IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 20 July 2023, accepted 18 September 2023, date of publication 25 September 2023, date of current version 4 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3319217

RESEARCH ARTICLE

RemOrphan: Object Storage Sustainability Through *Remo*ving Offline-Processed *Orphan* Garbage Data

JANNATUN NOOR[1,2], NAJLA ABDULRAHMAN AL-NABHAN[3], AND A. B. M. ALIM AL ISLAM[1]

[1]Next-Generation Computing (NeC) Research Group, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka 1000, Bangladesh
[2]Computing for Sustainability and Social Good (C2SG) Research Group, School of Data and Sciences, BRAC University, Dhaka 1212, Bangladesh
[3]Department of Computer Science, King Saud University, Riyadh 11362, Saudi Arabia

Corresponding author: Jannatun Noor (jannatun.noor@bracu.ac.bd)

The authors extend their appreciation to the Deputyship for Research & Innovation, Ministry of Education in Saudi Arabia for funding this research work through the project number (DRI-KSU-762).

ABSTRACT Nowadays, extremely large amounts of structured and unstructured types of data are stored in public, private, and hybrid cloud storage using object storage systems. Among these, storing multimedia data such as image, video, and audio pose unique challenges and long-term effects on object storage Sustainability. Three such challenges are smoother and more efficient video streaming, middleware placement for media processing, and lastly, management of orphan garbage data. In order to tackle these challenges, this paper presents a generalized architecture for smooth and efficient management as well as retrieval of multimedia data in cloud systems. To do so, first, we propose a new middleware package in the object server for supporting smooth video streaming and on-demand playable video segments. Here, we demonstrate that video segment download time improves by up to 30% when segmentation is done in the object server rather than in the proxy server. After, we focus on how to find orphan garbage data on media cloud storage and to what extent they can hamper data retrieval. Specifically, we present a generalized architecture named *'RemOrphan'* for detecting the orphan garbage data using OpenStack Swift hash Ring and scripts. We deploy a private media cloud SPMS and find that around 35% data can be orphan garbage data. Due to the huge amount of orphan data, rsync replication needs higher time and more network overhead which hampers the system's sustainability. We lower around 25% sync delay and 30% network overhead after deploying a deletion daemon to remove the orphan garbage data.

INDEX TERMS Object storage system (OSS), offline video processing, middleware, garbage collector, video segmenter, orphan garbage data.

I. INTRODUCTION

With myriad diversified applications, multimedia communication over the cloud has experienced a substantial surge in interest in recent times [1], [2], [3]. As of 2022, the Global Next-Generation Data Storage Market was valued at USD 58.35 Billion. The market is projected to experience a compound annual growth rate (CAGR) of 7.8% from 2023 to 2032. It is anticipated that the Worldwide Next-Generation Data Storage Market will reach USD 123.66 Billion by 2032 [4], [5]. Several unstructured data storage needs concurrent communication with cloud systems. Such communication entails general user services as well as special user services such as services to management personnel. The management personnel can be law-enforcing agency people, crowd-monitoring authority persons, etc. A classical example in this regard is the authority of Hajj crowd monitoring authority [6]. A use case for the authority for our focused context is shown in Figure 1. To serve such use cases, promising multimedia-based cloud systems are now emerging [7],

The associate editor coordinating the review of this manuscript and approving it for publication was Yufeng Wang.

[8], [9]. These systems often use various open-source Object Storage Systems (OSS) for faster and easier access to image and video-type data, which are the two foremost ingredients in multimedia communication over the Internet.

A. OBJECT STORAGE SUSTAINABILITY

Here comes the necessity of Object Storage Sustainability in the long run with respect to the continuous growth of unstructured data. Object storage sustainability refers to the design and implementation of sustainable practices in the context of object storage systems. Object storage is a method of storing and managing data as objects, which are typically stored with associated metadata and a unique identifier. Sustainability in object storage can encompass various aspects such as energy efficiency, resource, and storage optimization, and environmental impact reduction. Besides, Object storage Systems data management and communication is highly dependent on middleware design and placement of the middleware in proxy or storage servers [10]. Multiple copies of big data are stored in OSS to ensure data availability. Hence, regular syncing and checking are necessary for finding bit rot and file degradation to ensure long-term preservation storage. Several studies focus on data storage sustainability and its impact to ensure long-term sustainability to avoid undesirable consequences [11], [12].

In this paper, we primarily address two key concerns regarding the sustainability of object storage: 1) the impact of middleware placement within the object storage system, and 2) the deletion of orphan garbage data. By addressing these concerns, we aim to enhance the sustainability and effectiveness of object storage systems, promoting better resource utilization, performance optimization, and data management practices.

B. SUSTAINABILITY CONCERN: MIDDLEWARE PLACEMENT

Besides, several applications such as crowd management, real-time location-aware services, and medical systems need to access multimedia data from diversified remote devices (in Figure 1). As an example, crowd management of millions of pilgrims for performing Hajj, Umrah, and Kumbh Mela is challenging, and appropriate processing and communication from the cloud is a must [6], [13]. Hence, context-aware and location-aware cloud-based frameworks and services are emerging [14]. These frameworks need both online and offline processing of unstructured data such as images and videos. Additionally, real-time video streaming is another prominent feature for managing these kinds of services using cloud infrastructures [15].

Similarly, video experiences slower responses from important sites, as the sizes of video files are generally much higher than those of corresponding image files. Many video streaming service providers in this regard provide their services using cloud-based video storage systems. Here, efficient cloud-side operation management is needed to ensure different features such as smooth video streaming, dynamic adaptive streaming, etc. Besides, proper and updated video segments need to be supplied from the cloud storage systems to achieve the features. In this regard, Recent studies focus on several methods of mobile and web streaming [1], [2], [16], [17], gateway-based shaping methods for HTTP adaptive streaming (HAS) [18], quality of experience of HAS [19], optimal transcoding and caching for adaptive streaming in content delivery networks [20], etc.

However, none of these studies focuses on video streaming support for a cloud specifically for an Object Storage System, which is now treated as a well-adopted solution for cloud service development. Moreover, video streaming features of this kind are primarily designed and implemented utilizing multiple middleware components. Hence, our first concern centers around the placement of middleware within the object storage architecture, which is yet to be addressed in the literature. We examine the effects and implications of different placement strategies for middleware components, aiming to optimize their performance and efficiency within the overall storage system.

C. SUSTAINABILITY CONCERN: ORPHAN GARBAGE DATA

Yet another aspect worth investigating for image and video-delivering clouds is the efficient usage of cloud storage. Due to diversification in operations and usages, there can arise different types of data and objects in such storage. For example, components such as client database, AUTH server database, etc., can produce data and objects that will be never required at all. To be more specific, in the multimedia cloud storage [8], [21], there can be different versions of data for images and videos that are never going to be accessed by a user.

Irrespective of the future requirements, all the data or objects of storage are generally considered to be an asset for cloud storage, as data storage has no concern about what data is stored or whether the data is necessary or not. However, there are some other components such as client database, AUTH server database, etc., which are always necessary components for designing a complete system. Therefore, in reality, all data in the cloud storage may not always be an asset for other components. For example, a user can upload some personal images to a media cloud system. All the versions of the images were uploaded successfully, however, when returning the response the network got disconnected (in Figure 2).

Hence, there will be no information about these images in the AUTH server where all the lists of files are stored for users. This data can be useful for cloud storage, however, never be used for users' purposes. We use a new term for this kind of data - *orphan garbage data* - to imply such data is garbage as well as having no effective linkage to its ancestor. Such data gets generated by different types of cloud operations, e.g., when a cloud operation produces different versions of the same data. This happens in the case of offline

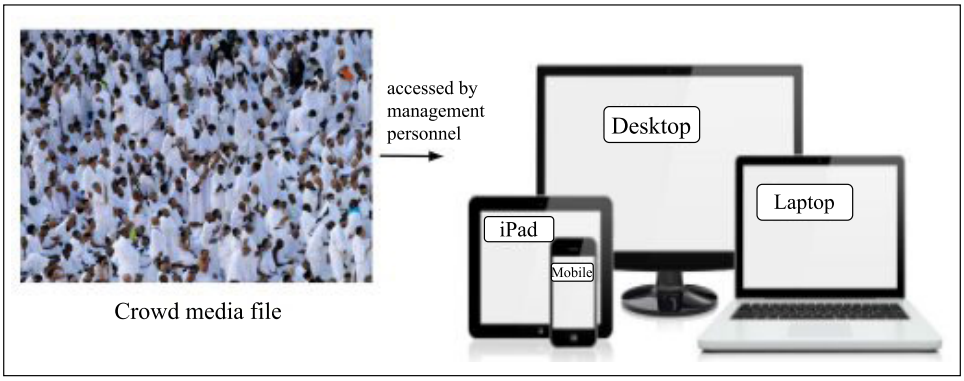


FIGURE 1. A use case of offline processing media data storage. Here, several crowd media files are accessed from cloud storage by the Hajj management personnel using several diversified remote devices whenever needed. Hence, different versions of media files (images and videos) are stored in the cloud storage using offline processing beforehand.

processing media clouds that produce different versions of data both for images and videos. Such productions result in orphan garbage data in multimedia cloud storage.

Furthermore, research studies focus on several aspects of such redundant data deletion architecture. Some studies present the memory garbage collector algorithms in big data context [22], [23]. Other studies, Linux container-based deletion [24], Smartbin-based deletion in wireless sensor networks [25], orphan process detection [26], [27], and assured deletion [28], [29] present some deletion approaches, which are not applicable to the case of orphan garbage data in cloud due to architectural as well as operational mismatches between cloud and the cases focused on these studies. Hence, our second concern relates to the deletion of orphan garbage data. Orphan garbage data refers to data that is no longer associated with any active objects or containers within the storage system. We explore techniques and approaches for identifying and removing such orphan data, contributing to improved storage efficiency and management.

D. MOTIVATION AND IMPLICATION

In summary, the substantial growth of unstructured data generated and stored in Object Storage Systems globally has resulted in a notable increase in energy consumption. Besides, middleware modules offer compression or optimization features that reduce the size of responses sent to clients. Smaller response sizes can result in reduced network bandwidth requirements, leading to potential energy savings. By reducing the data transfer overhead, less energy is consumed during transmission and storage. Research studies have yet to address the crucial aspects of multimedia cloud operation and communication, including the impact of middleware placement, the design of adaptive segmented video streaming [30], [31], [32], and the management of orphan garbage data for ensuring the sustainability of Object Storage.

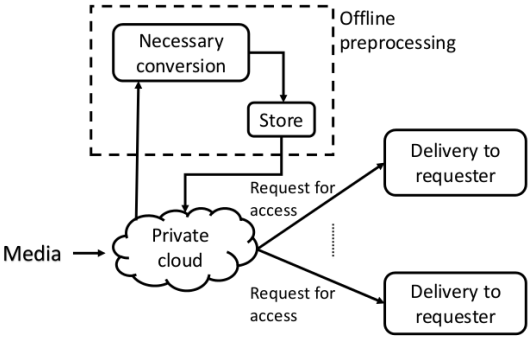
Therefore, in this study, we first propose a new middleware named 'VideoSegmenter', which is used for making video segments according to any kind of time range using FFmpeg [33]. One specialty of our middleware architecture is that it can give the user/streaming server any playable segment on the fly. Another specialty is the ability to deploy this middleware in the object server rather than in the proxy server in OpenStack Swift.

