Managed Video as a Service (MVaaS)

Main Document

DTU-Compute 2014-15

Internal Comments

This is a coarse structure for a document which can lead to a thesis. This document shall contain all the considerations, decisions, and designs made during the project, also those which have been discarded. It can also for the basis for writing articles. dated 14.07.14 The document has to use a uniform and consistent notation. Therefore do not copy and paste from other documents.

Christian suggestions: What are the challenges we get while writing instead of reading. Specifically, explain differences between writing and reading. Most of the existing systems have many optimization techniques on reading so how those techniques are differ in writing.

Needs to update the document with analysis of why cannot we use existing systems.

Updates:
section 1.2 is updated with current Milestone system architecture defects to give motivation for a new system for MVaaS after a brief discussion with Morten (MS) on 21.10.14

Discussion with Milestone Systems on 17.12.2014. Morten's notes for Karun's questions

1. Specific component failure, for instance, if serverA fails how the connected

cameraA can react and where the cameraA produced data can go.

- Again, the camera does not initiate the connection, so the camera does not react in any way! That means that if a server loses connection to the camera, the video produced until the server reconnects is lost. There are a couple of scenarios where the video from the downtime can be saved:
  (a) If there is a failover server assigned to serverA, then the failover will kick in and connect to the camera, thereby saving the video. The video produced in the duration when serverA failed and until the failover kicks in is lost.
  (b) If the camera supports edge storage and will be able to store video until serverA is back online or a failover kicks in. If the camera exceeds its edge storage capacity before a server has downloaded it, the video is lost.
- For some cameras, we can configure the camera to send an event to the server that there is motion. The server can then connect and get the video. This is not default, as default there is a constant videofeed. In case of failover, the failover server puts a new configuration on the camera, telling it where to send the event.

- The amount of lost video is around 1 minute, but maybe it can be brought down to 15 seconds.
- If cameraA fails, which is connected to serverA then how serverA can react or vice versa
  If a camera on a server fails, an event is raised and the camera is marked as offline. We try to reconnect every 10-15 seconds.

2. How can you recover the failed component data and how much time to recover the failed server position or data.
   If the data producer fails, it is not possible to get the data (as it has not been produced). If it is the sink (aka server) that fails, then it is possible to get the data from the downtime period, if the source supports edge storage. Edge data is retrieved "slowly", so we do not overwhelm the camera. It takes about n seconds to retrieve n seconds of video from the edge storage.

3. For instance, if serveA takes 10 seconds to recover by redundancy server then where 10 seconds video go, if you have a setup for redundancy facility.
   It is lost - unless the camera supports edge storage.

4. As you said, the management(mngt) server can manage the video servers, which has manual administration. In this case, how the mngt server manage long lasting connections, if video server is full or connection is failed

or etc
Basically, the Recording servers connect to the management servers and keep the connection alive. The management server receives different data from the recorder and can check connectivity by seeing whether this stream dies. It then waits for the recorder to reconnect. Otherwise it does nothing.

5. Is there any possibility to scale incoming video data with current setup at Milestone systems.
   Please elaborate on this - but my guess is no. However, you may be able to create some rules to handle this.

6. Any video compression, replication, load-balancing algorithms, internal structure of components, etc...

   - We store the video in the format we received it from the camera. This format is often configurable. When the video is read by another consumer, it can be transcoded. We do not have any replication or automatic load-balancing.
   - We can replicate the management server by using hardware replication or SQL replication. Also goes for event server. The recorders use Milestone failover.

7. Load scheduling mostly between nodes and BCs inside a DC but outside a DC could be high cost

Contents

Chapter 1

Introduction

1.1 Setting

The work that is presented in this text revolves around the concept of Managed Video as a Service (MVaas). MVaaS refers to an emerging hosting infrastructure on the cloud to store surveillance videos and to enable video monitoring services through Web. Surveillance videos can be used to prevent theft in both private homes and large crowded areas such as railway stations and airports. Managing surveillance videos is handled by service providers. They collect the video data which might come form private, public of company owned cameras. The service providers store and secure the data and make it accessible the legitimate users, which might be security companies, public institutions (police, fire departments), or private persons.

Providing a video service has been a topic of computer science since last decade, but has recently found its way to mainstream. In order to provide a video service and storage of surveillance videos, a next-generation software framework is needed to create and launch since the following reasons

- Rapid growing availability of surveillance devices, for instance, at least one video device per home.

- Concerns over security, crime and terrorism.
- Periodic growing of video quality.
- Increasing network bandwidth.

To give an idea of the demand, let us consider Denmark. To have 1 million video devices attached to the internet is not unrealistic. As per MS *storage calculator*[1] standards, one-hour uncompressed or low-compressed video file needs approximately 1.5GB storage capacity and 2MB/sec. bandwidth. It means 24 hours of video file needs 36GB storage capacity and for 30 days it needs 1080GB storage capacity by having the same 2MB/sec. bandwidth. In the case of Denmark, 1 million video files by having the same 2MB/sec. bandwidth require approximately 1030PB storage capacity for 30 days. The bandwidth and storage capacity can vary based on video quality.

The framework is based on the market knowledge and requirements of security companies like Securitas[2] AB, and the global security market presence of Milestone Systems[3] A/S (hereafter referred as to MS) which is an open platform company in IP video management software. To achieve the framework development, leading experts can help to solve the technical challenge of efficient storage, video surveillance, and examinations of unprecedented amounts of videos. The leading edge research from Technical University of Denmark (DTU)[4] and Aalborg University (AAU)[5], and on the unique technology from Nabto[6] A/S where MS is a service provider of surveillance videos.

The framework gives the solution to help reduce burglaries and increase the rate at which home robberies are solved. At the same time, the solution will result in resource savings in the security industry because it will reduce responses to many false alarms where Securitas could quickly get an overview of whether there is a real alarm or not.

In order to solve technical challenges of a next-generation scalable video surveillance framework, the leading experts systematically tackle the key technical challenges in the following six work packages (WP):

WP 1 Algorithms for Metadata: To store, search, and query large-scale meta-

[1]http://www.milestonesys.com/Support/Presales-Support/Storage-Calculator/: accessed July 2014

[2]http://www.securitas.com/

[3]http://www.milestonesys.com/

[4]http://www.securitas.com/

[5]http://www.aau.dk/

[6]http://nabto.com/

data that is automatically generated, DTU develops new scalable algorithmic techniques.

WP 2 Automatic Video Annotation: To annotate massive amounts of video data, AAU develops new unsupervised computer vision techniques for detecting abnormal video.

WP 3 Security: To protect video data against unauthorized access, DTU develops a new security model that extends existing security models with contextual information about accessing agents and environmental parameters for video.

WP 4 Architecture: To administer the storage of a large number of video streams in a distributed database, DTU develops new techniques for scheduling and virtualizing.

WP 5 and **WP 6** Integration: Deal with integrating the results of WP 1-4 into a product and with launching it by MS and Nabto.

1.2 Reason

The tremendous progress in the video surveillance systems poses new challenges on the communication and storage infrastructure. In contrast to read-heavy sites like YouTube[7], the challenge is posed by the fact that there are produced a very high amount of continuous video data which has to be stored while there are only very few requests for watching it. Compared with other file types such as Web pages, text files, etc., the video files have some particular characteristics [1] as follows:

1. The size is very large and multi gigabyte files are very common.
2. Video files are continuously streamed for a long period of time.
3. A specified termination time of video file is not known, this required storage for a given stream cannot be estimated beforehand.
4. A large high-quality video file needs large-storage space and bandwidth.

The video file characteristics impose a number of requirements on the framework to be designed. Further requirements come from users. Currently, MS uses a manually configure system where the video camera directly configures with a

[7]https://www.youtube.com

Recording server which is the actual storage facility, and this Recording server can be manageable by a federated management system. There is no possibility to have a scheduling and load balancing and as of our knowledge, MS is not using cluster of servers to manage huge amounts of data. In the case of failure of Recording server, MS simply replaces with a new Recording server and it takes lot of time to configure with the new Recording server, so there is no guarantee on data of failure server and no automatic re-assignment. On a high level, the framework has to meet the following general requirements or features:

- Ability to handle millions of video streams continuously.
- Give support to privacy and allow locality.
- Flexibility to handle rapidly growing storage demands.
- Allow service providers to add or modify features.
- Scalability to meet increasing demands.
- Reliability to avoid failures of an infrastructure.
- Load-balancing, redundancy, fault tolerance and recovery techniques to achieve high-availability and high-throughput.

With these general requirements, a completely new system or framework, called MVaaS, has to be introduced. The motivation to develop MVaaS is that existing systems support a large number of reads as compared to the writes and do not support massive continuous video writes. Furthermore, MVaaS has to have an efficient resource allocation algorithm to optimize the route between a video producer and storage system where the particular video stream is most optimally stored. The MVaaS architecture must support dynamic reconfiguration to best balance the loads on different components (see in Section 3.3) and best utilize the available resources (see in Section 3.6).

The development of MVaaS architecture by *DTU* is based on the following hypotheses:

Hypothesis 1 : A scheduling algorithm that efficiently distributes incoming data streams to a distributed storage facility and that scales to number of incoming streams and any size of the distributed storage.

Hypothesis 2 : A general system architecture that supports the decoupling of video devices (cameras) and storage facilities which receive videos from video devices in a general MVaaS architecture that meets the high-bandwidth requirements for both communication and storage in a MVaaS system.

The key idea is that a high degree of decoupling between the different system components supports dynamic reconfiguration and facilitates the exploitation of emerging technologies, such as cloud storage technology.

1.3 Requirements and choice of variables

The general requirements to be listed in section **??**, which have to be met by the MVaaS give rise to more specific requirements which we describe now. The specific requirements with possible setting variables or parameters allow the service providers to give the best service to users in all possible ways and to test the MVaaS performance in storage. The motivation to stated only performance in storage is MVaaS restricts at the storage-level end. It means MVaaS can get very few retrievals as compared to storage. The specific requirements are:

RX

Name : Name of requirement

Condition : Here is stated what has to happen, such that the requirement is met.

Verification : Here is stated, how one checks that the condition is fulfilled.

Setting of parameters or variables : If the condition or the verification contains numerical parameters, then possible settings state here.

In general, the specific requirement RX states that the standard description/format of following specific requirements.

R1

Name : Reliability.

Condition : At most $X\%$ of the data produced by producers[8] is loss i.e. not reaching the consumers [9] or being properly recorded

[8]A producer is a video device which produces video file.

[9]A consumer is a storage facility which receives video file from producer

Verification : Running the system for t hours and monitoring the lost data.

Setting of variables : $X = 0.01$ and $t = 200$.

R2

Name : Redundancy at multiple Levels

Condition : No data loss if k servers fail from n servers.

Verification : Turn k servers off out of n servers and check that all data is still available.

Setting of variables : $k = 2$ and $n = 100$.

R3

Name : Low latency (on average)

Condition : $X\%$ of data transfer to a server performs with at most t_s seconds delay.

Verification : Pass the data to an immediate neighbor server for t_h hours and measure the maximum delay.

Setting of variables : $X = 50, t_h = 24$ and $t_s = 0.1$.

R4

Name : Failure of producer

Condition : When a producer "often fails to send", this has to be detected within t seconds

Verification : Make one producer drop the data more often and check that is detected.

Setting of variables : $t = 1$.

R5

Name : Logs

Condition 1 : All actions[10] have to be logged within t_s seconds.

[10]Actions are high level, e.g.: change of meta data and video server but simply sending a single package will not be considered an action.

Verification : For a period of t_h hours check that there is log with all actions and that is updated within the requested time.

Setting of variables : $t_s = 50$ and $t_h = 24$.

