128

[10] Javaid, Adeel. (2013). Understanding Dijkstra Algorithm. SSRN Electronic Journal. 10.2139/ssrn.2340905.