Besides, we propose a novel approach *'RemOrphan'* for detecting and deleting orphan garbage data in a multimedia cloud. Here, we develop a deletion daemon to find and remove orphan data in an efficient manner to make the data storage sustainable along with enhancing CPU usage. In proposing all these new techniques, we present a video segmenter middleware, the impact of middleware placement, and a deletion daemon to eventually perform the tasks from the same OpenStack-like system. We evaluate the performance of the system in an actual setup that involves a server located in Canada and a client situated in Bangladesh. Our rigorous experimental results demonstrate that we can achieve up to 30% lower video segment download time, 30% reduced network overhead, and 25% reduced sync delay through utilizing our proposed techniques.

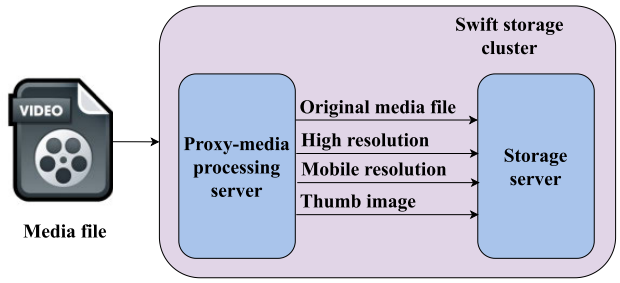
E. RESEARCH CONTRIBUTIONS AND ORGANIZATION

Our study yields the following contributions, as outlined in this paper:

- We analyze two important factors related to long-term Object Storage sustainability - the impact of middleware placement and orphan garbage data in the Object Storage System.
- We design a new middleware 'VideoSegmenter' for supporting HTTP adaptive streaming in OpenStack Swift-like systems such as SPMS. Accordingly, we implement a new package using setup-tools [34], which can be easily integrated into the existing OpenStack Swift. We analyze and present that the proposed middleware should be deployed in the object server to get faster responses to avoid extra overhead and maintain long-term sustainability.



(a) Offline processing in media cloud [8]



(b) Media file is converted to different resolutions to support diversified devices

FIGURE 2. Offline processing models for storing and retrieving media files from the cloud. Here, multiple versions of media files are processed and stored in the cloud. During the processing time, orphan data may be stored in the cloud.

- We present a new technique for finding unused orphan garbage data in multimedia storage systems, which poses a great threat to sustainable storage systems. This orphan data is responsible for unnecessary sync delay in replication and extra network overhead related to replica (r) and the number of objects per node (n).
- Moreover, we design a deletion daemon named *'RemOrphan'* for removing the orphan data using OpenStack rings and scripts in an efficient way. Our custom deletion daemon presents a configurable solution that offers options to run it once or in a periodic manner.
- Finally, we perform a rigorous experimental evaluation of our proposed methodologies in an actual testbed, which consists of a high-configuration server located in Canada and a client situated in Bangladesh. Our experimental results confirm the efficacies of all our proposed techniques against that of classical alternatives.

The organization of this paper is further segmented into different sections. Section II contains the literature reviews of recent papers related to our study. After, in Section III, we present three important concepts - an OpenStack Swift-like object storage system, middleware in OSS, and the definition of orphan garbage data. Next, system design and implementation are presented in Section IV. Besides, Section V contains experimental test-bed setup and performance evaluation. Furthermore, in Section VI, we present the discussion and comparative analysis of our proposed methodology with the existing literature. Finally, the conclusion and future prospects of this research are stated in Section VII respectively.

II. RELATED WORK

In this Section, we present the related studies exploring object storage sustainability, on-demand video segmentation, and streaming and orphan garbage data deletion. In the literature, we find very few studies on object storage sustainability. Study [11] demonstrates that long-term preservation storage requires more than just storing multiple copies of a file. It is also essential to regularly check those copies for bit rot and other types of degradation. Therefore, file integrity tools are necessary to ensure the ongoing integrity of the stored data. Another study [12] presents that the utilization of big data at a massive scale is likely to result in some negative repercussions. While some of these consequences can be predicted, others may be completely unforeseeable. This essay focuses on the sustainability-related issues that arise from the implementation of big data.

A. ON-DEMAND VIDEO SEGMENTATION AND MIDDLEWARE PLACEMENT

Efficient and smoother video streaming, dynamic adaptive streaming, etc., are some special features, which need both cloud server-side and streaming server-side operation management. For achieving these features, studies [7], [8], [35] mainly focus on the faster and more secure management of media files through designing middlewares. The increasing demand for video streaming services over mobile networks has outpaced the capacity of wireless links to handle the growing traffic load. As a consequence, the service quality of video streaming is adversely affected, leading to a subpar user experience. A research study in this regard [17] constructs a private agent for each active mobile user in the cloud to adaptively adjust the video quality utilizing scalable feedback-based video coding techniques.

Furthermore, study [2] has developed a new framework called EMS for streaming ultra-high-definition video. This framework combines erasure-coded storage with multi-source streaming. Moreover, they created two metrics, one for deadline awareness and the other for latency sensitivity, to measure the quality of service provided by video servers. Additionally, they propose a federated learning approach to adaptively update the service quality, which leverages the power of reinforcement learning techniques to dynamically select the most suitable servers for local user training, and a global aggregation of service quality. Study [1] introduced SPACE where, they have developed and analyzed several segment prefetching policies that vary in terms of resource usage, required radio and player metrics, and deployment complexity.

Moreover, study [36] presents the WVSNP-DASH framework, which relies on video segments that can be played independently and have a particular naming syntax that

conveys elementary metadata. This system facilitates flexible search, transfer, distribution, and playback of video segments. To enhance the adaptive video streaming performance in CCN, study [37] suggests a hop-by-hop adaptive video streaming approach known as HAVS-CCN. Other studies focus on several methods of mobile and web streaming [16], gateway-based shaping methods for HTTP adaptive streaming (HAS) [18], survey on quality of experience of HAS [19], etc.

In addition, existing CDNs may not be sufficiently cost-effective for distributing adaptive video streaming due to the lack of orchestration on storage, computing, and bandwidth resources. Hence, a research study [20] leverages the notions of media cloud to deliver on-demand adaptive video streaming services, where those resources can be dynamically scheduled minimizing the total operational cost by optimally orchestrating multiple resources. For this, the study formulates and utilizes an optimization problem by examining a three-way trade-off between the caching, transcoding, and bandwidth costs. However, there is no study on video streaming support for object storage systems and the impact of middleware placement on storage sustainability.

B. ORPHAN GARBAGE DATA DELETION

Furthermore, the research study [28] presents a notion of SmartBin in place of old-fashioned practices such as hiring people to regularly check and empty filled dustbins. SmartBin integrates the idea of IoT with Wireless Sensor Networks. Another study [29] discusses the need for assured deletion in the cloud along with identifying cloud features that pose a threat to assured deletion and describes various assured deletion challenges as well. Besides, study [38] focuses on analyzing the GREEDY Garbage Collector strategy under the condition of uniformly independently distributed write accesses.

Moreover, study [23] examines existing Big Data platforms and their memory profiles to investigate why traditional algorithms, which remain widely used, are inadequate. It also evaluates newly suggested memory management algorithms that are specifically designed for Big Data environments [39], [40], [41], [42], [43]. The research assesses the scalability of these recent memory management algorithms by comparing their throughput (improvement in application throughput) and pause time (reduction in application latency) to that of classic algorithms. Besides, study [22] performs a thorough evaluation of three widely used garbage collectors, namely Parallel, CMS, and G1, using four typical Spark applications. Their evaluation involves a comprehensive analysis of the relationship between the memory usage patterns of these big data applications and the GC patterns of the collectors, leading to several insights into GC inefficiencies. Based on the findings, the authors provide empirical guidelines for application developers and offer useful optimization strategies for developing garbage collectors that are suitable for big data environments.

On the other hand, in distributed systems, orphan processes may be generated as a result of remote procedure calls (RPC). There are two types of orphans: crash-orphans, which occur when the client crashes, and abort-orphans, which occur when the parent process is aborted. Orphan processes are problematic because they consume system resources and can result in inconsistent data. To address this issue, several studies have developed new methods for detecting orphan processes [26], [27]. However, none of these studies deal with the problem of removing orphan garbage data.

To the best of our knowledge, our proposed methodology is the first to focus on video streaming data retrieval and the impact of middleware placement in object storage systems. Besides, we present an architecture named *'RemOrphan'* for orphan garbage data detection and deletion to maintain healthy and sustainable storage, which is yet to be focused on in the literature.

III. BACKGROUND

Our goal is to focus on object storage sustainability by analyzing the impact of middleware placement and orphan garbage data deletion. Hence, in this section, first, we present a media cloud storage system named SPMS which is designed using OpenStack Swift. After, we describe the details of middleware in OSS. Finally, the definition of orphan garbage data is presented using appropriate examples.

A. SPMS (SECURE PROCESSING AWARE MEDIA STORAGE)

OpenStack Swift has gained popularity in the deployment of numerous media cloud storage. It is an open-source object storage system that offers various notable features, including eventual consistency, fault tolerance, high availability, and replication. Swift comprises two types of servers: the proxy servers responsible for management and processing, and the storage servers (account, container, and object servers) designed for storing databases and data objects [10].

1) READ AND WRITE REQUEST

When a client sends a read request for an object o, the proxy node initially searches for the storage nodes that store the replicas of o. Subsequently, it sends requests to all the replica (r) nodes of object o. OpenStack Swift typically employs a quorum-based voting mechanism for controlling replicas. Once the proxy node receives a sufficient number of valid responses ($\geq \lfloor r/2 \rfloor + 1$), it chooses the best response, which is the one with the most recent version of o, and then forwards it to the client. Conversely, when a write request is made for o, the proxy node dispatches the request to all r storage nodes that host o. The write operation is considered successful as long as a specific number ($\geq \lfloor r/2 \rfloor + 1$) of storage nodes respond with a ''successful write'' message.

2) PARTITION AND SYNCHRONIZATION

OpenStack Swift, like other popular storage systems, employs consistent hashing (also known as a distributed hash table or DHT) to organize data partitions. Specifically, Swift

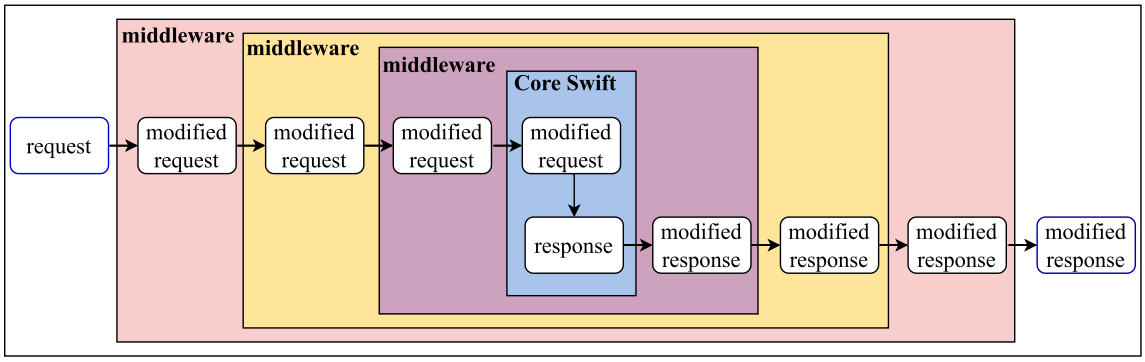


FIGURE 3. How different layers of middleware work in Web Server Gateway Interface (WSGI) for Object Storage System.