R6

Name : Scalability

Condition : The system must be able to handle any number of producers and consumers [or up to X producers and up to Y receivers].

Verification : Implement the system on different architectures and measure the performance.

Possible : Milestone will define three sets of parameters.

R7

Name : Response

Condition 1 : If a user requests a video, then the server has to supply it within t_s seconds.

Verification : Generate the user requests at a t_x seconds rate and check that the response happens within t_s seconds.

Setting of variables : $t_s = 2$ and $t_x = 0.1$.

Condition 2 : The video transmission to the user has to be stable.

Verification : No more than X interrupts per minute and no interrupt longer than t seconds

Setting of variables : $X = 2$ and $t = 2$.

Condition 3 : If n users make a request at some time, then maximum response time has to be within t seconds for all requests.

Verification : Generate the users request and check that all are answered in the given time.

Setting of variables : $n = 20$ to 2000 (MS will decide), and $t = 1$.

R8

Name : Bandwidth (BW) consumption

Condition : If no motion then internet link uses lowest possible bandwidth that is needed for keys.

Verification : Set some producers to ‘no motion’ then check the BW X and set some producers to ‘motion’ then check the BW Y.

Setting of variables : $X = 100KB$ (low) and $Y = 1 \ldots 100MB$.

R9

Name : Load balance

Condition : An active servers (servers which are receiving the data from producers) are balanced within t minutes.

Verification : Take average current load of any k servers from n servers within t time and set every server load is less than the average load of k servers.

Setting of variables : $t = 60, n = 100$, and $k = 1, 2, ..10$.

R10

Name : Manage

Condition : If the load (or bandwidth) demand exceeds the capacity, then the system has to manage writes (either prioritize or redistribute)

Verification : Set $X\%$ extra load on current maximum ($Y\%$) load and check that $X\%$ load is loaded.

Setting of variables : $X = 10\%$ and $Y = 90\%$.

R11

Name : Fault-tolerance

Condition1 : If a group of k servers fail then the system maintains same performance.

Verification : Switch off k servers and check the system still works (within t seconds)

Setting of variables : $t = 5$ and $k = 10$.

Condition2 : If k servers fail in a cluster[11], then the system is stable to write and read operations.

Verification : Make k servers fail and check the system is still working in within t seconds.

Setting of variables : $t = 1$ and $k = 2$.

Condition3 : If p producers fail, then the system has to be notified.

Verification : Switch off p producers and check that there is a notification within t seconds.

Setting of variables : $t = 1$ and $p = 10$.

Condition4 : If bandwidth fails at some point, then the system receives the data afterwards.

Verification : Cut off the bandwidth and notify within t seconds. Once reconnects it, check the old X GB data is still available.

Setting of variables : $t = 1$ and $X <= 1GB$.

R12

Name : Central access.

Condition : If Multi national companies (MNCs) want to view all their companies surveillance video data then the system supports all types of requests

Verification : Set X clusters in different areas, and check $Z\%$ metadata is transmitted or not.

Setting of variables : $X = 5$ and $Z = 100\%$.

R13

Name : Localization of data (local clouds).

Condition : Ability to isolate data collected at some specific locations form the rest of the system, and supply specific access rules.

Reason : Some companies or public institutions do not want that video which has been collected at their facilities to be anyplace, but on a *private cloud*[12]. For MNCs, this private cloud can become quite large.

[11]A cluster is a group of servers connect each other through LAN

[12]A private cloud is a model of cloud that involves a distinct and secure cloud based environment in which only the specified user can operate

Verification : Set X clusters for one area, and check $Z\%$ data of X clusters is available or not.

Setting of variables : $X = 3$ and $Z = 100\%$.

R14

Name : Continuous streaming.

Condition : Handle continuous video streams to video storage for unbounded time.

Verification : Set X video devices, and run the system for t_d days to check the video handled or not.

Setting of variables : $X = 10$ and $t_d = 10$.

R15

Name : Large files storage.

Condition : Handle continuous video storage without any file size bounds.

Verification : Set X video files from different streams, run the system for t_d days to store Z PB data without any interruption.

Setting of variables : $X = 10$, $t_d = 10$, and $Z = 20$.

R16

Name : Handle small files.

Condition : Handle small files which are less than 1MB size.

Verification : Set X files, run the system for t_s seconds to check the file is handled or not.

Setting of variables : $X = 2$, and $t_s = 5$.

Chapter 2

Existing Systems

2.1 Overview

Existing infrastructures are based on the traditional client-server model [2, 3, 4] where a server is a program that provides services on behalf of its clients of computer networks, instead of the peer-to-peer model [5, 6, 7] where the software running at each server is equivalent in functionality without any centralized control or hierarchical organization. That is, users connect to a hosting server to which they upload, and from which they download files. Compared to peer-to-peer networks, central file hosting is more demanding in terms of the hosting server-side infrastructure required, and usually delivers faster and more reliable transfers [8]. In the hosting server-side infrastructure, a single server is unable to handle the streaming video-storage because it is unable to meet the demand of storage requirements and user requests.

Distributed systems offer more advantages in the form of availability and durability than centralized strategies[9]. Such systems like Google File System (GFS) [10], Amazon Dynamo [11], Cassandra [12], Hbase [13], OceanStore [14], Coda [15], and Frangipani [16] utilize storage utility inheritance and redundancy for highly available and easily accessible file system interface. On such systems storage, bandwidth and computational power are adjusted dynamically [17]. Furthermore, existing well noted video streaming systems use the distributed storage system, but they allow or support only limited length videos. For instance, YouTube[1] gets 100 hours of video every minute and Netflix[2] follows Amazon S3[3] storage infrastructure which supports upload file sizes from 1 byte to 5 terabytes while we expect larger files than 5 terabytes. The following section describes well noted existing distributed storage systems and explains why MVaaS system is not simply use one of them for massive video storage.

2.2 Distributed Storage systems

2.2.1 Google File Systems

GFS is a scalable distributed file system that is built for hosting the state of Google[4] internal applications. The GFS consists of a single master and multiple read slaves (chunk servers) and is accessed by multiple clients. The GFS uses BigTable [18] that is built for hosting structured data and settles on a sharding approach [19]. BigTable treats data as un-interpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control locality of their data through careful choices in their schemas.

Advantages

- Generally, GFS deals with large files that can be difficult to manipulate using a traditional computer file system.
- Large files split into chunks of 64MB each.
- Easy to see which computers in the system are near capacity and, which are underused. It is also easy to port chunks from one resource to another to balance the workload across the system.
- GFS is scalable, i.e., ease of adding capacity to the system. Scalability is a top concern since GFS requires a very large network of computers to handle all of its files.
- Performance is high. The performance should not suffer from scalability.

[1]http://www.youtube.com/yt/press/statistics.html : accessed June 2014
[2]http://www.netflix.com/
[3]http://aws.amazon.com/s3/: accessed June 2014
[4]https://www.google.com/

- Autonomic computing. GFS able to diagnose problems and solve them in real time without need for human intervention.
- Design simplicity. A simple approach is easier to control, even when the scale of the system is the use.
- Familiar interface has basic file commands like open, create, read, write and close and also have specialized commands: append and snapshot.
- Reliability through replication, i.e, each chunk replicated across 3+ chunk servers.
- No data caching so system gets the benefit due to large data sets and streaming reads.

Disadvantages

- There are a large number of components, *they often fail*[5].
- Each chunk has a unique 64KB identification number called chunk handler. Even adding a small file like an email, GFS generates the 64KB size identification number which is heavy to the system.
- Centralized architecture loads heavy burden on the master server. Having a single master causes the scalability bottleneck, however, by applying interaction strategies, the scalability was achieved to remove the bottleneck to a large extent [10].
- There is no localization of storage since all Google applications data can store in Google's data centers which are at very few locations (United States of America, Singapore, Ireland, etc) but not the location of the data where it belongs.
- GFS goes all the way down to much smaller file sizes. However, "1 MB seems a reasonable compromise in GFS environment" [6]
- Failover and recovery takes about 10 seconds now. This could be acceptable in batch situation but not OK from a latency point of view for a user facing application which described in the case *study of GFS*[7] [8]

Requirements to meet (+) or fail (-) MVaaS

[5]Referred from web community: accessed June 2014

[6]http://queue.acm.org/detail.cfm?id=1594206 : accessed July 2014

[7]A discussion between Kirk McKusick and Sean Quinlan about the origin and evolution of the GFS

[8]http://www.theregister.co.uk/

- GFS supports large streaming reads in contrast to writes of MVaaS requirement R14.

+ The main features of distributed systems include scalability, reliability, high-throughput are familiar to MVaaS requirements

- MVaaS needs to keep support on isolated data (requirement R13) so single controller does not suitable for it.

Conclusion

MVaas expects to support large streaming writes, localization of data and individual control of local data or one area data. Therefore, GFS can't support MVaaS requirements.

2.2.2 Amazon Dynamo

Amazon has developed a highly available key-value based storage system called Dynamo [11] in a cluster environment for its internal use. Dynamo is a decentralized distributed storage system. In Dynamo, storage nodes or servers are organized on a ring-based distributed hash table (DHT)[20] and each data item is asynchronously replicated on the successor storage nodes.

Advantages

- Handle software and hardware failures. It is done automatically and transparently migrating data to new hardware as hardware fails or reaches its end of life [21].
- Simple interface has write, read, and delete objects containing from 1 byte to 5TB of data each. The number of objects are unlimited.
- Each object is stored in a bucket and retrieved via a unique, developer-assigned key.
- Objects stored in a region never leave the region unless transfer them out. For example, objects stored in the Europe Region never leave the Europe.
- Authentication mechanisms are provided to ensure that data is kept secure from unauthorized access. Objects can be made private or public, and rights can be granted to specific users.
- Common features of scalability, reliability, flexible, and high-throughput.
- Dynamo has built-in fault tolerance, automatically and synchronously replicating data across multiple availability zones in a region for high

availability and to help protect data against individual machine, or even facility failures[9].

- Tightly integrated with Elastic Map Reduce (EMR) that helps to perform complex analytics of large datasets.

Disadvantages

- Dynamo is an expensive and extremely low latency solution, If you are trying to store more than 64KB per item.
- It can't support larger than 5TB file.
- No information on continuous streamed million writes.
- Support partial isolation of the data but not full fledged.
- If asynchronous writes are rejected, they may simply translate into a retry loop. This solution is not good if we expect to be able to read the value that we have written immediately after the write. This is not guaranteed by Dynamo.
- Since availability zones are not geographically dispersed, it needs cross-region replication which is very time consuming and introduces large delays [22].

Requirements to meet (+) or fail (-) MVaaS

- Dynamo meets maximum requirements of MVaaS, however, it can't support larger continuous streams (requirement R14) and fully isolated data whenever user needed (requirement R13).

+ The term data center is familiar to build a datacenter for isolated data in MVaaS.

+ Formation of ring of servers idea is also close to MVaaS internal structures of storage facilities to exchange data for load balancing.

Conclusion

Some of MVaaS requirements are satisfied by Dynamo, however, MVaaS has more additional essential requirements like continuous streaming and larger files than 5TB which are not satisfied by Dynamo. Thus, Dynamo architecture does not fit for MVaaS requirements.

[9]http://www.slideshare.net/saniyakhalsa/dynamo-db-pros-and-cons

2.2.3 Cassandra

Cassandra [12] follows decentralized architecture to store very large amounts of structured data in distributed data centers. A data center is a grouping of nodes or servers configured together for replication purposes. Cassandra provides highly available services with no single point of failure and handles high write throughput while not sacrificing read efficiency. Cassandra has a column oriented model. Core design principals are inspired by BigTable data model for structured data, and high availability key-value store from Dynamo.

Advantages

- Cassandra can replicate data easily and safely. If one of the replication node goes down, it can replicate data after it goes up.
- The data replication protects against hardware failure and other problems that cause data loss in a single data center.
- Failure detection is completely controlled locally.
- There are commons features of the distributed file system which ensure high-availability, fault-tolerance, scalability, etc.
- Ability to access and deliver data in near real-time performance to interactive, Web-based applications at different scales.
- Ability to deploy across geographically dispersed data centers.
- Cassandra uses elastic scalability so it can be easily scaled-up and scaled-down.
- It supports MapReduce model that reads and writes data column family.

Disadvantages

- Consistency model wasn't a good enough for messages.
- There is no proof for continuous video streams write and how much size it can handle.
- Maintenance of localization is very hard in the Cassandra decentralized model.
- A single column value must not be larger than 2GB so there is no streaming or random access.
- The maximum number of cells (rows x columns) in a single partition is 2 billion.

Requirements to meet (+) or fail (-) MVaaS

+ Grouping servers scenario is familiar to MVaaS internal structures of storage facility.
- MVaaS streaming requirement R14 can't support in Cassandra.
- Cassnadra data model makes it well suited for write-heavy applications that can take advantage of the fact that rows are not required to have the same fixed number of columns for every row in a column family[10], however, the column value is limited so it's against to MVaaS requirement R6.

Conclusion
Cassandra data center model is partly suited for MVaaS storage facility, however, it cannot support all requirements of MVaaS.

2.2.4 Hadoop Distributed File System

HDFS is a distributed file system that is well suited for storing large files. HDFS is highly fault-tolerant and provides high-throughput access to application data that have large data sets. In a large cluster, thousands of servers both host directly attached storage and execute user application tasks. By distributing storage and computation across many servers, the resource can grow with demand while remaining economical at every size [23].

Advantages

- HDFS allows for many attributes (column represents an attribute of an object) to be grouped together into what are known as column families, such that the elements of a column family are all stored together[11].
- Provides low latency access to small amounts of data from within a large data set.
- Provides the flexible data model, reliably, high-availability, fault-tolerance, scalability, etc.
- Data can partition across many (thousands) of hosts and the execution of application computations in parallel close to their data.

[10]http://blog.credera.com/technology-insights/java/cassandra-explained-5-minutes-less/
[11]http://aosabook.org/en/hdfs.html

- It has a very simple architecture (master-slave).
- HDFS works best when manipulating large files (GBs and PBs).
- Support partition/block structured file system so each file partition into a fixed size and are stored across a cluster.

Disadvantages

- HBase is not optimized for classic transactional applications.
- HDFS has storage or network level encryption which is a big concern in security.
- Inefficieny in handling of small files, and it lacks transparent compression. "As HDFS is not designed to work well with random reads over small files due to its optimization for sustained throughput"[12].
- The global synchronization or sharing of mutable data is not good for system high-throughput.

Requirements to meet (+) or fail (-) MVaaS

- Single Master controls the system which contrasts to MVaaS locality requirement R13.

- No large file write information of requirement R15, however, it reads very large files until PB size.

+ Partition of large file and have common features of the distributed file system.

- Unable to handle small files where MVaaS handles less than 1MB size file (requirement R16).

Conclusion
Most of the MVaaS requirements are not satisfied with HDFS features except the common features of the distributed file system.

2.2.5 OceanStore

Ocean Store [14] is a globally scalable storage utility, allow users to utilize a distributed storage service. OcenStore aim is to provide a storage utility inherently means that data must be highly available, secure, easily accessible file system interface, and support guarantees on Quality of Service(QoS).

[12]Web community

Advantages

- OceanStore gives consistency guarantee.
- It has scalability, and availability through replication features.

Disadvantages

- Performance of OcenaStore is unproven for large-scale storage systems, however it has provided little evidence that the system can perform efficiently.
- It is unclear whether the added load on the network caused by the routing and redundancy will overshadow the benefits of local data.
- Security issues are remained. Once key is compromised it does not explain what will happen.
- Globally accessible, but not clear about local control.
- Finally, this model apparently is best for distributing static, not dynamically generated data, and data consistency presents problems.

Requirements to meet (+) or fail (-) MVaaS

- It does not support large file (requirement R14) and no proof on continuous streamed data maintenance either read or write (requirement R15).
- No localization mechanism available which is against to requirement R13.
- No control on data since it follows locally centralized peer-to-peer model which is almost opposite direction to MVaaS requirements R1, R12, and R13 .

Conclusion
OceanStore cannot fit into MVaaS requirements at any of its specific requirements like continuous streaming, large file storage, control of local data and localization.

2.2.6 Frangipani

Frangipani is a peer-to-peer distributed storage system. It offers users excellent performance as it stripes data between servers, increasing performance along with the number of active servers. It provides a file system like interface that is completely transparent to users and applications. Frangipani is designed to operate and scale within an institution and thus machines are assumed to be interconnected by a secure, high-bandwidth network under a common administrative domain [16].

Advantages

- Frangipani can provide redundancy.
- Common features of distributed storage system which ensure scalability, availability and high-throughput.

Disadvantages

- Scalability is only for medium size distributed systems.
- No proofs to handle large files, localization, security, continuous streamed data read or write.
- The performance is dropped when a server writing a file while other servers read the file.

Requirements to meet (+) or fail (-) MVaaS

- Frangipani cannot meet most of the MVaaS requirements R7, R10, R11, R12, R13, R14, and R15.

Conclusion

MVaaS expects large-scale storage-system which has to have specific requirements which are not satisfied by Frangipani.

Below table explains about well known distributed storage systems features in concern of MVaaS requirements.

Distributed storage	Master level	Data model	Interface	Continuous writes	Localization
GFS	master			limited	no
Dynamo	de-central	key value	API	limited	limited
Cassandra	de-central	key value	API	limited	no
HDFS	master	key value	API	limited	no
OceanStore	peer-to-peer	key value		very limited	no
Frangipani	peer-to-peer	key value		very limited	no
MVaaS	de-central	key value	API	unlimited	fully

Table 2.1: An overview of the existing system features in the concern of MVaaS

As of chapter 2, most of the existing systems are fast in reads rather than writes, but no system is designed to maintain continuous video streams, for the reference see in Table 2.1, specifically the existence of large-scale video-storage domains. Therefore, the MVaaS needs a specific system which supports all the requirements 1.3.

CHAPTER 3

MVaaS Architecture

3.1 Overview

The motivation to build a new system is to fulfill all the requirements 1.3 of MVaaS, specifically, the MVaaS system

- continuously handles millions of video streams, for instance, as a rough estimation, the system must support a service to 10^6 continuous writes at any time
- supports localization since the access to surveillance videos has to be limited for legal reasons. Furthermore, a company might want to store all video information on local servers to avoid industry espionage.

In addition, the MVaaS system has a possibility to have existing storage systems 2 features concern to video storage, for instance,

- supports reading video, might be parallel with storing the video (live-streaming while storing),

- constantly monitors itself and detects, tolerate and recover promptly from software failures on a routine basis and also manages hardware failures [10],
- efficiently implements well-defined splitting and append mechanisms of large-video files for writes and reads [10].

Furthermore, the MVaaS system has a familiar file system interface, i.e., files are organized hierarchically in directories and identified by path names [10]. It supports the usual operations to create, delete, open, close, read, re-read, and write files. Simple read and write operations to a video data is uniquely identified by a key-values. The key-value stores are easier to configure on a per-application basis [11].

For the localization purposes, the MVaaS system storage infrastructures can be geographically distributed and easily aggregate each other through a possible method hierarchy. The hierarchy can aggregate multiple storage infrastructures, and present them to a user under a unified name space.

The motivation to select a hierarchical storage system is to get the substantial benefits in several ways:

- Localization of data by using a hierarchical level's storage strategy (Figure 3.1). These levels keep local data on high speed at storage system.
- Data activity can monitor by an immediate higher-level management system.
- Possibility to keep archival data in lower-level in the hierarchy.
- Maintaining data on high performance storage clusters and configure reliable redundancy.
- Possibility to expand storage capacity on different levels in the hierarchy.
- Quick and reliable access the data which is localized.
- Low-cost to accommodate a large-scale video storage [24].

3.2 Architecture

The MVaaS system is a hierarchical architecture. Its main component is a *datacenter*, DC for short. DCs are entities, which can autonomously provide all the services needed for a MVaaS system. At the same time, DCs can be embedded into each other to form a scalable, hierarchical system. The embedding structure is a rooted tree, for the reference see in Figure 3.1.

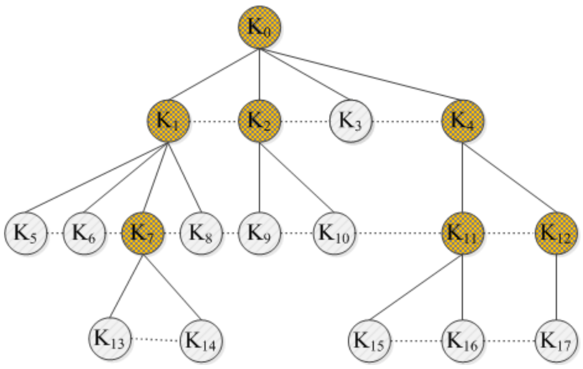

Figure 3.1: Architecture of a hierarchical datacenter; where dark-shaded circles are DCs (consumers), and light-shaded circles are basic consumers. The hierarchical heavy-lines are direct links, and parallel dotted-lines are links, which can be established by immediate-higher level DCs.

MVaaS system states a DC $\mathcal{K}$ is *above* (or *higher-level* to) DC $\mathcal{K}'$ if it is closer to the root of the tree than $\mathcal{K}'$. The terms *below* (or *lower-level*) are defined symmetrically. The DC $\mathcal{K}$ is *immediately above* DC $\mathcal{K}'$ if $\mathcal{K}'$ is a child to $\mathcal{K}$ in the tree, again *immediately below* being defined symmetrically. The DCs $\mathcal{K}$ and DC $\mathcal{K}'$ are *at the same level* if they are same distance from the root. For instance, in Figure 3.1, DC K_1 is immediately above DC K_7, immediately below DC K_0 and same level to DC K_2 and K_4.

The motivation to have a single top-most DC in the hierarchy is that the top-most DC vastly simplifies MVaaS design and enables to make sophisticated data placement decisions using its knowledge about immediate lower-level DCs. However, MVaaS system must minimize the involvement of the top-most DC in storage and control operations. Therefore, having a single top-most DC does not become a bottleneck [25] in a hierarchical distributed control and storage system. In the hierarchy, there is no single DC which needs to have a global knowledge [26], i.e., whole system-level knowledge.

In order to meet the localization requirement R13, horizontal links (links between DCs at the same level) are not allowed by default. However, a DC can allow a direct connection between DCs which are immediately below. Such links cannot be established directly by two DCs at the same level, but require the involvement of the DC immediately above, which acts as a security manager. The security manager is a DC which allows communication between the same level of immediately below DCs for performing operations.

3.3 Components

Inside a DC has the following components, for the reference see in Figure 3.2. MVaaS system uses the term “inside a DC” to denote the fact that the component is directly embedded in the DC, that is, not in a below DC.