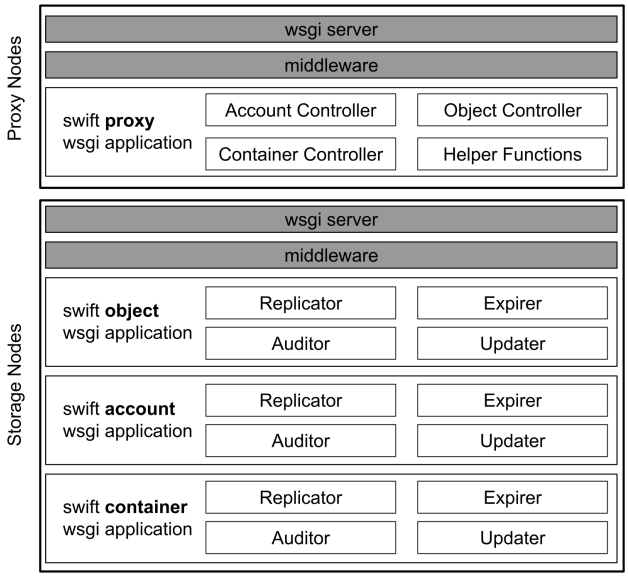


FIGURE 4. Different consistency processes and layers in proxy and storage nodes of OpenStack Swift.

constructs a logical ring, referred to as the object ring or partition ring, which represents the entire storage space. This logical ring consists of multiple equivalent subspaces, with each subspace representing a partition. Each partition encompasses a certain number of h^2 objects associated with that partition.

The value of h in consistent hashing dynamically adjusts with the scale of the system. Additionally, each partition is replicated r times on the logical ring and physically mapped to r different storage nodes. Assuming homogeneity among all N storage nodes in the logical ring, each node hosts a proportional number of partitions, which is calculated as $\frac{r \times p}{N}$, where p represents the total count of unique partitions. To uniquely identify each object, a unique identifier is assigned, typically generated by computing the MD5 hash value of the object's path.

SPMS, which is designed using OpenStack Swift through adding several new middlewares. SPMS system has every feature of Swift. Moreover, SPMS provides additional features related to media management, including media securing, conversion of image data to PJPEG format, resizing of images to multiple dimensions, and transcoding and resizing of videos to various sizes [8], [44]. Since SPMS-like media storage systems are commonly employed for diverse media management tasks such as video streaming and storing multiple versions of objects, it becomes essential to address the challenges of optimizing multimedia retrieval and effectively managing orphan garbage data to ensure long-term sustainability.

B. MIDDLEWARE IN OBJECT STORAGE SYSTEM

An Object Storage System like OpenStack Swift is constructed on Python's Web Services Gateway Interface (WSGI) model and configured through the Python Paste framework. In the WSGI model, middlewares are a vital part and they are designed to pass the requests through several layers to reach the core application. Besides, middleware wraps other middleware one by one down to the core application in the center without knowing anything about the other layers. Hence, developing middleware codes are easy and straightforward for the developer to design new features.

Figure 3 presents how a middleware system with multiple layers works. When a user request enters the system, the request is potentially altered by each middleware layer as it moves inward toward the final processing by the core application. The response then travels back out the layers of middleware, with each layer having the option to modify the response. Each middleware layer has the capability to modify an incoming request or allow it to pass through without any changes. Eventually, the final response is sent back to the user. Each middleware layer can inspect, modify, or short-circuit a request or response. Different features are implemented using middleware. In OpenStack Swift, processing-related tasks are handled in the proxy server. There are other consistency services to maintain replication, audit, and update of objects in both proxy and storage servers. Figure 4 presents different consistency processes and layers in proxy and storage nodes (servers) of OpenStack Swift.

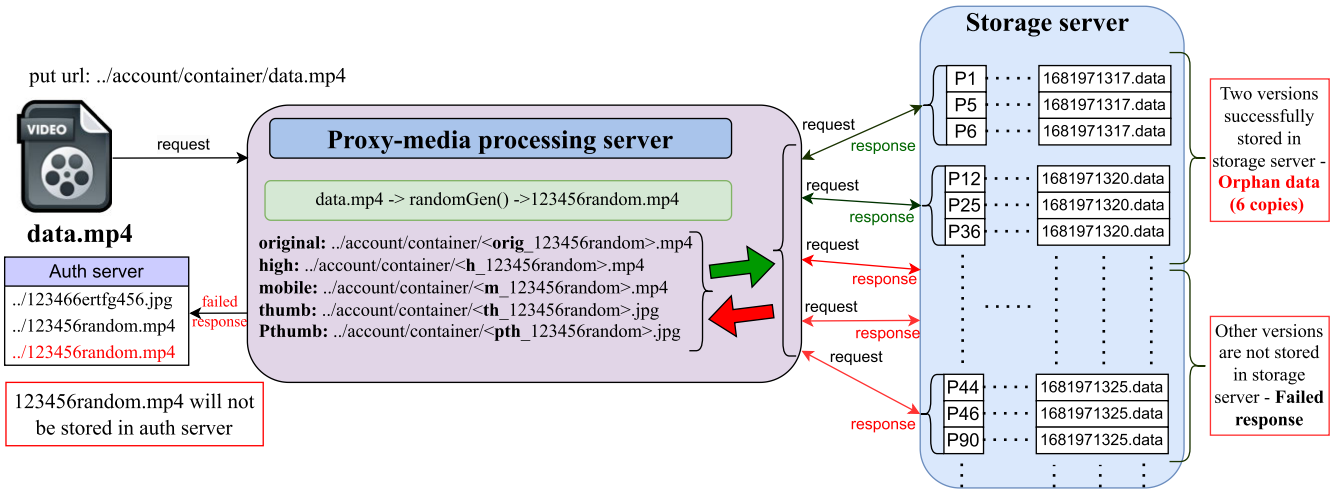


FIGURE 5. How orphan garbage data is created due to network disconnection, client timeout problem, object versioning, etc. Here, the data.mp4 file is uploaded from the client. For this single video file, five versions are uploaded in the storage nodes having three copies for each version. Two versions are successfully stored when background processing, but somehow other versions are not uploaded successfully due to several reasons. Hence, the final response failed and the URL is not stored in the AUTH database. The above six copies are orphan garbage data that are still in the storage server but will never be used.

Here, each node consists of middleware layers to perform several tasks. Hence, middleware placement is a concerning issue for the long-term sustainability and efficiency of an Object Storage System.

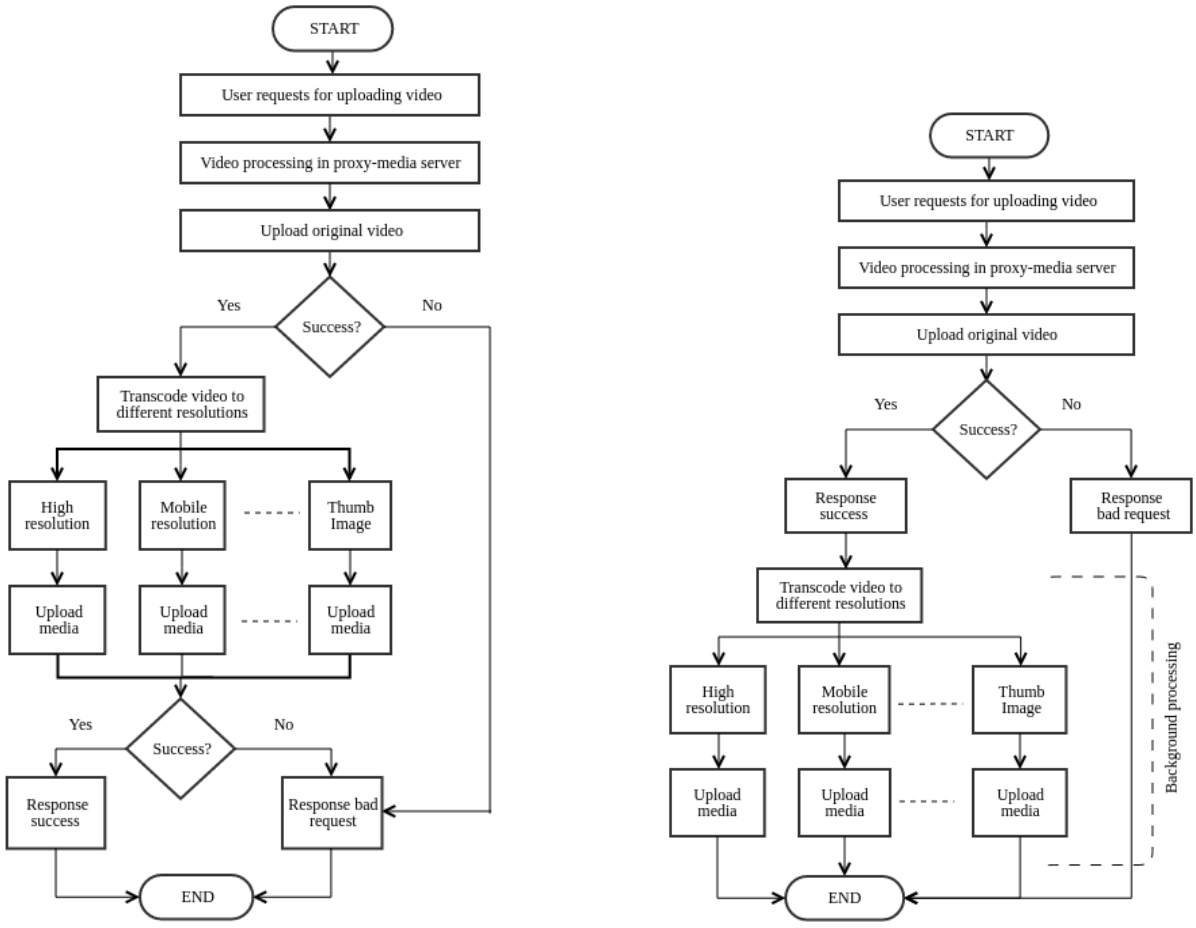
C. ORPHAN GARBAGE DATA

Data management with optimization of CPU and memory usage in data centers is now the most challenging topic for data scientists. Any kind of data is valuable, as all the data in a storage system can be used for further processing or mining purposes. A big question is now, is there any garbage data in the cloud? For example, resizing images and transcoding videos into multiple resolutions to cater to various remote devices and data versioning must be needed in the media cloud. To do so, recent studies present offline processing models for storing and retrieving media files from the cloud (in Figure 2a). Therefore, for a single object, there are several different resolutions of the original one (in Figure 2b). The data which is never used for quite a long time, is referred to as 'unused data'. This data can be archived using an erasure coding policy or different kinds of mechanisms. However, there is some kind of data that exists in cloud storage, which has no information both on the client side or in the AUTH database. This can happen for network disconnection, client timeout problems, etc. This is the orphan garbage data that can greatly threaten data storage sustainability.