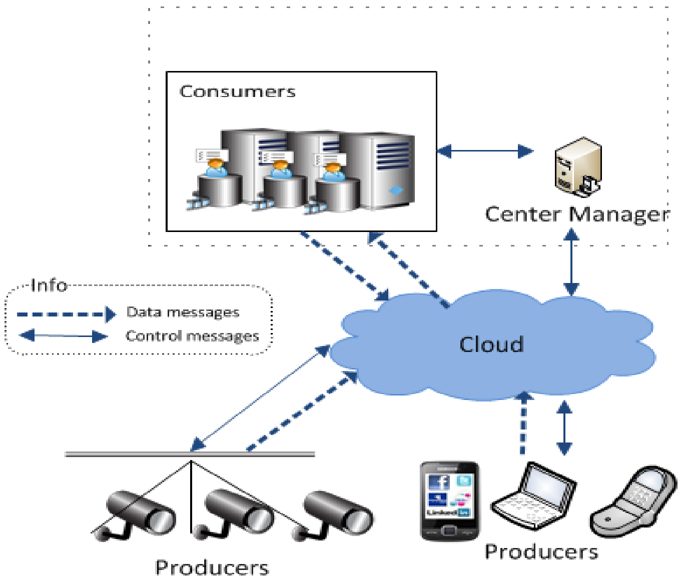

Figure 3.2: Architecture of a datacenter

Producers are devices which produce one ore more video streams.

Consumers are devices which absorb one ore more video streams.

Center Manager is the responsible for monitoring the status of the DC and assigning the producers to consumers via selected links.

Links are the (network) connections between the components

It should be emphasized, that producers and consumers are not fixed physical devices but *rolls*. For example a hard disk can act as a consumer when storing data, and as a producer when transferring data to another device. A human user will act as a consumer when viewing a video. Embedded DCs will also act as consumers or producers for the DC immediately above. Details are given in the next sections.

3.3.1 Producers

Devices which can generate a data stream, act as producers when doing so. Note, that some devices can act as producers at some time and as consumers at a different time. This is especially true for storage devices, which usually consume data. However, when a storage device transfers data to another device, a background storage for example, it becomes a producer.

3.3.2 Consumers and Basic Consumers

Devices which can absorb a data stream, act as consumers when doing so. Typically, all storage devices act as consumers. In the hierarchical system, an embedded DC can act as a consumer. In order to distinguish between virtual consumers like DCs and "real", physical consumers like disks, we introduce the concept of a basic consumer. A *basic consumer* offers all functionalities of a consumer to the outside, but it is not a DC. Especially, it does not have CM and cannot schedule storage requests "further down". Examples are memory cards, disks, and clusters of disk. They are at the bottom of the storage hierarchy.

3.3.3 Links

We assume that all consumers and producers inside the DC can communicate through links with each other. There are links form CM up and down to DCs.

3.4 Center Manager

Every DC has it own *center manager*, CM for short. The tasks of the CM are

- To monitor the state of the DC. It means status of all consumers inside DC include load of each consumer and their health (working condition).
- To receive storage requests. Such a request usually comes from a producer inside the DC. It might, however, also come from the (CM of the) DC immediately above.
- To receive read requests form consumers.
- To handle the incoming requests by assigning a producer to a consumer and assigning an appropriate link between those two.
- To load-balance the components in the DC.
- To replicate itself. The replicated CM takes over the role of the primary CM in case the latter fails.
- To give access to migrate data between consumers inside a DC.

It is important to note that the actual video stream is not passing through the CM. The CM assigns the connection between a producer and a consumer.

3.4.1 Information Available to and Communication possibilities of the CM

- A CM knows its place in the hierarchy, i.e., knows the the complete hierarchy tree below it and, in addition, the DCs immediately above.
- A CM can communicate to all CMs of DCs immediately below and the CM of the the DC immediately above.
- A CM knows all consumers and producers inside a DC.
- A CM can communicate with all producers inside a DC.
- A CM can communicate with all consumers inside a DC. In case of a consumer is a DC this happens via the CM, in case of a basic consumer, it communicates to the interface (see Section 3.5 below) of this consumer.

At any point in time, every component (consumer, producer, link) belongs to a unique DC. It is therefore also associated to a unique CM. It is this center manager, to which the component sends request, error messages and from which the component receives the information to which other components it is associated. Also every user is associated to a unique DC (and CM) at any point of time. This assignment can, however, change with time. A human user, a guard, associated to $\mathcal{K}$ might change position and move to the area covered by another DC $\mathcal{K}'$. Then the lowest DC $\mathcal{K}^*$ has to be informed and will assign the user to $\mathcal{K}'$.

3.5 Basic Consumers

A basic consumer can be a single-device or single-server (also called node) or a cluster of nodes with a fixed and reliable internal connection's infrastructure such as a wired LAN. Every basic consumer has a *communication interface.* This allows the CM of the DC into which a consumer is embedded to communicate with the basic consumer. This communication includes:

- Querying the status of the basic consumer (health, free space, computation power, bandwidth, etc.).
- Passing a storage request to the basic consumer to storage.
- Passing a migrate request to the basic consumers for load-balancing.

Basic consumers are considered and organised into two structures namely unstructured and structured.

3.5.1 Unstructured, Single Devices

The single devices can be memory cards, single disk, RAID servers, etc. Most of the unstructured, single devices cannot give high-throughput since single devices unable to meet the requirements of the MVaaS unlike continuous video streaming R14 and load-balancing R9. However, these single devices make creating and managing an archive significantly easier.

3.5.2 Structured, Organized Multi-Devices

The organized multi-devices are mostly clusters which are a group of loosely/tightly coupled servers or nodes. The nodes communicate with each other through links define the *interconnection network* or *topology.* The following physical/logical topologies are described on how nodes can communicate and share data or messages [27].

3.5.2.1 Ring

Physical servers, referred to in the cluster as nodes can form in a Ring topology for faster communication. In local-area networks where the Ring topology is used, each node is connected to its two neighbours for communication purposes, for the reference see in Figure 3.3. The communication passes through each node connected to the Ring in one direction (either clockwise or counterclockwise), from node to node, until it reaches its destination. Each node is fully capable of serving any storage or retrieval requests. The Ring topology can serve efficiently in data partition, redundancy and in fail-over. Rather than using a legacy master-slave, well known existing systems use a peer-to-peer distributed Ring topology that is much more elegant, easy to set up, and maintain [11, 12]. One

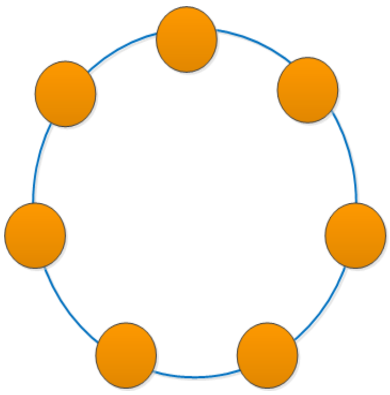

Figure 3.3: An example of Ring topology

of the key design requirements for Cassandra and Dynamo is the ability to scale incrementally by dynamically partitioning the data over the set of nodes in a Ring. The Ring uses consistent hashing [28]. In consistent hashing the output range of a hash function is treated as a fixed circular space or Ring [11, 12]. Each node is same in the Ring, and all nodes communicate with each other with different protocols, for instance, Gossip protocol at Cassandra. The communication process runs regularly, and exchanges status messages with up to three other nodes within the Ring of Cassandra's data center. In Dynamo, each node is assigned a random value which represents its position on the Ring. Each node is responsible for the region in the Ring between it and its predecessor node on the Ring. The principal advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors, and other nodes remain unaffected [11].

One of the challenges in the Ring is heterogeneity in the performance of nodes.

However, by analyzing load information of each node on the Ring can be resolved the challenge, where lightly loaded nodes move on the Ring to alleviate heavily loaded nodes as described in [5].

Using Ring topology has the following advantages:

- If a node becomes unavailable due to link failures then the load handled by the node is evenly dispersed across the remaining available nodes [12].
- When a node becomes available again, or a new node is added to the ring, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes [12].
- It does not require a network server to control the connectivity between workstations. It performs better under a heavy work load.
- It is relatively easy to install and reconfigure.
- Easy to identify the problem if the entire network shuts down.

Disadvantages of Ring topology

- Only one node can transmit on the network at a time.
- A failure of one node will cause the entire network to fail. The failure in any cable or device or node breaks the loop and can take down the entire network.
- Ring networks distribute traffic fairly, but not efficiently. A lack of links between nodes limits their ability in areas of performance and reliability. However, cross connecting multiple Rings can yield good results.

3.5.2.2 Mesh

In a Mesh topology, each node is connected to every other node in the network (see in Figure 3.4). Every node not only sends its own signals but also relays data from other nodes. The Mesh topology provides a layout that makes it much easier for constant flow of data. This type of topology is very expensive as there are many redundant connections, thus it is not mostly used in computer networks [27]. It is commonly used in wireless networks, for instance, commercially available parallel computers: T3D (Torus, 3-Dimensional), SGI (Silicon Graphics, Inc.), IBM (International Business Machines) Blue Gene. Advantages of Mesh topology

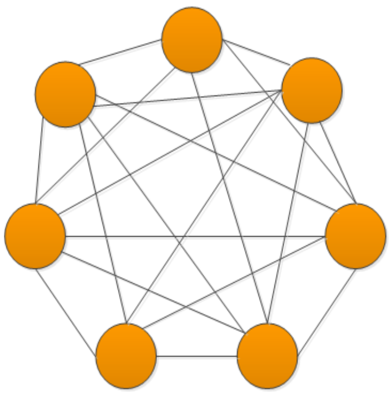

Figure 3.4: An example of Mesh topology

- Data can be transmitted from different nodes simultaneously.
- It has always an alternative to replace, if one of the components fails, so data transfer does not get affected.
- Expansion and modification in topology can be done without disrupting other nodes.
- It eliminates traffic problems in links sharing. If one link becomes unusable, it does not incapacitate the entire system. Thus, act as robust[1].
- Point-to-point link makes fault identification easy.

Disadvantages of Mesh topology

- There are high chances of redundancy in many of the network connections.
- Overall cost of this network is way too high as compared to other network topologies [29].
- Set-up and maintenance is very difficult. Even administration is also difficult.

3.5.2.3 Star

In Star topology, each node on the network to have a point to point connection to the central device called "hub", a router or a switch. The hub acts as a

[1]Web community

signal booster or junction to connect different nodes present and allows signals to travel greater distances. So every node is indirectly connected to every other node by the help of hub, for the reference see in Figure 3.5. All the data on the star topology passes through the hub before reaching the intended destination. The hub can also communicate with other hubs of different network. Some of

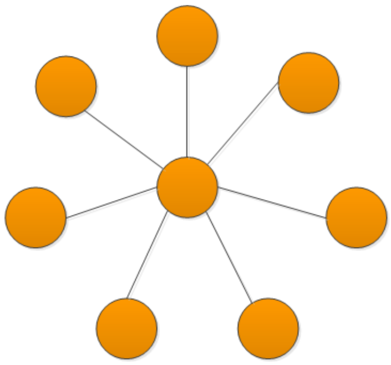

Figure 3.5: An example of Star topology

the advantages of Star topology are listed below:

- Easy to install and configure. Easy to administrate since it has centralized management to monitor network.
- The working process is simple to establish, understand and navigate. Any fault in the devices connected to the central hub can be detected easily as the malfunctioned device or link is isolated, which allows easy probing into the matter [29].
- If one link fails, only that link is affected, other links remain active.
- Easy to detect failures or replace failure nodes without affecting rest of the network.

Disadvantages of a Star topology

- If the hub fails, the whole network goes down since the hub is central device to connect all nodes, i.e., too much dependency.
- It is expensive since it requires more network cable to connect all nodes, hubs, etc.
- Performance depends on capacity of the hub.