Moreover, several studies introduce and design middleware for image and video processing tasks in the Object Storage System. In SPMS-like systems, for covering diversified remote devices, many versions of images (200, 300, 600, etc., to aspect ratio) along with progressive JPEG images are stored. The same applies to video files, e.g., high-resolution (720 to aspect ratio) video files, mobile-resolution (400 to aspect ratio) video files, etc. Besides, study [21] presents that the different versions of photos viewed on Facebook are around 80-100 Billion. Among them, the Thumbnails version is viewed around 10.2%, the Small version is 84.4%, 0.2% is the Medium version, and the Large version is 5.2%. This is one of the main reasons for getting orphan garbage data created, where no information either on the client side or in the AUTH database about the orphan data gets stored. The non-existence of information can occur due to network disconnection, client timeout problem, object versioning, etc., [8], [35]. Furthermore, transcoding large video files into different resolutions is a challenging task, hence, some research studies propose background processing rather than waiting for all the versions to upload and send a successful response.

In study [8], the researchers propose two different designs to convert and upload high resolution, mobile resolution, and other versions along with the original video file. In the first case (Case-1: 'Response after all uploading'), the system sends a response after successfully uploading all the versions, hence it takes a much longer time and the possibility of a failed response is higher (in Figure 6a). On the other hand, in the second case (Case-2: 'Quick response with background processing'), the system sends a successful response immediately after getting a successful response for the original video file and starts background processing for all other versions (in Figure 6b). For both cases, the system may create orphan garbage data due to network disruption, client timeout, internal server communication error, and so on.

We present an example of how orphan garbage data is created due to network disconnection, client timeout problems, object versioning, etc. Here, the data.mp4 file is uploaded from the client (in Figure 5). Then, to maintain security and integrity, the file name is changed using some random function. After, for this single video file, the proxy server sends five requests to the storage/object server, and each version of



(a) Case-1: 'Response after all uploading'

(b) Case-2: 'Quick response with background processing'

FIGURE 6. **Diagram illustrating the flow of video uploading and transcoding for two distinct scenarios/cases [8].**

the data has three copies stored. Two versions are successfully stored when background processing, but somehow other versions are not uploaded successfully due to several reasons. In the storage node, the object is stored using the system's own convention (i.e. <epoch-time>.data). Hence, the final response is failed and the URL is not stored in the AUTH database. However, the six copies which are successfully stored through internal communication from proxy to storage nodes, will remain as the orphan garbage data.

IV. SYSTEM DESIGN AND IMPLEMENTATION

In Figure 7, we illustrate the general architecture of how we design the structure of this paper to find out the impact of object storage sustainability. At first, we focus on the impact of middleware placement in the object storage system. To do so, we propose a framework for managing multimedia data smoothly and efficiently in a media cloud storage system. Then, we investigate another problem related to orphan garbage data. We present our proposed architecture for these two important realms in the following subsections.

A. VIDEO SEGMENTER MIDDLEWARE

Nowadays, smoother and more efficient video streaming including dynamic adaptation of streaming has become popular for its diversified usages. To understand its underlying methodology, first, we need to know how streaming works in a large system. Figure 8 presents the architecture of how a streaming server and cloud system get interconnected while streaming videos to multiple clients. Here, first, the streaming server collects necessary segments or full files from the storage server. Then, the streaming manager creates small chunks from the segment to send those chunks to the clients. There is another component named the viewer server which is responsible for publishing the chunks to clients.

It is worth mentioning that, for streaming a video, the streaming server needs all the information about the video so that it can download the full video or any segments

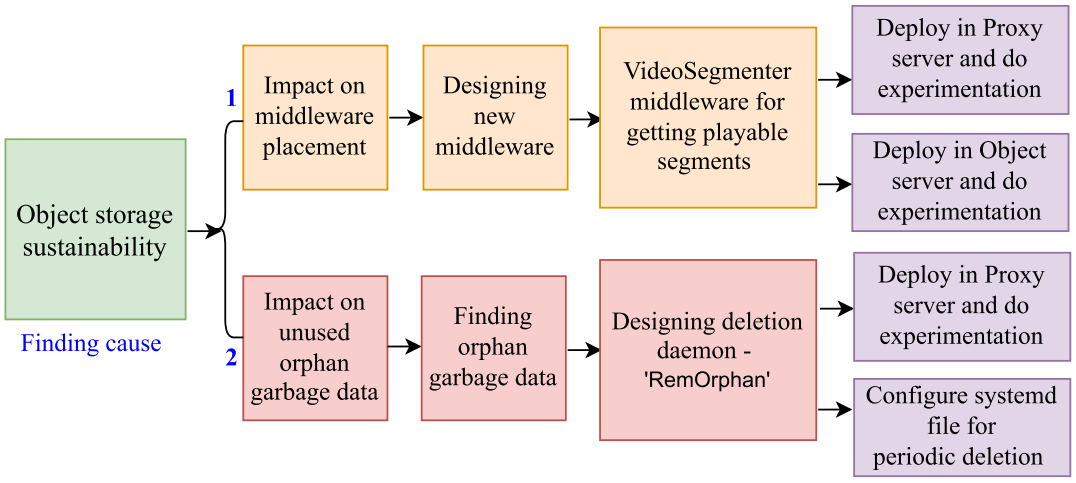


FIGURE 7. The general architecture of how we design the structure of this study to find out the impact of object storage sustainability. At first, we focus on the impact of middleware placement in object storage systems. Then, we investigate another problem related to orphan garbage data.

Algorithm 1 Algorithm for the VideoSegmenter Middleware

```
procedure VideoSegmenter(app, env, start_response) ▷ Each middleware has a start_response method which need status and headers of the response
    timeRange ← env.get(TimeRangeHeader)
    if requestMethod = GET and timeRange then
        itr ← app(env, start_response)  ▷ Here, itr is a dictionary having important attributes in OpenStack Swift, or list insatance if any error occurs
        if isinstance(itr, list) then
            return itr
        end if                                  ▷ isinstance, list both are python keyword
        sT, eT ← timeRange.split()
        validTimePattern ← CheckRegex           ▷ Check valid time pattern using Regular expressions
        outF ← SegmentedFileLocation
        outputFp, etag ← SegmentVideo(itr._data_file, outF, sT, eT)
        itr._fp ← open(outputFp, rb)
        itr._diskfile._data_file ← outputFp
        itr._obj_size ← os.path.getsize(output_fp)
        itr._etag ← etag
        UpdateHeader()                          ▷ According to new content-length and new etag of video segment
        start_response(staus[0], headers[0])
        return itr
    end if
    return app
end procedure
```

from the original file. To save unnecessary processing, the storage server emphasizes storing segments of the original file beforehand. In this case, when the storage server has no segment stored or the downloaded segment is found to be corrupted, 'VideoSegmenter' middleware comes into play. We design the middleware having it similar to that for partial range requests. The only difference is that our design takes the input of a time range header, *X-Time-Range: startTime-endTime*. Such a range request is supported by HTTP protocols and it only gives some bytes within the requested range. However, segment requests of our proposed middleware provide any playable segment within the requested time range. Figure 9 presents a time range request example of our proposed middleware along with the object storage architecture of OpenStack Swift.

As Swift has its own architecture, predefined variables, iterators, etc., for downloading an object, we need to investigate Swift object storage implementation very deeply and find out which parts of the implementation need to be refactored when returning segments rather than the full object.

Algorithm 2 Algorithm for Segmenting Video File

procedure SegmentVideo(*inP*, *otP*, *sT*, *eT*) ▷ Here, inP = InputFilePath, otP = OutputFilePath, sT = startTime, eT = endTime
  temp, *fileWOExt* ← *inP.rsplit*()
  swiftExt ← *.data*
  videoExt ← *.mp4*
  otP += *join*(*fileWOExt.clip*(*swiftExt*), *sT*, *eT*)
  matchedFile ← *glob.glob*(*otP*)
  tStamp ← *time.time*()
  if (*mathedFile*) **then**
    pathWBF, *etag* ← *matchedFile.rsplit*()
    otP ← *join*(*pathWBF*, *etag*, *tStamp*) + *videoExt*
    os.rename(*matchedFile*[0], *otP*)
    return otP, etag
  else
    tempPath ← *otp* + *randomInt* + *videoExt*
    clipCmnd ← *FFmpegCommand* ▷ FFmpeg command for clipping the video segment using sT and Et [33]
    output, *error* ← *subprocess.Popen*(*clipCmnd*).*communicate*()
    if *output* **then**
      raise Exception
    end if
    etag ← *md5*(*tempPath*) ▷ New etag calculation for segment video
    otP ← *join*(*otP*, *str*(*etag*), *str*(*tStamp*)) + *videoExt os.rename*(*tempPath*, *otP*)
    return otP, etag
  end if
end procedure

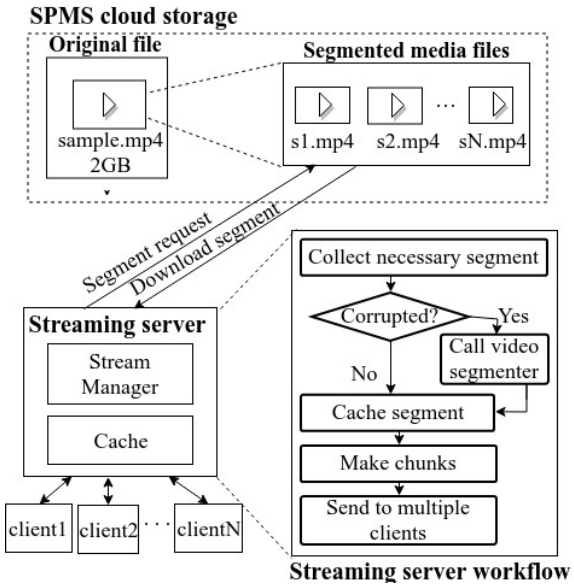


FIGURE 8. Our proposed architecture of VideoSegmenter middleware, which presents how a streaming server requests for a segment from the cloud storage.

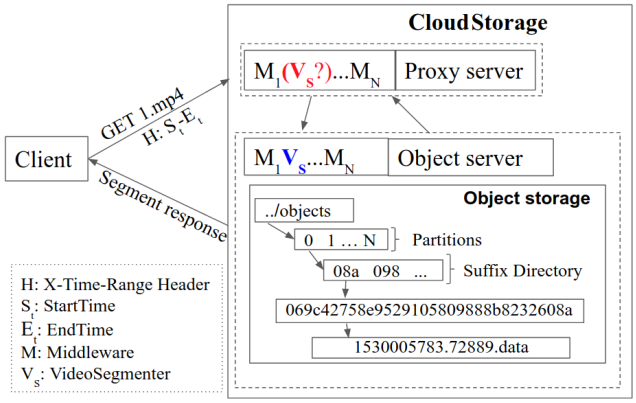


FIGURE 9. An example of segment GET request using proposed X-Time-Range-Header. (VideoSegmenter middleware (V_s) is deployed in object server.)

Moreover, to enable ease of deployment and maintenance, our target is not to change the open source code, but rather to deploy a new middleware package so that it can be deployed and integrated easily with the system.

Accordingly, we find that two classes (BaseDiskFile and DiskFileReader) from two separate files in OpenStack Swift (/usr/lib/python3.8/site-packages/swift/obj/mem_diskfile.py, /usr/lib/python3.8/site-packages/swift/obj/diskfile.py) are responsible for downloading an object. We change the values according to new segment size, etag (MD5 hash in this files of the downloading object), and location (*_fp*, *_diskfile._data_file*, *_obj_size*, *_etag*). We present the VideoSegmenter algorithm here (Algorithm 1 and 2).