3.5.2.4 Bus

In Bus topology, each node is connected to the single bus cable which is a backbone to link all the nodes in a network cable (see in Figure 3.6). A terminator is required at each end of the bus cable to prevent the signal from bouncing back and forth on the bus cable [29]. Advantages of Bus topology

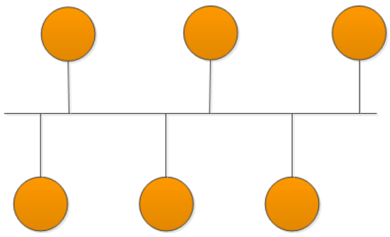
Figure 3.6: An example of Bus topology

- Easy to connect and configure.
- Requires less cable length than mesh or star topologies.
- Mostly use in small networks.

Disadvantages of Bus topology

- If the network cable breaks, the entire network will be down.
- Only one cable is used to connect all nodes, it can be the single point of failure. .
- Difficult to identify the problem if the entire network shuts down.
- Efficiency reduces, if the number of nodes connected to it increases.
- It is not suitable for networks with heavy traffic.

3.5.2.5 Tree

In Tree topology, a central 'root' node (the top-most node in a hierarchy) is connected to one or more other nodes that are one lower-level (i.e., second level) in the hierarchy with a point-to-point link between each of the second-level nodes [29]. Second level root nodes also have one more other nodes that are one level lower in the hierarchy (i.e., third level), for the reference see in Figure 3.7. The top-most root node has no other node above it in the hierarchy. Advantages of a Tree topology.

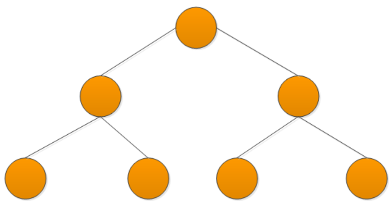

Figure 3.7: An example of Tree topology

- It is an extension of Star and Bus topologies, so in networks where these topologies cannot be implemented individually for reasons related to scalability[2].
- The whole network divided into segments (star networks), which can be easily managed and maintained.
- Error detection and correction is easy.
- If one segment is damaged, other segments are not affected.

Disadvantages of a Tree topology.

- Overall length of each segment is limited by the type of cabling used.
- More difficult to setup, configure and add or remove nodes.
- Because of its basic structure, it relies heavily on the main cable, if it breaks whole network goes down.
- Scalability of the network depends on the type of cable used.

3.5.2.6 Hypercube

All nodes can communicate with each other through an interconnected link in a Hypercube topology. The nodes of the Hypercube correspond to the individual

[2]Web community

processors, and the edges correspond to the message links between them [30]. A Hypercube of d dimensions consists of 2^d nodes, for the reference see in Figure 3.8. The nodes send messages to adjacent nodes through the links. Messages sent between non-adjacent nodes are routed through intermediate nodes until they reach their target node.

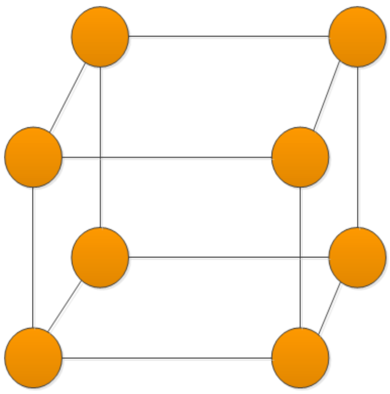

Figure 3.8: An example of 3-Dimensional Hypercube topology

Advantages

- Having only one main cable connecting all of the stations together drastically reduces the time and cost of installation and makes troubleshooting particularly easy.
- Hypercube architecture offers an unusually good combination of a high nodes connectively, software flexibility, and system reliability [31].

Disadvantages

- Poor scalability.
- Having a single cable causes single point failire.
- Administration is so difficult.
- More difficult to setup and configure.

The Table 3.1 illustrates the features of different topologies.

Topologies	Scalability	Cost	Performance	Management
Ring	high	low	high	easy
Mesh	poor	high	high	difficult
Star	poor	high	-	easy
Bus	poor	cheap	-	average
Tree	high	high	-	easy
Hypercube	poor	high	high	difficult

Table 3.1: An overview of the different topologies features. Note: "-" means the performance depends on the cable that the topology used

3.6 Parameters

The system's components are associated a set of parameters: producers have the following parameters

- *bandwidth* required to transit video data to consumers,
- *free-storage capacity* needed for a time on a consumer to store video data. The needed storage is based on various factors include image resolution used, compression type and ratio, frame rate and scene complexity.

and consumers have

- *free-storage capacity* of a node which is physically unoccupied disk space to store video data,
- *non-reserved storage capacity* of a node which is abstracted non-reserved disk space to store video data,
- *available computational power* (ACP) of a node which is un performed processing power. The ACP is needed to take the video data as much as fast.

Therefore, all components must have the following parameters: bandwidth, free-storage capacity, non-reserved storage capacity, and available computation power of each node of the basic consumers. The parameters are can vary and depends on each component, i.e., the parameters are dynamically changed.

3.7 Hardware and Software

The roles of the components in the physical world need hardware and software considerations to perform actions of produce, store and transfer video data.

3.7.1 Hardware features

The following hardware features may be considered when planning to build a DC.

- A cluster is a group of disks or nodes to make up storage where the data can be stored.
- Disk compression is technology that increases the apparent capacity of disk storage by encoding the data to take up less physical storage space on disk [32]. The compression and decompression of video data are performed automatically at the time storage.
- Network cables to connect nodes in the cluster and arrangements to connect geographically distributed clusters.
- Automated storage system, a DC, can be built with embedded clusters and a server which runs CM.

3.7.2 Software features

The following software features may be considered when planning to build a DC.

- Hierarchical storage system mechanism is on the top of operating system.
- Installation of all producer drivers in the clusters.
- Registration of all producers and nodes of basic consumers.
- Installation of CM API on each DC.

Chapter 4

The DTU-MVaaS Functionality

4.1 Interaction

Inside a DC, the system components are connected possibly through links with each other to produce video data and transfer it to storage system. In order to transfer video data from producers to consumers, different types of requests will be performed by the components of system since the system is request driven. Each request consists of

- a type (at least one of these.): a - assignment, ra - re-assignment, c - connect, s - storage or write, r - retrieval or read, se - search, d - disconnect, hc - health check, re - replication, aa - authentication and authorization, u-un-assignment request,
- arguments specifying the involved components,
- a return value indicating success or failure of the request and possibly more information. This return value is passed back to sender, by every component which the request passes.

4.1.1 Assignment Request

In a *assignment request* $\mathrm{Req}_a(X,\mathcal{K})$ component X has to be assigned to DC $\mathcal{K}$. This request is received by a CM. If the CM is the CM of the target DC $\mathcal{K}$ then it will perform the assignment operation. It depends on the type of component X since X could be either consumer or any other source.

If CM is not the CM of the target DC $\mathcal{K}$, but the target is below then, it will pass the request down to $\mathcal{K}$ and return the result of the operation to the sender of the request. If CM is not the CM of the target DC $\mathcal{K}$, and the target is not below then, it will pass the request upwards to the CM of the DC immediately above.

Note: An assignment does not involve any physical data connection inside the DC. It merely registers the component X with the DC $\mathcal{K}$, that is the CM knows that X is in $\mathcal{K}$ and the CM gets the relevant information (links etc), and is able to perform low intensity communication. No communication of X with components in $\mathcal{K}$ other than the CM is established.

4.1.2 Un-assignment Request

In a *un-assignment request* $\mathrm{Req}_u(X,\mathcal{K})$ component X has to be disconnected from any component in DC $\mathcal{K}$. This request is send by the CM of $\mathcal{K}$. If the CM is not the CM of the target DC $\mathcal{K}$ then it will perform the un-assignment operation. It depends on the type of component X since X could be any component (user, producer, consumer). The component X then is no longer part of DC $\mathcal{K}$.

4.1.3 Re-assignment Request

In a *re-assignment request* $\mathrm{Req}_{ra}(X,\mathcal{K},\mathcal{K}')$ some component X (user, producer, consumer) which currently is assigned to DC $\mathcal{K}$ has to be assigned to a different DC $\mathcal{K}'$, and to be disconnected from $\mathcal{K}$ by an un-assignment request $\mathrm{Req}_u(X,\mathcal{K})$. The request is passed upwards from $\mathcal{K}$ in the hierarchy until it reaches the lowest common ancestor $\mathcal{K}^*$ of $\mathcal{K}$ and $\mathcal{K}'$. The CM of $\mathcal{K}^*$ sends a un-assignment request $\mathrm{Req}_u(X,\mathcal{K})$ to $\mathcal{K}$. When this has succeeded, the CM of $\mathcal{K}^*$ sends a assignment request $\mathrm{Req}_a(X,\mathcal{K}')$ to $\mathcal{K}'$.

4.1.4 Storage Request

In a *storage request* $\text{Req}_s\,(X, \mathcal{C})$ some component X has to be requested a assignment request $\text{Req}_a\,(X, \mathcal{K})$ to DC $\mathcal{K}$ for storage. This request is received by the CM of $\mathcal{K}$. The CM will be performed an assignment operation depending on the type of component X. If the CM of $\mathcal{K}$ is unable to handle the current storage request, then *re-assignment request* $\text{Req}_{ra}\,(X, \mathcal{K}, \mathcal{K}')$ re-assigns the storage request $\text{Req}_s\,(X, \mathcal{C})$.

4.1.5 Connection Request

In a *connection request* $\text{Req}_c\,(X, \mathcal{C}, V, R)$ some X (acting a producer for the video stream V) has to be pysically connected to consumer $\mathcal{C}$. This request is received by the CM of that DC $\mathcal{K}$ to which X is assigned. The CM of $\mathcal{K}$ will perform the connection operation. That is the CM finds a route R from X to $\mathcal{C}$ inside $\mathcal{K}$ along which V can be sent.

If the CM is unable to handle the current connection request, then a *re-assignment request* $\text{Req}_{ra}\,(X, \mathcal{K}, \mathcal{K}')$ is issued. This assignes the producer X to a different DC $\mathcal{K}'$. Then the CM of $\mathcal{K}'$ issues $\text{Req}_c\,(X, \mathcal{K}', V, R)$.

Alternatively: the CM of $\mathcal{K}$ finds a DC $\mathcal{K}'$ issues $\text{Req}_{ra}\,(X, \mathcal{K}, \mathcal{K}')$ and a new storage request in $\mathcal{K}'$: $\text{Req}_s\,(X, \mathcal{K}')$.

4.1.6 Retrieval Request

In a *retrieval request* $\text{Req}_{re}\,(X, \mathcal{K})$, a user has to be connected a DC K with an assignment request $\text{Req}_a\,(X, \mathcal{K})$ to read or watch video. If requested video is not in the connected DC $\mathcal{K}$ then, the re-assignment request $\text{Req}_{ra}\,(X, \mathcal{K}, \mathcal{K}')$ re-assigns to requested DC $\mathcal{K}^*$. The targeted DC $\mathcal{K}^*$ could be below or above the DC $\mathcal{K}$ in the hierarchy.

4.1.7 Health check Request

In a *health check request* $\text{Req}_{hc}\,(X, \mathcal{K})$ some component X inside a DC $\mathcal{K}$ has to be reported its status to the CM of $\mathcal{K}$. The component X could be a producer, or a consumer, or a node, or a video stream. The components inside DC $\mathcal{K}$ have to be connected to the CM of DC $\mathcal{K}$ to report their health status regularly to the CM. The *health check request* always issued by the CM of $\mathcal{K}$.

4.1.8 Search Request

The basic search query service should be fitted in WP1.

4.1.9 Authentication and Authorization Request

The authentication and authorization should be fitted in WP3. The sample request is: In a *authentication and authorization request* $\mathrm{Req}_{aa}(A, \mathcal{K})$, a user A has to be sent an authentication request to the CM of $\mathcal{K}$ to get authorization for view requested video. The request always goes to unique entry point, i.e., the CM of $\mathcal{K}$.

4.2 Service Level Agreement

Producers, users and consumers and their services engage in a Service Level Agreement (SLA) [33, 34]. SLA is to guarantee that the system can assign video streams to consumers in a bounded time without any interruption. Every single component in the system needs to deliver its functionality with even tighter bounds. For instance, a service guaranteeing that the system will assign 99.9% of storage requests within t_s seconds for a peak load as of system requirement R7.

4.3 Parameters

We require that the MVaaS system's components must supply basic parameters to quantify the system performance. The parameters include free-storage capacity, available computational power, and bandwidth. In our system we model this by a single, non-negative number which we call *demand*. Corresponding, every consumer has associated a non-negative number which we call *capacity*. The reason for using one number only is that in the presence of more parameters, all of them on the demand side have to be less than the corresponding ones on the capacity side. Thus the abstraction to a single number is suitable. For video streams, the demand is a combination of bandwidth and storage space to ensure a longer-time connection. This will enable the system to match incoming storage request with storage facilities.

Using a single number storage request minimizes communication overhead of the scheduling algorithms to schedule it to a corresponding capacity of the consumer [35]. The single number demand helps to enforce proportional sharing of capacity of the consumer. Each storage request has the corresponding demand, and scheduling algorithm reserves for it a minimum share of the capacity of the consumer proportional to its demand [36].

4.4 Storage request Generation

The MVaaS system is a request driven system where the requests are defined in Section4.1. A request triggers various actions in the system, for instance, a producer interacts with the system through a storage request $\text{Req}_s\,(X, \mathcal{K})$. A storage request is generated by either producer or any other source, for instance, guards, customers, consumers, etc. to make an interaction between a producer and its associated DC $\mathcal{K}$. In a case of data transfers between DCs $\mathcal{K}_i$ and $\mathcal{K}_j$, a consumer of $\mathcal{K}_i$ or $\mathcal{K}_j$ can generate storage requests. A storage request is not a video stream itself. A storage request asks for a video stream to be routed to a consumer which has enough free capacities to store the stream.

The CM of DC $\mathcal{K}$ handles all storage requests and is responsible for assigning video streams to consumers of $\mathcal{K}$ based on their capacities. If the CM of $\mathcal{K}$ is not able to satisfy a storage request locally, then this request is passed on the higher level DC $\mathcal{K}'$. The entry point for a storage request, generated by any component inside $\mathcal{K}$, is the CM of the DC $\mathcal{K}$. A component is associated with any point in time, every component (consumer, producer, link) is associated to a unique CM of a unique DC.

4.5 Dynamics of the MVaaS

In order to store video streams, the CM of $\mathcal{K}$ has to establish a connection between associated producers and consumers of $\mathcal{K}$, check the health status of all components and video streams and error reports in $\mathcal{K}$, if we consider a single DC $\mathcal{K}$, i.e., the way the request control system, which we described in the following sections with text, pseudocodes and figures.

4.5.1 Prerequisites for all Requests

The CM of $\mathcal{K}$ maintains *status information* of DC $\mathcal{K}$, which is the current status of its consumers health, their free capacities and their mapping, health status of all of its associated producers and their mapping, all types of requests mapping, etc.

Typically, the producers mapping is comprised with IP address, physical location information, unique number of producer.

Consumers mapping consists of its address/routing information. Consumers of $\mathcal{K}$ regularly report their health and storage conditions to the CM of $\mathcal{K}$ such that the status information is updated regularly on the CM.

The way of specific requests works in the system is:

The way of the request req_x system

1. **Source:** : Here is stated, where that request comes from.
2. **Target:** : Here is stated, where that request reaches.
3. **Process:** : Here is stated, how that request is processed with possible connection interaction.

In general, the way of the specific request req_x system states that the standard description/format of following requests ways.

4.5.2 Storage Request

The following steps are needed to execute a storage request $\text{Req}_s\,(X, \mathcal{K})$ in the system, which are depicted in the Figure 4.1 and detailed in the algorithm 1.

1. **Source:** A storage request $\text{Req}_s\,(X, \mathcal{K})$ is issued by some component. This can be a (an intelligent) camera, a storage device which needs to connect to a producer of a storage device which wants to send data to another place.
2. **Target:** The request is send to the CM of that DC $\mathcal{K}$ where the issuer is currently assigned to. The CM of $\mathcal{K}$ waits and collects a batch Q of storage requests in a time interval.

3. Process: 1. The CM of $\mathcal{K}$ executes $scheduleReport(\mathcal{K}, Q)$ algorithm 2 to get the scheduling report S_{report} 4.5.2.1.

2. Un-handled storage requests Q_{rest}, if any, which are reported in S_{report} will be sent to the DC immediately above.

3. Let us consider V_i, $i = 0, \dots, \ell$, the video streams which are associated with storage requests $\text{Req}_s\,(X_i, \mathcal{K})$. Based on the scheduling report S_{report}, the CM knows for each V_i the consumer $\mathcal{C}_{j(i)}$ to which the request $\text{Req}_s\,(X_i, \mathcal{K})$ was associated.

   Then the CM issues a connection request $\text{Req}_c\left(X_i, \mathcal{C}_{j(i)}, V_i, R_i\right)$ and executes $StorageConnection(X_i, \mathcal{C}_{j(i)}, V_i, R_i)$ algorithm 1 to connect video stream. Thereby the stream V_i is routed from X_i to $\mathcal{C}_{j(i)}$

Algorithm 1 Request control system for a Storage request in a DC - $StorageConnection(X_i, \mathcal{C}_{j(i)}, V_i, R_i)$

```
1: ◁ Require ▷
2: Let $Q = (\text{Req}_s\,(X_i, \mathcal{K})\,, i = 0, \dots, \ell)$ be a batch of storage requests;
3: Let $\mathcal{K}'$ be the DC immediately higher level of a DC $\mathcal{K}$;
4:
5: ◁ Procedure ▷
6: $S_{report} \leftarrow scheduleReport(\mathcal{K}, Q)$                ▷ Algorithm 2;
7: Extract $Q_{rest}$ and list of tuples $\langle X_i, \mathcal{C}_{j(i)}, V_i, R_i \rangle$ from $S_{report}$;
8: if ($Q_{rest} \neq \emptyset$) then
9:     send $Q_{rest}$ to $\mathcal{K}'$;
10: end if
11: Establish video stream $V_i$ connection (or routing $R_i$) from producer $X_i$ to
    consumer $\mathcal{C}_{j(i)}$;
12: for all tuples $\langle X_i, \mathcal{C}_{j(i)}, V_i, R_i \rangle$ do
13:     call $\text{Req}_c\left(X_i, \mathcal{C}_{j(i)}, V_i, R_i\right)$;
14: end for
```

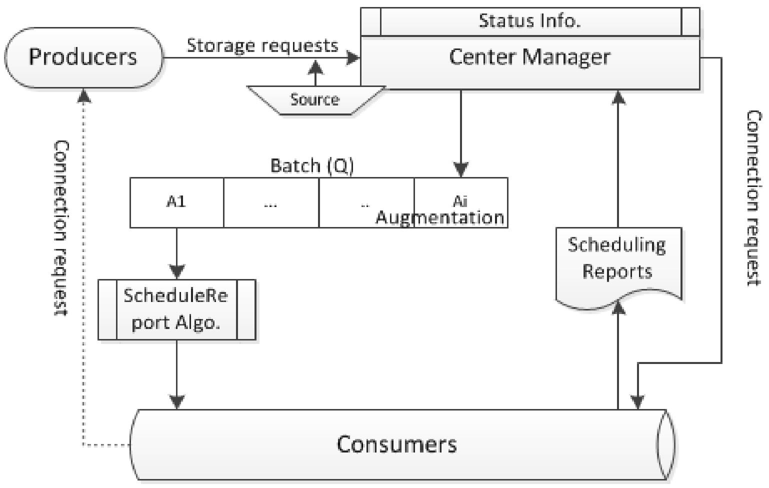

Figure 4.1: A request control system for a Storage request

4.5.2.1 Scheduling report

Scheduling report is a report which generates by $scheduleReport(\mathcal{K}, Q)$ algorithm 2 and runs on CM of DC $\mathcal{K}$. The scheduling report S_{report} consists of:

- A batch Q_{rest} of storage requests which are un-handled, if any, and
- A list of tuples $\langle X_j, \mathcal{C}_{i(j)}, V_j, R_j \rangle$ where consumer $\mathcal{C}_{i(j)}$ is able to handle video stream V_j which is produced by X_j. Let R_j be the routing information which routs the video stream V_j from X_j to $\mathcal{C}_{i(j)}$.

Algorithm 2 Scheduling report in a DC - *scheduleReport*$(\mathcal{K}, Q)$

```
1:  ◁ Require ▷
2:  Let $Q = (\mathrm{Req}_s(X_i, \mathcal{K}), i = 0, \ldots, \ell)$ be a batch of storage requests;
3:  ◁ Procedure ▷
4:  if $\mathcal{K}$ is DC then
5:      Let $\mathcal{C}_0, \ldots \mathcal{C}_k$ be the consumers inside $\mathcal{K}$
6:      for $i = 0, \ldots, k$ do
7:          Compute $F(\mathcal{C}_i)$;
8:      end for
9:      Sort consumers according to $F(\mathcal{C}_i)$;
10:     Re-index such that $F(\mathcal{C}_0) \geq \ldots \geq F(\mathcal{C}_k)$;
11:     $i = 0$
12:     while $(Q \neq \emptyset$ and $i \leq k)$ do
13:         Let $Q_i \subseteq Q$ such that $D(Q_i) \leq \lambda F(\mathcal{C}_i)$;
14:         $Q_{rest} \leftarrow scheduling(\mathcal{C}_i, Q_i)$;                    ▷ Algorithm 8;
15:         $Q_{assigned} \leftarrow (Q_i \setminus Q_{rest})$;
16:         $Q \leftarrow (Q \setminus Q_{assigned})$;
17:         for all $\mathrm{Req}_s(X_j, \mathcal{K})$ in $Q_{assigned}$ do
18:             Let $V_j$ be the video stream associated with component $X_j$ in
    $\mathrm{Req}_s(X_j, \mathcal{K})$;
19:             Let $R_j$ be the routing information;                ▷ To be computed;
20:             add tuple $\langle X_j, \mathcal{C}_{i(j)}, V_j, R_j \rangle$ to $S_{report}$;
21:         end for
22:         $i = i + 1$;
23:     end while
24:     add $Q$ to $S_{report}$;        ▷ Now, $Q$ consists un-handled $\mathrm{Req}_s(X_i, \mathcal{K})$
25:     return $S_{report}$;
26: else($\mathcal{K}$ is a cluster of nodes)
27:     for all nodes $M$ in $\mathcal{K}$ do
28:         Compute $F(M)$;
29:     end for
30:     Let $M^*$ be the node with max $F(M)$
31:     $Q_{rest} \leftarrow scheduling_b(M^*, Q)$;                          ▷ Algorithm 9
32:     $Q_{assigned} \leftarrow (Q \setminus Q_{rest})$
33:     for all $\mathrm{Req}_s(X_j, \mathcal{K})$ in $Q_{assigned}$ do
34:         Let $V_j$ be the video stream associated with component $X_j$ in
    $\mathrm{Req}_s(X_j, \mathcal{K})$;
35:         Let $R_j$ be the routing information;                    ▷ To be computed;
36:         add tuple $\langle X_j, M_j^*, V_j, R_j \rangle$ to $S_{report}$;
37:     end for
38:     add $Q_{rest}$ to $S_{report}$;      ▷ $Q_{rest}$ consists un-handled $\mathrm{Req}_s(X_i, \mathcal{K})$
39:     return $S_{report}$;
40: end if
```

4.5.3 Connection Request

4.5.3.1 Connection Request with passive producers in same DC

The following steps are needed to execute a connection request $\mathrm{Req}_c\left(X_i, \mathcal{C}_{j(i)}, V_i, R_i\right)$ in a DC $\mathcal{K}$, which are depicted in the figure 4.2 and detailed in the Algorithm 3.