Subsequently, we consider another key aspect- as the proxy server is responsible for all the processing, can we deploy VideoSegmenter in the proxy server? Here, the problem is, that for segmenting a video file, we need the full file first. Hence, the proxy server needs to download the full file in some temporary location and then clip the video to send segments to the requester. This procedure is slower and needs more network overhead. On the contrary, if we deploy VideoSegmenter in the object server, full object downloading is not needed anymore as the clipping is done directly on the data location. Here we need r times processing in all replicated locations of the original object. However, the segment will automatically be deleted from its location after a predefined time in this case.

All the middleware is written in /usr/lib/python3.8/site-packages/swift/common/middleware/ package of OpenStack Swift. If we add a new middleware there, then the upgradation of a new release will be required making the implementation process complex. Hence, we implement a new distribution of VideoSemter middleware using Python setup-tools [34]. Besides, in our deployed SPMS object server, we change the *object − server.conf* file and add VideoSegmenter egg file for including our new middleware (*use = egg : video_segmenter#video_segmenter* in paste.filter_factory [45]).

We formulate some relationship with network overhead and sync delay to our proposed architecture.

$$V_d = \sum_{0<=i<=d} V_{d_i - d_{i+n}} \tag{1}$$

$$V_d = \{V_{d_1}, V_{d_2}, V_{d_3}, \ldots, V_{d_n}\} \tag{2}$$

where, V_d = Full video file, i = Segment/chunk, n = Given time for chunk size. If the network overhead is o and sync delay is l, then the impact of processing in different servers can be denoted as -

$$\{o, l\}_{P_{N_{ob}}, S_{S_d}} < \{o, l\}_{P_{N_{px}}, S_{S_d}} < \{o, l\}_{P_{S_d}, S_{S_d}} \tag{3}$$

Here, $P_{N_{ob}}$ = Segmentation done in the object server
$P_{N_{px}}$ = Segmentation done in the proxy server
P_{S_d} = Segmentation done in the streaming server
S_{S_d} = Streaming done in the streaming server

From equation IV-A, we can see that as the range request segmentation is done in the object storage server directly, no need to download the full video in the proxy or streaming server. Hence, the network overhead and sync delay will be less in comparison to a proxy or streaming server. We present the detailed results in Section V. In the next subsection, we delineate orphan data deletion daemon pertinent to both video and image data.

B. 'RemOrphan': ORPHAN DATA DELETION

Data that is never used for quite a long time is referred to as unused data. This data can be archived using some erasure policy or using different types of mechanisms. However, there is another kind of data called orphan data or garbage data, which exposes a great threat to data storage sustainability. The main reasons for the threat of orphan data include storing each object using some random names, the existence of different types of object versions [8], [35], network connection timeout with client or AUTH database, etc. Hence, we propose an architecture for detecting such orphan data using OpenStack Swift hash rings and scripts.

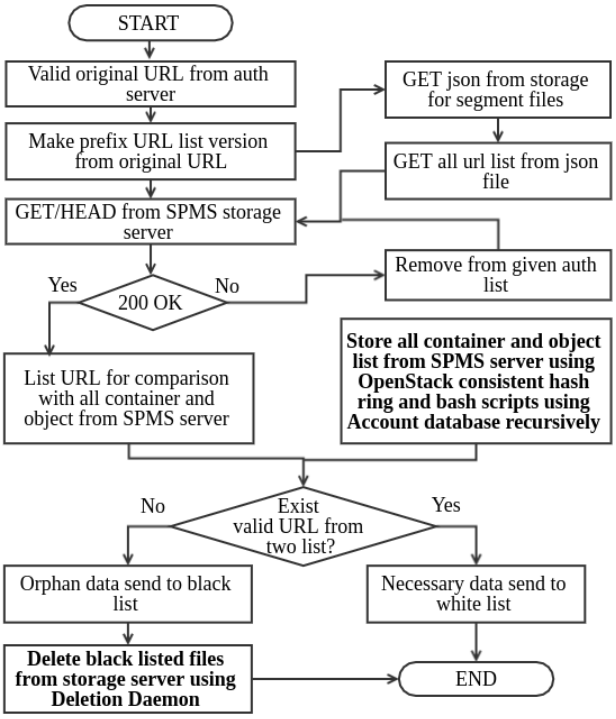


FIGURE 10. Flow diagram for orphan data deletion daemon.

It is worth mentioning that the main two design goals of OpenStack Swift and similar systems are eventual consistency and high availability through replicated data objects across multiple nodes. When dealing with data-intensive scenarios ($r > 3$ and $n \gg 1000$), the process of synchronizing objects experiences substantial delays and results in extensive network overhead. This issue is commonly known as the *sync bottleneck problem* within OpenStack Swift. Specifically, during each synchronization round, the storage node responsible for a particular data partition (referred to as P) compares its local fingerprint of P with the fingerprints of all the other replicas of P ($r - 1$ in total). This synchronization process results in a network overhead of $r(r-1)$ sync messages. Since a storage node can host multiple partitions (approximately n/h partitions, where each sync message contains h hash values), the number of exchanged hash values per storage node becomes as significant as $\theta(n \times r)$ in a single sync round. Consequently, this leads to substantial unnecessary network overhead [46]. Hence, a high number of objects are always responsible for sync delay and network overhead. Moreover,

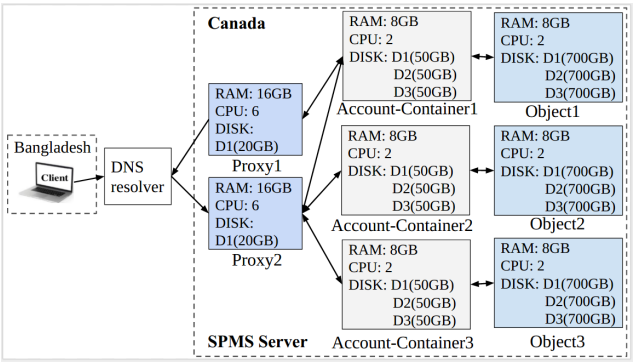


FIGURE 11. The testbed setup consists of servers located in Canada and a client situated in Bangladesh.

if this scenario happens for some unnecessary orphan data, then it appears to be a great loss for storage service providers.

To tackle this problem, we propose some key steps in our orphan data detection and deletion daemon. 1) We collect all data lists from the client/AUTH database for a certain time interval. 2) Then, we collect all object lists of all accounts from the Swift account and container database for the same time interval. 3) We create the black list and white list from all versions of the objects. 4) We delete blacklisted files using bulk DELETE request [10]. Figure 10 presents the flow diagram of the orphan data deletion daemon.

Our custom daemon server is configurable from the perspective of its considered time interval for the collection of lists. This offers us options to run once or in forever mode by daily, weekly, or monthly based on our configured time interval.

V. PERFORMANCE EVALUATION

We assess the performance of our proposed architectures by implementing them in a real-world environment. Firstly, we provide a concise overview of our experimental testbed setup. Following that, we present the experimental results and findings for each of our three distinct architectures.

A. EXPERIMENTAL TESTBED SETUP

In our testbed deployment in Canada, we utilize high-resource machines to ensure optimal performance. The media storage cluster consists of two proxy servers, three account-container servers, and three object servers. These servers are equipped with AMD Opteron 62xx class CPUs and run on CentOS 7 operating system. The memory and disk configurations of the Swift servers are as follows: 1) Two proxy servers with 8 GB of memory and 20 GB of disk storage each. 2) Three account-container servers, each with 8 GB of memory, and three disks, each with a capacity of 50 GB. 3) Three object servers, also with 8 GB of memory and three disks, each with a capacity of 700 GB. Additionally, each server is equipped with six 1 GB network interface cards. The experimental setup of our testbed is depicted in Figure 11,

TABLE 1. The configuration of machines used in the testbed setup.

Informations	Proxy server	Object server	Account-container server	Client machine
Architecture	x86_64	x86_64	x86_64	x86_64
CPU(s)	16	48	16	1
On-line CPU(s) list	0-15	0-47	0-15	0
Thread(s) per core	2	1	2	1
Core(s) per socket	4	12	4	1
Socket(s)	2	4	2	1
NUMA node(s)	2	8	2	1
CPU family	6	16	6	6
Model name	Intel(R) Xeon(R) CPU E5620 @2.40GHz	AMD Opteron(tm) Processor 6174	Intel(R) Xeon(R) CPU E5620 @2.40GHz	QEMU Virtual CPU version 1.5.3
CPU MHz	2394.141	2199.967	2394.103	2393.998
Virtualization type	VT-x	AMD-V	VT-x	full Storage

and Table 1 provides detailed information about the machines used in the setup.

Furthermore, as part of our setup, we deploy a private media cloud SPMS using OpenStack Swift, specifically the stable Newton branch. This SPMS configuration includes three replicas ($r = 3$) and 16384 partitions ($p = 16384$). For the account, container, and object ring files, we utilize nine devices. As per the OpenStack Swift guide [10], these devices are mounted in the location $/srv/node/ < server >$, resulting in approximately 5461 partitions per device. Here, a server can be an account, container, or object. In addition, we develop and implement a social site that caters to both mobile and web users. Over a period of approximately eight months, we facilitate the uploading of various types of data from clients to the development server. Besides, in order to evaluate our proposed architecture, we upload large media files from a benchmark video surveillance data set. We use 125 videos ranging in size from 3.8 MB to 1.4 GB and upload the videos in a periodic manner. Furthermore, as part of our testing, we generated a total of 10,000 accounts and 10,000 containers within the Swift cluster. Subsequently, we upload approximately 1 million images and video files into these accounts, resulting in a total object count (n) of 1 million within our testbed server. The uploaded data amounts to around 1.5 terabytes (TB), resulting in a total data size of $1.5TB \times 3$ within our development server.

1) ADDING VideoSegmenter ENTRY-POINT IN SETUP.CFG FILE

We present the relevant code for the setup.cfg file pertaining to our implementation. The setup.cfg file serves as the configuration file for OpenStack Swift. All the middleware of OpenStack Swift is developed using a filter factory pattern in Paste package [10]. Hence, it is necessary to include each middleware in the entry-point section of the setup.cfg file.

```
[metadata]
name = swift
. . .
[pbr]
skip_authors = True
. . .
[files]
packages = swift
scripts = bin/swift-account-audit
. . .
[entry_points]
paste.filter_factory = healthcheck =
swift.common.middleware.healthcheck:filter_factory
. . .
videosegmenter = swift.common.middleware
.videosegmenter : filter_factory
%Ourimplementation
. . .
```

Here, "[ ]" denotes several sections for setup.cfg file, and ". . ." refers existence of more codes in the file that we omit here.

2) ADDING VideoSegmenter MIDDLEWARE IN PROXY-SERVER.CONF FILE

In this Section, we present the configuration change needed in the proxy-server.conf file relevant to our implementation. The proxy-server.conf file serves as the configuration file for the proxy server in OpenStack Swift. The majority of the middleware implementations are operated through the proxy server. In order to assess the impact of middleware placement, we include our developed middleware in the proxy-Server.conf file in the first experimentation.

```
[DEFAULT]
# bind_ip = 0.0.0.0
bind_port = 8080
. . .
[pipeline:main]
pipeline = catch_errors gatekeeper healthcheck
proxy-logging · · · videosegmenter · · · proxy-logging
proxy-server % our implementation
[app:proxy-server]
use = egg:swift#proxy
[filter:catch_errors]
use = egg:swift#catch_errors
[filter:healthcheck]
use = egg:swift#healthcheck
. . .
[filter : videosegmenter]% our implementation
use = egg : swift#videosegmenter
% our implementation
. . .
```

Within the [pipeline:main] section, the various middleware implementations are listed and separated by spaces. The order in which the middleware implementations are listed in the pipeline determines the sequence in which they are called. The pipeline includes middleware implementations such as catch_errors, gatekeeper, health check, proxy-logging, and others, which are part of the OpenStack Swift framework. Additionally, we develop and include VideoSeg- menter middleware in the pipeline.