1. **Source:** The CM of DC $\mathcal{K}$ sends a connection request $\mathrm{Req}_c\left(X_i, \mathcal{C}_{j(i)}, V_i, R_i\right)$ to its consumer $\mathcal{C}_{j(i)}$.

2. **Target:** The consumer $\mathcal{C}_{j(i)}$ receives the request $Req_c(X_i, \mathcal{C}_{j(i)}, V_i, R_i)$.

3. **Process:** 1. $\mathcal{C}_{j(i)}$ has the drivers and knows component X_i through routing R_i. The $\mathcal{C}_{j(i)}$ begins to pull the stream V_i and record it.

   2. The CM of $\mathcal{K}$ issues health check request regularly $\mathrm{Req}_{hc}\left(V_i, \mathcal{C}_{j(i)}\right)$ to collect the failures of video stream V_i, if any.

Algorithm 3 Request control system for the connection request in a DC - $videoConnection(X_i, \mathcal{C}_{j(i)}, V_i, R_i)$

```
1: ◁ Require ▷
2: A list of tuples $\langle X_i, \mathcal{C}_{j(i)}, V_i, R_i \rangle$ from $S_{report}$;
3:
4: ◁ Procedure ▷
5: for all tuples $\langle X_i, \mathcal{C}_{j(i)}, V_i, R_i \rangle$ do
6:     call $\mathrm{Req}_c\left(X_i, \mathcal{C}_{j(i)}, V_i, R_i\right)$;
7:     call $\mathrm{Req}_{hc}\left(V_i, \mathcal{C}_{j(i)}\right)$;
8: end for
```

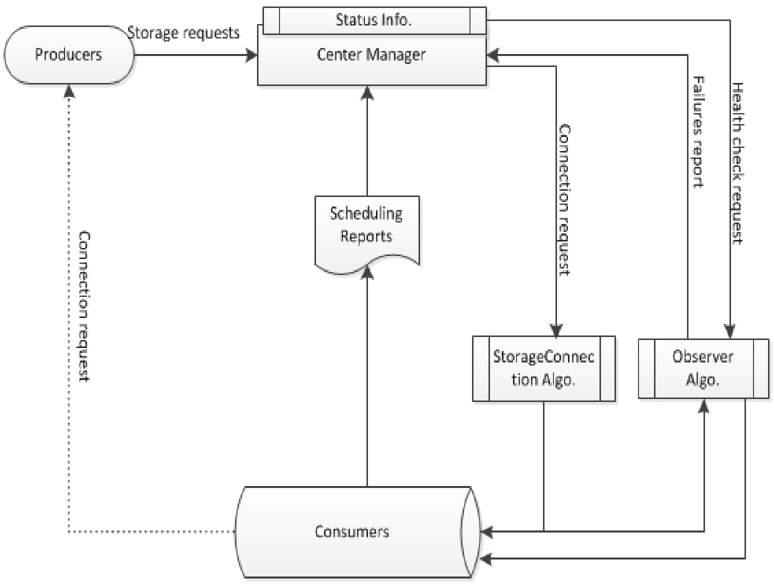

Figure 4.2: A request control system for Connection request with passive producers in same DC

4.5.3.2 Connection Request with Passive Producers to Consumer in Lower Level DC

The following steps are needed to execute a connection request $\text{Req}_c\,(X_i, \mathcal{C}_i^*, V_i, R_i)$, where X_i is in DC $\mathcal{K}$ and consumer $\mathcal{C}_i^*$ is in DC $\mathcal{K}^*$, and $\mathcal{K}^*$ is below $\mathcal{K}$ in the hierarchy. The executed steps are depicted in the Figure 4.3 and detailed in the algorithm 4.

1. **Source:** The CM of DC $\mathcal{K}$ sends connection request $\text{Req}_c\,(X_i, \mathcal{C}_i^*, V_i, R_i)$ to a consumer $\mathcal{C}_i^*$ of DC $\mathcal{K}^*$ through $\mathcal{K}^*$'s CM.
2. **Target:** CM of the $\mathcal{K}^*$.
3. **Process:** 1. The CM of DC $\mathcal{K}^*$ performs re-assignment request $\text{Req}_{ra}\,(X_i, \mathcal{K}, \mathcal{K}^*)$. If this succeeds, then X_i is associated with DC $\mathcal{K}^*$ and no longer associated with $\mathcal{K}$.
   2. Connect X_i to $\mathcal{C}_i^*$ within the DC $\mathcal{K}^*$ to pull the stream V_i and record it, as described in Section 4.5.3.1.
   3. The CM of $\mathcal{K}^*$ issues health check request to collect failures inside a DC $\mathcal{K}^*$, if any.

Algorithm 4 Request control system for a Connection request in a lower level DC - $videoConnectionLowLev(X_i, \mathcal{C}_i^*, V_i, R_i)$

```
1: ◁ Require ▷
2: A tuples $\langle X_i, \mathcal{C}_i^*, V_i, R_i \rangle$ from $S_{report}$;
3: Let $\mathcal{K}^*$ be the DC lower level of DC $\mathcal{K}$ which contains $\mathcal{C}_i^*$;
4:
5: ◁ Procedure ▷
6: call $\text{Req}_{ra}(X_i, \mathcal{K}, \mathcal{K}^*)$;
7: call $\text{Req}_{c}(X_i, \mathcal{C}_i^*, V_i, R_i)$;
8: call $\text{Req}_{hc}(V_i, \mathcal{C}_i^*)$;
```

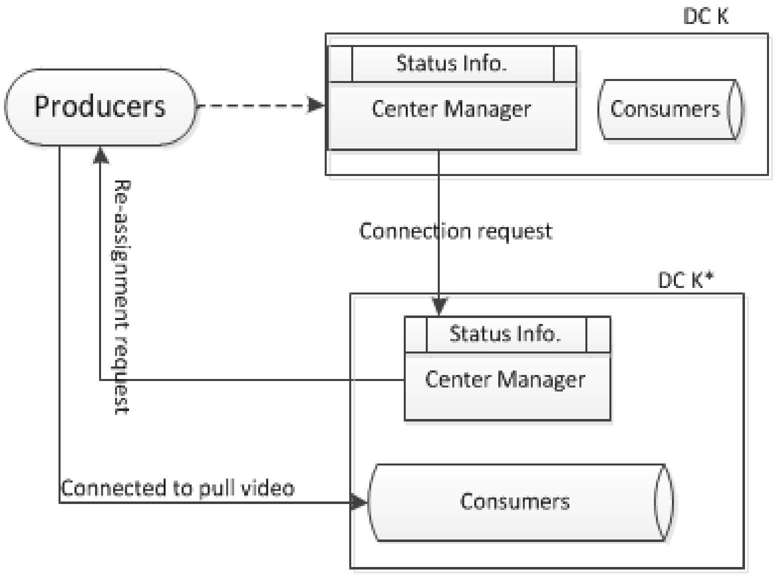

Figure 4.3: A request control system for a Connection request with passive producers to Consumer in lower level DC

4.5.3.3 Connection Request with passive producers to Consumer in Higher Level DC

The following steps are needed to execute a connection request $\text{Req}_c(X_i, \mathcal{C}_i^{**}, V_i, R_i)$, where X_i is in DC $\mathcal{K}$ and basic consumer $\mathcal{C}_i^{**}$ is in a DC $\mathcal{K}'$, where $\mathcal{K}'$ possibly, be same level but not below the DC $\mathcal{K}$ in hierarchy. The DC $\mathcal{K}^{**}$ is above $\mathcal{K}$ in the hierarchy and the lowest common ancestor to $\mathcal{K}$ and $\mathcal{K}'$. The executed steps are depicted in the Figure 4.4 and detailed in the algorithm 5.

1. **Source:** The CM of $\mathcal{K}$ sends connection request $\text{Req}_c\,(X_i, \mathcal{C}_i^{**}, V_i, R_i)$ to the CM of $\mathcal{K}^{**}$.

2. **Target:** The CM of $\mathcal{K}^{**}$.

3. **Process:**
   1. If $\mathcal{C}_i^{**}$ is not below $\mathcal{K}$ then $\mathcal{C}_i^{**}$ is a Basic consumer in some DC $\mathcal{K}'$. Let $\mathcal{K}^{**}$ be the lowest common ancestor to $\mathcal{K}'$ and $\mathcal{K}$.
   2. $\mathcal{K}$ sends re-assignment request $\text{Req}_{ra}\,(X_i, \mathcal{K}, \mathcal{K}^{**})$ upwards to $\mathcal{K}^{**}$, then $\mathcal{K}^{**}$ sends the request downwards in the hierarchy, as described in Section 4.5.3.2. If this succeeds, then X_i is associated with DC $\mathcal{K}^{**}$ and no longer associated with $\mathcal{K}$.
   3. Connect X_i to $\mathcal{C}_i^{**}$ to pull the stream V_i and record it.
   4. The CM of $\mathcal{K}^{**}$ issues health check request to collect failures inside a DC $\mathcal{K}^{**}$, if any.

Algorithm 5 Request control system for a Connection request in a higher level DC - $videoConnectionHighLev(X_i, \mathcal{C}_i^{**}, V_i, R_i)$

```
1: ◁ Require ▷
2: A tuple $\langle X_i, \mathcal{C}_i^{**}, V_i, R_i \rangle$ from $S_{report}$;
3: Let $\mathcal{K}^{**}$ be the DC and the lowest common ancestor to $\mathcal{K}$ and a DC $\mathcal{K}'$
   which contains basic consumer $\mathcal{C}_i^{**}$.
4:
5: ◁ Procedure ▷
6: call $\text{Req}_{ra}\,(X_i, \mathcal{K}, \mathcal{K}^{**})$;
7: call $\text{Req}_c\,(X_i, \mathcal{C}_i^{**}, V_i, R_i)$;
8: call $\text{Req}_{hc}\,(V_i, \mathcal{C}_i^{**})$;
```

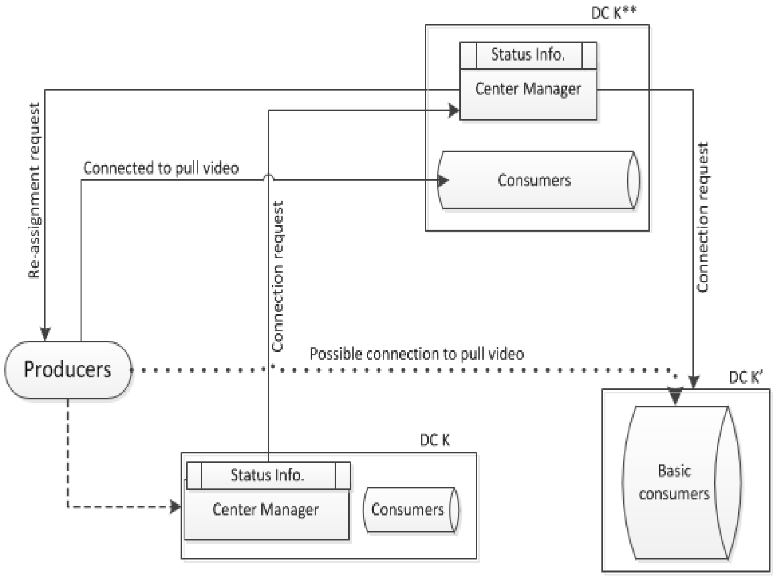

Figure 4.4: A request control system for a Connection request with passive producers to Consumer in above level DC

4.5.4 Re-assignment Request

The following steps are needed to execute a re-assignment request $\mathrm{Req}_{ra}\left(X_i, \mathcal{K}, \mathcal{K}'\right)$, where X_i is in DC $\mathcal{K}$, which will be assigned to different DC $\mathcal{K}'$. The DC $\mathcal{K}'$ could be higher level or lower level or possibly, lowest common ancestor to $\mathcal{K}$ and a DC $\mathcal{K}''$.