3) ADDING VideoSegmenter MIDDLEWARE IN OBJECT-SERVER.CONF FILE

Object-Server.conf is the object server configuration file of OpenStack Swift. To find out the impact of middleware placement in the object server, later we include the developed middleware in Object-Server.conf file. We present the code of Object-Server.conf file pertinent to our implementation below:

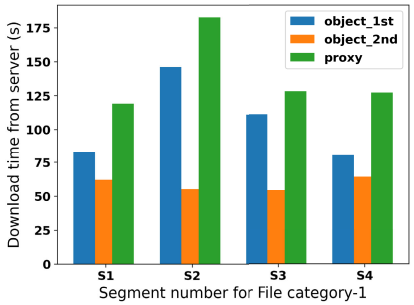
Here, we present the configuration change needed in the object-server.conf file relevant to our implementation. The object-server.conf file serves as the configuration file for the object server in OpenStack Swift. To find out the impact of middleware placement in the object server, later, we include the developed middleware in object-server.conf file.

```
[DEFAULT]
# bind_ip = 0.0.0.0
bind_port = 6200
. . .
[pipeline:main]
pipeline = healthcheck recon · · · videosegmenter · · ·
object-server % our implementation
[app:object-server]
use = egg:swift#object
[filter:healthcheck]
use = egg:swift#healthcheck
. . .
[filter : videosegmenter]% our implementation
use = egg : swift#videosegmenter
% our implementation
. . .
```

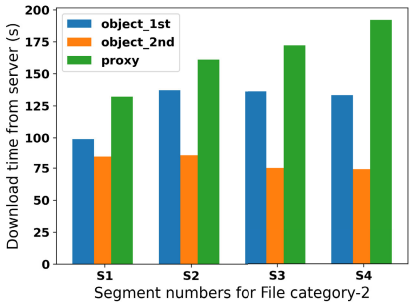
Here, within the [pipeline:main] section, the various middleware implementations are listed and separated by spaces for the object-server.conf file. We develop and include VideoSegmenter middleware in the pipeline.

B. EXPERIMENTAL RESULTS

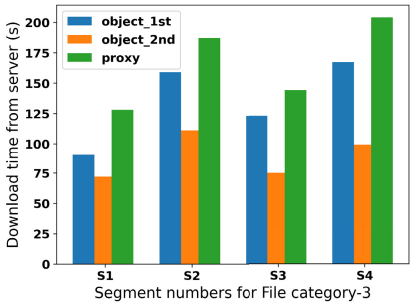
For testing VideoSegmenter middleware, we make different segments of 10 minutes and 15 minutes of different video files (Category-1 having an average size of 0.49GB and average duration of 94.85 minutes; Category-2: average size of 0.66GB and average duration of 152.2 minutes; Category-3: average size 0.87GB and average duration 177.7 minutes). We deploy VideoSegmenter middleware in both the proxy server and the object server. There was some difference in the middleware code for the proxy server. In our proposed architecture of VideoSegmenter for object servers, we store or cache the segment for around one to two days in the object segment location. Hence, a download request of the



(a) Download time for four segments for file category-1



(b) Download time for four segments for file category-2



(c) Download time for four segments for file category-3

FIGURE 12. Comparison of download time for segments of three different file categories. S1, S2, S3, and S4 denote the average segment of 10 and 15 minutes of 1st to 4th segments respectively. In the graph, 1st bar (blue) represents the download time of the segment from the object server at the first time request, 2nd bar (orange) represents the same segment download time from the object server at the second time request. Besides, 3rd bar (green) represents the download time of the same segment from the proxy server.

TABLE 2. Time improvement percentage status for different segments with respect to retrieving the segment from 2nd time vs 1st time from object server if we place the middleware in object server. Moreover, segment download time comparison is presented by placing the middleware in the proxy server (object vs. proxy).

File category			% Time improvement (1st segment)		% Time improvement (2nd segment)	
	Avg. Size (GB)	Avg. Duration (min)	Object_2nd vs Object_1st	Object vs Proxy	Object_2nd vs Object_1st	Object vs Proxy
Short file	0.49	94.85	25.3	30.25	62.33	20
Medium file	0.66	152.2	15.15	25	37.96	14.91
Long file	0.87	177.7	20	29.69	30.19	14.97
			% Time improvement (3rd segment)		% Time improvement (4th segment)	
File category	Avg. Size (GB)	Avg. Duration (min)	Object_2nd vs Object_1st	Object vs Proxy	Object_2nd vs Object_1st	Object vs Proxy
Short file	0.49	94.85	50.45	13.28	19.75	36.22
Medium file	0.66	152.2	44.12	20.93	43.61	30.73
Long file	0.87	177.7	38.21	14.58	40.72	18.14
Avg time improvement		**Object_2nd vs Object_1st: 22.39%**			**Object vs Proxy: 35.65%**	

same segment (second download request onward) needs a lower time than the first-time request from the object server. Figure 12 and Table 2 present a comparison of download times from the object server and proxy server for three different categories. Here, we show download times for different segments of different files. We take 100 iterations for every single segment for each type of download and present an average of the 100 iterations in the graph. The download times correspond to three different types of download- 1) download from the object server for the first time, 2) download from the object server for the second time, and 3) download from the proxy server. As Figure 12 demonstrates, downloading from the object server always takes much less time than that from the proxy server. Nevertheless, the second download from the object server takes less time than the first download utilizing the caching.

Furthermore, Table 3 presents that around 35% of data is orphan data according to our setup testbed at the first round when we run the deletion daemon. After removing the orphan data, sync delay and network overhead get lower by up to 25% and 30% respectively. Next, several users upload a bulk amount of images and videos regularly and we run the deletion daemon after one year again. Table 3 presents the values after removing the new orphan data in the second experiment. Furthermore, we run the deletion daemon after two days of the second experiment. Table 3 presents the values after running the deletion daemon a third time.

TABLE 3. Orphan garbage data deletion status per node for testbed server.

1st round - Deletion status

Metrics	Before deletion	After deletion	Improvement
Object count	1M	650K	lower 35% data
Sync delay	1440s	1080s	lower 25% delay
Network overhead	500MB	350MB	lower 30% overhead

2nd round - Deletion status after one year

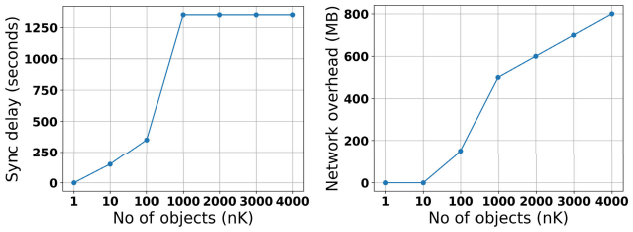
Metrics	Before deletion	After deletion	Improvement
Object count	1G	630M	lower 30% data
Sync delay	1280s	965s	lower 25% delay
Network overhead	500MB	350MB	lower 30% overhead

3rd round - Deletion status after two days

Metrics	Before deletion	After deletion	Improvement
Object count	700M	678M	lower 4% data
Sync delay	1280s	1100s	lower 20% delay
Network overhead	500MB	350MB	lower 30% overhead

C. OVERHEAD ANALYSIS

Figure 13 presents the relationship between sync delay (in seconds) and network overhead (in MB) concerning the number of objects (n) per node/server, with a replica count



(a) (b)

FIGURE 13. Relation between sync delay and network overhead with respect to no of objects per node (n) . Here, the value mentioned as nK (in x-axis), i.e., 10 value represents 10,000 objects.

TABLE 4. CPU and memory overhead for video Segmentation and orphan data deletion.

Metrics	Video Segmentation	Orphan data deletion
CPU	3%	14%
Memory	5%	16%

(r) of 3. As the value of n increases, the sync delay also increases in proportion to r. However, it is worth noting an interesting observation that when n surpasses 1 million, the sync delay exhibits a relatively slower growth rate (for a fixed r). This phenomenon can be attributed to the number of partitions involved [46]. When n exceeds 1 million and becomes significantly larger, such as 2.5 million, the number of sync messages remains stable. Nevertheless, the size of each sync message continues to increase with n, as each message contains additional hash values of more data objects. Consequently, the network overhead continues to grow with n when n exceeds 1 million. Moreover, Table 4 provides information on the memory and CPU overhead of the overall architecture.

VI. DISCUSSION

This study, for the first time in the literature, establishes the necessity of object storage sustainability as long-term storage has diverse effects such as performance, efficiency, energy consumption, fault tolerance, and so on. We investigate two concerns related to object storage sustainability - the effect of middleware placement in proxy or object servers, and the impact of unused garbage data due to the offline processing of multimedia data. To do so, we design and develop a new video segmenter middleware in OpenStack Swift, examine the efficiency of placement, and implement a deletion daemon.

A. COMPUTATIONAL COMPLEXITY ANALYSIS

We propose a system for finding orphan garbage data and removing the data from the object storage system. Some studies present distributed memory garbage collector, however, these studies are not similar to our proposed system. In this Section, we delineate computational complexities of commonly used garbage collection algorithms i.e. Mark and Sweep, Copying (e.g., Cheney's algorithm), Mark and Compact, Generational, Distributed Garbage Collector, and our proposed one. Table 5 presents some commonly used garbage collection algorithms and their associated computational complexities.

TABLE 5. Computational complexities of commonly used garbage collection algorithms. Here, k represents the number of live objects in the heap, b represents the number of live objects being traced and copied, N signifies the total count of files existing in the filesystem, m refers to the count of immediate child elements present under a specific directory, and a indicates the number of files stored within a particular directory.

Algorithm	Type	Computational Complexity
Mark and Sweep [47]	Memory Garbage Collector	$O(k)$
Copying (e.g., Cheney's) [48]	Memory Garbage Collector	$O(b)$
Mark and Compact [49]	Memory Garbage Collector	$O(k)$
Generational [50]	Memory Garbage Collector	Varies (typically sublinear: $O(sqrt(k))$ or $O(log(k))$)
Distributed [23]	Memory Garbage Collector	Varies based on algorithm and system architecture
RemOrphan (our proposed)	Orphan Data Collector	$O(mlogN) + O(a + logN)$

The complexity of distributed garbage collection can vary significantly depending on the specific algorithm and the underlying distributed system architecture. The complexity analysis for distributed garbage collection is often more nuanced, considering factors such as message overhead, graph traversal, synchronization, load balancing, scalability, and fault tolerance. The complexity of distributed garbage collection is typically higher than that of single-node garbage collectors due to the added challenges of coordination and communication across multiple nodes.

Moreover, we demonstrate the computational complexity of different object storage systems and data structures [51]. We consider Consistent Hash (CH), Content Addressable Storage (with Multi-Layer Index), Compressed Snapshot, OpenStack Swift (CH with a File-Path DB), and our proposed system. Table 6 presents a qualitative comparison of time and storage complexity for different object storage systems using several data structures. The time needed for listing and copying a directory using OpenStack Swift hash rings is $O(mlogN)$ and $O(a + logN)$ respectively. Hence, the time complexity of our proposed *'RemOrphan'* algorithm is $O(mlogN) + O(a + logN)$.

B. COMPARATIVE ANALYSIS

In this Section, we discuss some important aspects of our study and present a comparison of related research studies according to several features. In Table 7, we illustrate and compare the features such as algorithm type, memory and storage management, developers' deployment effort, intended platform, focus metrics, orphan data collection,

TABLE 6. A comparison of computational complexity for different object storage systems using several data structures [51]. Here, *N* signifies the total count of files existing in the filesystem, *m* refers to the count of immediate child elements present under a specific directory, *a* indicates the number of files stored within a particular directory and *R* is the resizing and transcoding time.

Computational complexity	Consistent Hash (CH)	Content Addressable Storage	Compressed Snapshot	OpenStack Swift	Our proposed system
File Access	$O(1)$	$O(1)$	$O(N)$	$O(1)$	$O(1)$
MKDIR	$O(1)$	$O(N)$	$O(1)$	$O(1)$	$O(1)$
RMDIR, MOVE	$O(a)$	$O(N)$	$O(N)$	$O(a)$	$O(a)$
LIST	$O(N)$	$O(m)$	$O(N)$	$O(mlogN)$	$O(mlogN)$
COPY	$O(N)$	$O(N)$	$O(N)$	$O(a+logN)$	$O(a+logN)$
Store/PUT	-	-	-	$O(a+logN)$	$O(a+logN)+O(R)$
Retrieve/GET	-	-	-	$O(R)+O(mlogN)$	$O(mlogN)$
Garbage data deletion	-	-	-	-	$O(mlogN)+O(a+logN)$

middleware placement, and deployment, etc. In the algorithm feature, we discuss what kind of algorithm the studies present in their work.

Some studies use the Modified Garbage Collector algorithm, i.e., they extended the traditional Garbage Collection algorithms [41], [42], [52], [54], [55], [56], [57]. On the other hand, several studies do not modify the traditional Garbage collection algorithms, hence we refer to them as Unmodified Garbage Collector [39], [40], [43]. From the management perspective, we discover mainly four types of categories - Memory Garbage Collector [38], [39], [40], [41], [42], [43], [52], [54], [55], [56], [57], Garbage VM Collector [24], [53], Orphan Process Collector [26], [27], and Orphan Garbage Data Collector.

However, no existing literature focuses on the later issue i.e. orphan garbage data. Besides, we present the system deployment effort as high, low, medium, and none with respect to the algorithms proposed in recent studies. Next, the target platform and performance metrics are presented in the comparison table. Most of the studies focus on the metrics - throughput and latency. Besides, study [7] works on throughput, latency, and concurrency whereas our proposed architecture improves the throughput, latency, and concurrency and makes the system more fault-tolerant. By adopting a completely new methodology, we can achieve long-term object storage sustainability by analyzing the proper middleware placement and removing orphan garbage data regularly using a deletion daemon.

C. LIMITATIONS OF THE PROPOSED SYSTEM

However, our proposed system may have certain limitations, including:

1) COMPLEXITY

OpenStack object storage middlewares are complex to deploy, configure, and manage. Hence, to reproduce our proposed system, any vendor requires a deep understanding of the underlying architecture and components, making it challenging for organizations with limited expertise or resources.

2) COMPATIBILITY

Our proposed middleware and the deletion daemon may have compatibility issues with certain applications or legacy systems. This can require additional effort for integration and may limit interoperability in multi-cloud or hybrid-cloud environments.

3) TIME DELAY

The deletion process in our system can introduce a time delay before objects are completely removed. This delay is due to the design considerations for data durability and fault tolerance.

4) METADATA CLEANUP

When objects are deleted, the associated metadata may not be immediately removed. Over time, this can lead to a buildup of unused metadata, which can impact system performance and storage utilization. Proper metadata cleanup mechanisms need to be in place to ensure efficient resource management.

5) INCOMPLETE DELETIONS

In certain scenarios, the deletion process may encounter failures or inconsistencies, resulting in incomplete deletions. This can leave remnants of deleted objects or metadata in the system, potentially impacting storage capacity and data integrity.

6) SCALABILITY CHALLENGES

As the proposed system scales and handles a large number of deletions, the deletion daemon may face scalability challenges. Our system needs to efficiently manage and prioritize deletion requests to ensure timely and accurate removal of objects.

7) SYNCHRONIZATION DELAY

In the proposed systems, the deletion process may involve synchronization across multiple storage nodes or data centers. Synchronization delays can occur, which means that the deletion may not be immediately reflected across all replicas or locations, potentially leading to inconsistencies in data availability.

To overcome these limitations, we plan to design appropriate data life-cycle management strategies, backup mechanisms, and monitoring processes to address the challenges associated with the deletion daemon in object storage systems.

TABLE 7. A qualitative comparison of related research studies according to several features such as algorithm type, memory and storage management, developers deployment effort, orphan data collection, middleware placement, and deployment, etc. Here, GC refers to garbage collection.

Research study	Algorithm	Management	Deployment effort	Platform	Focus metrics	Middleware placement	Orphan garbage collector
Althaus et al., 2022 [38]	Greedy Garbage Collection	Memory GC	Medium	Solid State Drives (SSDs)	Throughput	X	X
Noor et al., 2022 [3]	Modified scan scripts and DCT quantization	Storage management, no orphan data deletion	Low	Object storage	Throughput, Latency	✓	X
PokeMem (Kweun et al., 2022) [52]	Modified GC	Memory GC	Medium	Processing (Enhanced Spark)	Throughput	X	X
Noor et al., 2021 [7]	Resizing and security enforcement	Storage management, no orphan data deletion	Low	Object storage	Throughput, Latency, Concurrency	✓	X
GC-CR (Louati et al., 2017) [24]	Checkpoint-Restart	Decentralized GC (snapshot)	High	Storage	Latency	X	X
iCSI (Kim et al., 2017) [53]	Lightweight VM collector	Cloud Garbage VM Collector	High	Stoarge	Latency	X	X
NG2C (Bruno et al., 2017) [54]	Modified GC	Memory GC	Low	Processing, storage	Latency	X	X
Deca (Lu et al., 2016) [40]	Unmodified GC	Memory GC	High	Processing (Spark)	Throughput	X	X
Taurus (Maas et al., 2016) [39]	Unmodified GC	Memory GC	Low	Processing, storage	Latency	X	X
Broom (Gog et al., 2015) [42]	Modified GC	Memory GC	High	Processing (Naiad)	Throughput	X	X
FACADE (Nguyen et al., 2015) [43]	Unmodified GC	Memory GC	Low	Iterative processing	Throughput	X	X
NumaGiC (Gidra et al., 2015) [41]	Modified GC	Memory GC	None	Processing, storage	Throughput	X	X
DSA (Cohen and Petrank et al., 2015) [55]	Modified GC	Memory GC	Medium	Processing, storage	Throughput	X	X
Sabbaghi et al., 2013 [27]	Orphan process detection	Remote Procedure Call	None	Processing	Fault tolerance	X	X
C4 (Tene et al., 2011) [56]	Modified GC	Memory GC	None	Processing, storage	Latency	X	X
Jahanshahi et al., 2005 [26]	Orphan process detection	Remote Procedure Call	None	Processing	Fault tolerance	X	X
G1 (Detlefs et al., 2004) [57]	Modified GC	Memory GC	None	Processing, storage	Latency	X	X
***RemOrphan* (our proposed)**	**Orphan garbage data collector**	**Storage management**	**Low**	**Processing, object storage**	**Throughput, Latency, Concurrency, Fault tolerance**	✓	✓

VII. CONCLUSION AND FUTURE WORK

In this paper, we delineate three important realms related to multimedia data management on the cloud and the diverse effects on storage sustainability. Here, we point to three key vacancies in the literature comprising retrieval of video streaming data, middleware placement based on their responsibility, and detection and deletion of orphan garbage data (a new type of data that is of no use but retained for a long time over cloud storage). Hence, we designed a new middleware in the object server for downloading time interval playable video segments which can be easily integrated in OpenStack Swift and similar systems such as SPMS.

Furthermore, we propose a mechanism for removing orphan garbage data from cloud storage. We perform rigorous experimentation over a real setup established in Canada and accessed from Bangladesh. Our experimentation covers both system-level and subjective evaluations. The evaluation results confirm that we can achieve substantial performance improvement using our proposed mechanisms. Our future work includes the exploration of SSYNC for account, container, and object servers using multiple replicas. We also plan to explore recursive deletion daemon algorithms using different hash rings. Besides, experimenting with different server setups with large-scale simulations is yet another aspect worth investigating in the future.

ACKNOWLEDGMENT

(Jannatun Noor and Najla Abdulrahman Al-Nabhan contributed equally to this work.)

REFERENCES

[1] J. Aguilar-Armijo, C. Timmerer, and H. Hellwagner, "SPACE: Segment prefetching and caching at the edge for adaptive video streaming," *IEEE Access*, vol. 11, pp. 21783–21798, 2023.

[2] L. Pu, J. Shi, X. Yuan, X. Chen, L. Jiao, T. Zhang, and J. Xu, "EMS: Erasure-coded multi-source streaming for UHD videos within cloud native 5G networks," *IEEE Trans. Mobile Comput.*, early access, Jan. 20, 2023, doi: 10.1109/TMC.2023.3238356.

[3] J. Noor, M. N. H. Shanto, J. J. Mondal, M. G. Hossain, S. Chellappan, and A. A. Al Islam, "Orchestrating image retrieval and storage over a cloud system," *IEEE Trans. Cloud Comput.*, vol. 11, no. 2, pp. 1794–1806, Apr./Jun. 2023.

[4] S. Brume. *Improve Sustainability: By Storing More Data*. Accessed: Jun. 13, 2023. [Online]. Available: https://community.ibm.com/community/user/storage/blogs/shawn-brume1/2022/02/15/improve-sustainability-by-storing-more-data

[5] Spherical Insights. *Global Next-Generation Data Storage Market Insights Forecasts to 2032*. Accessed: Jun. 13, 2023. [Online]. Available: https://www.sphericalinsights.com/reports/next-generation-data-storage-market

[6] N. Al-Nabhan, S. Alenazi, S. Alquwaifili, S. Alzamzami, L. Altwayan, N. Alaloula, R. Alowaini, and A. A. Al Islam, "An intelligent IoT approach for analyzing and managing crowds," *IEEE Access*, vol. 9, pp. 104874–104886, 2021.

[7] J. Noor, S. I. Salim, and A. A. Al Islam, "Strategizing secured image storing and efficient image retrieval through a new cloud framework," *J. Netw. Comput. Appl.*, vol. 192, Oct. 2021, Art. no. 103167.

[8] J. Noor, H. I. Akbar, R. A. Sujon, and A. A. Al Islam, "Secure processing-aware media storage (SPMS)," in *Proc. IEEE 36th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2017, pp. 1–8.

[9] J. Noor, R. H. Ratul, M. R. A. Uday, J. J. Mondal, M. S. I. Sakif, and A. A. Al Islam, "Sherlock in OSS: A novel approach of content-based searching in object storage system," 2023, *arXiv:2303.02105*.

[10] J. Arnold, *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*. Springfield, MO, USA: O'Reilly, 2015.

[11] L. Goddard and D. Seeman, "Negotiating sustainability: Building digital humanities projects that last," in *Doing More Digital Humanities*. Evanston, IL, USA: Routledge, 2019, pp. 38–57.

[12] C. J. Corbett, "How sustainable is big data?" *Prod. Operations Manag.*, vol. 27, no. 9, pp. 1685–1695, 2018.

[13] M. Yamin, "Managing crowds with technology: Cases of Hajj and Kumbh Mela," *Int. J. Inf. Technol.*, vol. 11, no. 2, pp. 229–237, Jun. 2019.

[14] A. Rahman, E. Hassanain, and M. S. Hossain, "Towards a secure mobile edge computing framework for Hajj," *IEEE Access*, vol. 5, pp. 11768–11781, 2017.

[15] A. Ahmad, M. A. Rahman, F. U. Rehman, A. Lbath, I. Afyouni, A. Khelil, S. O. Hussain, B. Sadiq, and M. R. Wahiddin, "A framework for crowd-sourced data collection and context-aware services in Hajj and Umrah," in *Proc. IEEE/ACS 11th Int. Conf. Comput. Syst. Appl. (AICCSA)*, Nov. 2014, pp. 405–412.

[16] M. Chen, "AMVSC: A framework of adaptive mobile video streaming in the cloud," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2012, pp. 2042–2047.

[17] X. Wang, M. Chen, T. T. Kwon, L. Yang, and V. C. Leung, "AMES-cloud: A framework of adaptive mobile video streaming and efficient social video sharing in the clouds," *IEEE Trans. Multimedia*, vol. 15, no. 4, pp. 811–820, Jun. 2013.

[18] C. Ben Ameur, E. Mory, and B. Cousin, "Evaluation of gateway-based shaping methods for HTTP adaptive streaming," in *Proc. IEEE Int. Conf. Commun. Workshop (ICCW)*, Jun. 2015, pp. 1777–1782.

[19] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia, "A survey on quality of experience of HTTP adaptive streaming," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, pp. 469–492, 1st Quart., 2015.

[20] Y. Jin, Y. Wen, and C. Westphal, "Optimal transcoding and caching for adaptive streaming in media cloud: An analytical approach," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 12, pp. 1914–1925, Dec. 2015, doi: 10.1109/TCSVT.2015.2402892.

[21] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: Facebook's photo storage," in *Proc. OSDI*, vol. 10, 2010, pp. 1–8.

[22] L. Xu, T. Guo, W. Dou, W. Wang, and J. Wei, "An experimental evaluation of garbage collectors on big data applications," in *Proc. 45th Int. Conf. Very Large Data Bases (VLDB)*, 2019, pp. 1–14.

[23] R. Bruno and P. Ferreira, "A study on garbage collection algorithms for big data environments," *ACM Comput. Surv.*, vol. 51, no. 1, pp. 1–35, Jan. 2019.

[24] T. Louati, H. Abbes, C. Cérin, and M. Jemni, "GC-CR: A decentralized garbage collector component for checkpointing in clouds," in *Proc. 29th Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Oct. 2017, pp. 97–104.

[25] S. R. J. Ramson and D. J. Moni, "Wireless sensor networks based smart bin," *Comput. Electr. Eng.*, vol. 64, pp. 337–353, Nov. 2017.

[26] M. Jahanshahi, K. Mostafavi, M. S. Kordafshari, M. Gholipour, and A. T. Haghighat, "Two new approaches for orphan detection," in *Proc. 19th Int. Conf. Adv. Inf. Netw. Appl.*, 2005, pp. 461–464.

[27] A. Sabbaghi, "A new approach to detect and eliminate orphan processes," *J. Mech. Eng. Vibrat.*, vol. 4, no. 3, pp. 13–19, 2013.

[28] J. Joshi, J. Reddy, P. Reddy, A. Agarwal, R. Agarwal, A. Bagga, and A. Bhargava, "Cloud computing based smart garbage monitoring system," in *Proc. 3rd Int. Conf. Electron. Design (ICED)*, Aug. 2016, pp. 70–75.

[29] K. M. Ramokapane, A. Rashid, and J. M. Such, "Assured deletion in the cloud: Requirements, challenges and future directions," in *Proc. ACM Cloud Comput. Secur. Workshop*, Oct. 2016, pp. 97–108.

[30] Y. Chen, K. Sherren, M. Smit, and K. Y. Lee, "Using social media images as data in social science research," *New Media Soc.*, vol. 25, no. 4, pp. 849–871, Apr. 2023.

[31] P. Cuesta-Valiño, P. Gutiérrez-Rodríguez, and P. Durán-Alamo, "Why do people return to video platforms? Millennials and centennials on TikTok," *Media Commun.*, vol. 10, no. 1, pp. 198–207, 2022.

[32] A. McCrow-Young, "Approaching Instagram data: Reflections on accessing, archiving and anonymising visual social media," *Commun. Res. Pract.*, vol. 7, no. 1, pp. 21–34, Jan. 2021.

[33] *FFMPEG*. Accessed: Mar. 28, 2023. [Online]. Available: https://www.ffmpeg.org/

[34] *SetupTools*. Accessed: Mar. 28, 2023. [Online]. Available: https://pypi.org/project/setuptools/

[35] J. Noor and A. A. Al Islam, "IBuck: Reliable and secured image processing middleware for OpenStack swift," in *Proc. Int. Conf. Netw., Syst. Secur. (NSysS)*, Jan. 2017, pp. 144–149.

[36] A. Seema, L. Schwoebel, T. Shah, J. Morgan, and M. Reisslein, "WVSNP-DASH: Name-based segmented video streaming," *IEEE Trans. Broadcast.*, vol. 61, no. 3, pp. 346–355, Sep. 2015.

[37] Z. Liu and Y. Wei, "Hop-by-hop adaptive video streaming in content centric network," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2016, pp. 1–7.

[38] E. Althaus, P. Berenbrink, A. Brinkmann, and R. Steiner, "On the optimality of the greedy garbage collection strategy for SSDs," in *Proc. IEEE 42nd Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2022, pp. 78–88.

[39] M. Maas, K. Asanović, T. Harris, and J. Kubiatowicz, "Taurus: A holistic language runtime system for coordinating distributed managed-language applications," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 457–471, Jun. 2016.

[40] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng, "Lifetime-based memory management for distributed data processing systems," 2016, *arXiv:1602.01959*.

[41] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen, "NumaGiC: A garbage collector for big data on big NUMA machines," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 661–673, May 2015.

[42] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard, "Broom: Sweeping out garbage collection from big data systems," in *Proc. 15th Workshop Hot Topics Operating Syst.*, 2015, pp. 1–7.

[43] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, "FACADE: A compiler and runtime for (almost) object-bounded big data applications," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 675–690, 2015.

[44] J. Noor, R. H. Ratul, M. S. Basher, J. A. Soumik, S. Sadman, N. J. Rozario, R. Reaz, S. Chellappan, and A. A. Al Islam, "Secure processing-aware media storage and archival (SPMSA)," 2023, doi: 10.2139/ssrn.4556078.

[45] *Paste deployment*. Accessed: Mar. 28, 2023. [Online]. Available: http://pastedeploy.readthedocs.io/en/latest/#paste-filter-factory

[46] T. T. Chekam, E. Zhai, Z. Li, Y. Cui, and K. Ren, "On the synchronization bottleneck of OpenStack swift-like cloud storage systems," in *Proc. IEEE INFOCOM 35th Annu. IEEE Int. Conf. Comput. Commun.*, Apr. 2016, pp. 1–9.

[47] B. Zorn, "Comparing mark-and sweep and stop-and-copy garbage collection," in *Proc. ACM Conf. LISP Funct. Program.*, May 1990, pp. 87–98.

[48] L. Birkedal, N. Torp-Smith, and J. C. Reynolds, "Local reasoning about a copying garbage collector," in *Proc. 31st ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, Jan. 2004, pp. 220–231.
[49] R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Hoboken, NJ, USA: Wiley, 1996.
[50] D. Doligez and X. Leroy, "A concurrent, generational garbage collector for a multithreaded implementation of ML," in *Proc. 20th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 1993, pp. 113–123.
[51] M. Zhao, Z. Li, E. Zhai, G. Tyson, C. Qian, Z. Li, and L. Zhao, "H2Cloud: Maintaining the whole filesystem in an object storage cloud," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, pp. 1–10.
[52] M. Kweun, G. Kim, B. Oh, S. Jung, T. Um, and W.-Y. Lee, "Pokémem: Taming wild memory consumers in apache spark," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Jun. 2022, pp. 59–69.
[53] I. K. Kim, S. Zeng, C. Young, J. Hwang, and M. Humphrey, "ICSI: A cloud garbage VM collector for addressing inactive VMs with machine learning," in *Proc. IEEE Int. Conf. Cloud Eng.*, Apr. 2017, pp. 17–28.
[54] R. Bruno, L. P. Oliveira, and P. Ferreira, "NG2C: Pretenuring garbage collection with dynamic generations for HotSpot big data applications," in *Proc. ACM SIGPLAN Int. Symp. Memory Manag.*, Jun. 2017, pp. 2–13.
[55] N. Cohen and E. Petrank, "Data structure aware garbage collector," in *Proc. Int. Symp. Memory Manag.*, Jun. 2015, pp. 28–40.
[56] G. Tene, B. Iyengar, and M. Wolf, "C4: The continuously concurrent compacting collector," in *Proc. Int. Symp. Memory Manag.*, Jun. 2011, pp. 79–88.
[57] D. Detlefs, C. Flood, S. Heller, and T. Printezis, "Garbage-first garbage collection," in *Proc. 4th Int. Symp. Memory Manag.*, Oct. 2004, pp. 37–48.

JANNATUN NOOR received the B.Sc., M.Sc., and Ph.D. degrees in computer science and engineering from the Bangladesh University of Engineering and Technology (BUET). She is currently an Assistant Professor with the Department of CSE, School of Data and Sciences, BRAC University. Besides, she is also a Graduate Research Assistant under the supervision of Prof. Dr. A. B. M. Alim Al Islam with the Department of CSE, BUET. Prior to BRACU, she was a Team Lead of the IPV-Cloud Team, IPvision Canada Inc. Her research interests include cloud computing, HCI, ML, wireless networking, data mining, big data analysis, and the Internet of Things.

NAJLA ABDULRAHMAN AL-NABHAN received the B.S. (Hons.) and master's degrees in computer science from King Saud University, and the Ph.D. degree in computer science from the College of Computer and Information Science (CCIS), King Saud University. She was also a Visiting Ph.D. Student with George Washington University (2012–2013). She is currently an Assistant Professor in computer science with CCIS, King Saud University. Her research interests include the Internet of Things, emergency and crowd management systems, wireless sensor networks, mobile computing, distributed systems, cloud and fog computing, smart grid, and network security.

A. B. M. ALIM AL ISLAM received the B.Sc. and M.Sc. (Engineering) degrees in computer science and engineering from the Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, and the Ph.D. degree in computer science and engineering from the School of ECE, Purdue University, West Lafayette, USA, in 2012. He is currently a Professor with the Bangladesh University of Engineering and Technology. His research interests include wireless networks, embedded systems, modeling and simulation, computer networks security, and reliability analysis.