1. **Source** The CM of $\mathcal{K}$ sends re-assignment request $\mathrm{Req}_{ra}\left(X_i, \mathcal{K}, \mathcal{K}'\right)$ to the CM of $\mathcal{K}'$.

2. **Target** The CM of $\mathcal{K}'$ receives re-assignment request.

3. **Process** The CM of DC $\mathcal{K}$ issues re-assignment request $\mathrm{Req}_{ra}\left(X_i, \mathcal{K}, \mathcal{K}'\right)$ then the producer X_i is no longer associated with DC $\mathcal{K}$ since it has to be disconnected with the CM of $\mathcal{K}$ by un-assignment request $\mathrm{Req}_{u}\left(X_i, \mathcal{K}\right)$. Now, X_i is associated with $\mathcal{K}'$. The re-assignment request is passed upwards from $\mathcal{K}$ in the hierarchy until it reaches the lowest common ancestor $\mathcal{K}''$ of $\mathcal{K}$ and $\mathcal{K}'$.

Algorithm 6 Request control system for a Re-assignment request in a DC - $reAssignmentReq(X_i, \mathcal{K}, \mathcal{K}')$

```
1: ◁ Require ▷
2: Let $\mathcal{K}'$ be a DC either lower or higher level of the DC $\mathcal{K}$ and $\mathcal{K}''$ be the
   lowest common ancestor of $\mathcal{K}$ and $\mathcal{K}'$.
3:
4: ◁ Procedure ▷
5: call $\text{Req}_{ra}(X_i, \mathcal{K}, \mathcal{K}')$;        ▷ it calls un-assignment request $\text{Req}_{u}(X_i, \mathcal{K})$
```

4.5.5 Failures detection

The following steps are needed to execute a health check request $\text{Req}_{hc}(X, \mathcal{K})$, where X could be any component in the system. The executed steps are detailed in the algorithm 7.

Source: The CM of DC $\mathcal{K}$ sends a health check request $\text{Req}_{hc}(X, \mathcal{K})$.

Target: The target is the component X.

Process: The CM of $\mathcal{K}$ detects components failures inside $\mathcal{K}$. For instance, a producer failed because of lack of bandwidth. The CM of $\mathcal{K}$ receives failures report.

Algorithm 7 Request control system for a Health check request in a DC - $Observer(X, \mathcal{K})$

```
1: ◁ Require ▷
2: Let $X$ be any component inside a DC $\mathcal{K}$;
3: ◁ Procedure ▷
4: call $\text{Req}_{hc}(X, \mathcal{K})$
5: if failures detect inside $\mathcal{K}$ then
6:     report to CM of $\mathcal{K}$;
7: end if
```

4.5.6 Mechanism for self-maintenance

The system requires two fundamental properties for self-maintenance: fault tolerance and automatic repair. When the system components fail or misbehave, the most important system property is its capability to continue operating correctly, though possibly at reduced performance. Such fault tolerance, however, is not sufficient for self-maintenance, because further failures might render correct operation impossible. Instead, a system must, with high probability, detect and repair a failure before further failures cause a catastrophic error or disrupt service [37]. After detecting a failure in the system by a health check request, the system can repair itself with a 'Automatic repair Request' to be developed.

4.5.6.1 Fault tolerance / Automatic repair Request

An external research results will come here....

4.6 Details of Storage request Handling

To demonstrate storage request handling, we take the advantage of the scalability feature of the system to consider a single scalable DC. Considering a single DC demonstration reduces the time and memory required for de-centralized scheduling [38] of storage requests because lower-level DCs are much smaller than the entire hierarchy or top-most DC or whole system. The DC has at least one basic consumer or a consumer (DC) or both. The motivation for decentralized storage request handling or scheduling is to "avoid to suffer from diminished fault-tolerance and potential for performance bottleneck" [39].

In the de-centralized approach, the CM of $\mathcal{K}$ gets required information of all consumers inside a $\mathcal{K}$ rather than entire system information for storage request handling or to make scheduling decisions. The consumers relay the possible required information include load, computation power and storage capacity to the CM. For storage request handling inside a DC $\mathcal{K}$, the CM of $\mathcal{K}$ consults immediate lower-level consumers to distribute storage requests across the consumers which are directly embedded to the $\mathcal{K}$ instead of all lower-level consumers which can be geographically distributed and interconnected by WAN. The immediate lower-level consumers process recursively. In case that the consumers inside a $\mathcal{K}$ cannot (recursively) handle the request, then the CM of $\mathcal{K}$ will contact the immediate higher-level DC of $\mathcal{K}$.

In storage request handling,

- First, reduce the communication latency of storage requests scheduling by determining a consumer with maximum of free capacities to process the storage requests.

- Second, reduce re-schedules to schedule storage requests in the system to store a complete video stream.
- Finally, exploit a smart virtual routing technique to transfer the storage requests between consumers.

4.6.1 Scheduling

For scheduling, consumers employ a scheduling mechanism described below where the system first considers the situation in which only storage requests arrive. The consumers relay the required information to the CM, which in turn makes scheduling decisions. The scheduling decisions adapt to the number of storage requests to match the workload variation. The workload depends on CPU speed, memory, cache structure, motion or event detection, management functions, compression, decompression and even the program type. Therefore, a scheduling takes into account the number of storage requests, computational power, bandwidth and storage of each consumer for making scheduling decisions. As mentioned above (section 4.3), capacities can be described by a single, non-negative number. In case of a storage request, this number specifies the amount of capacities required by the storage request. We call this the *demand*.

Let consider a batch $Q = (\mathrm{Req}_s\,(X_i, \mathcal{K})\,, i = 0, \ldots, \ell)$ of storage requests. This can be the storage requests collected in the current time interval I or a single storage request in case of immediate processing. The reason to collect a batch of storage requests instead of a single storage requests is to get highest services from minimal service resources [40]. For the moment, we do not consider the bandwidth restriction, but assume that any desired connection can be established. Let $\mathcal{C}_0, \ldots, \mathcal{C}_k$ be consumers (either basic consumers or DCs or both) inside a DC $\mathcal{K}$.

In a DC $\mathcal{K}$, the CM of $\mathcal{K}$ runs a scheduling algorithm *scheduling*$(\mathcal{K}, Q)$ which collects a batch of storage requests (Q) in an interval and schedules to a consumer with maximum free capacity. Storage requests that cannot be handled by this consumer are returned to the CM and assigned to another consumer.

4.6.1.1 *scheduling*$(\mathcal{K}, Q)$ Algorithm

Let $D(\mathrm{Req}_s\,(X_i, \mathcal{K}))$ denote the demand of a storage request $\mathrm{Req}_s\,(X_i, \mathcal{K})$. For all consumers, it specifies the amount of free capacity that the consumer can offer and is denoted by $F(\mathcal{K})$. The CM computes the demand $D(Q)$ of the batch of storage requests by

$$D(Q) = \sum_{i=0}^{\ell} D(\mathrm{Req}_s\,(X_i, \mathcal{K}))\ . \tag{4.1}$$

The CM then polls all consumers to receive the information of their free capacities.

Consider a basic consumer $\mathcal{C}$ first and let $M_0, \dots, M_s$ be the nodes in $\mathcal{C}_i$. Let $F(M_j)$ denote the free capacities of node M_j. The CM computes the total free capacities $F(\mathcal{C})$ for a basic consumer by

$$F(\mathcal{C}) = \sum_{j=0}^{s} F(M_j)\ . \tag{4.2}$$

The free capacities of a non-basic consumer (i.e., a DC) $\mathcal{K}$ are computed recursively: Let $\mathcal{C}_0, \dots, \mathcal{C}_k$ be the consumers inside $\mathcal{K}$. The free capacities of $\mathcal{K}$ are

$$F(\mathcal{K}) = \sum_{j=0}^{k} F(\mathcal{C}_j)\ . \tag{4.3}$$

In case $\mathcal{K}$ is a basic consumer, let M^* be a node which has the most free capacities of all nodes in $\mathcal{K}$, i.e.,

$$F(M^*) = \mathrm{argmax}\{F(M_j) \mid M_j \in \mathcal{K}\} \tag{4.4}$$

A slack margin is introduced into decision of scheduling the storage requests to $\mathcal{K}$ if $D(Q) \leq \lambda F(\mathcal{K})$, and scheduling the storage requests to M^* in $\mathcal{K}$ if $D(Q) \leq \lambda F(M^*)$, for some $\lambda \in [0, 1]$.

The Algorithm 8 visits the consumers inside a $\mathcal{K}$ in decreasing order of free capacities. Let $\mathcal{C}_i$ be the currently visited consumer. The algorithm 8 selects and removes some storage requests from the batch Q into a sub-batch Q_i such that their total demand is less than the free capacities of $\mathcal{C}_i$ (possibly including a slack): $D(Q_i) \leq \lambda F(\mathcal{C}_i)$ and assigns Q_i to $\mathcal{C}_i$. The Q_i can be selected in FIFO (First In, First Out) order from the Q. As mentioned above, it can happen that $\mathcal{C}_i$ cannot handle all of Q_i. Then the un-handled storage requests are again added to Q. If $\mathcal{C}_i$ is a DC, then the algorithm proceeds recursively. If $\mathcal{C}_i$ is a basic consumer, then Algorithm 9 is called, which terminates the recursion.

Algorithm 8 Scheduling in a DC - *scheduling*$(\mathcal{K}, Q)$

```
1: $\triangleleft$ Require $\triangleright$
2: $Q = (\mathrm{Req}_s(X_i, \mathcal{K}), i = 0, \ldots, \ell)$;
3: Let $\mathcal{C}_0, \ldots \mathcal{C}_k$ be the consumers inside a DC $\mathcal{K}$;
4: Compute $D(Q)$;
5:
6: $\triangleleft$ Procedure $\triangleright$
7: if $\mathcal{K}$ is DC then
8:     for $i = 0, \ldots, k$ do
9:         Compute $F(\mathcal{C}_i)$;
10:    end for
11:    Sort consumers according to $F(\mathcal{C}_i)$;
12:    Re-index such that $F(\mathcal{C}_0) \geq \ldots \geq F(\mathcal{C}_k)$;
13:    $i = 0$
14:    while ($Q \neq \emptyset$ and $i \leq k$) do
15:        Let $Q_i \subseteq Q$ such that $D(Q_i) \leq \lambda F(\mathcal{C}_i)$; $\triangleright$ $Q_i$ selected in FIFO order from $Q$
16:        $Q_{return} \leftarrow scheduling(\mathcal{C}_i, Q_i)$;    $\triangleright$ Algorithm 8;
17:        $Q \leftarrow (Q \setminus Q_i) \cup Q_{return}$;
18:        $i = i + 1$;
19:    end while
20:    if $Q \neq \emptyset$ then
21:        return $Q$;    $\triangleright$ Now, $Q$ consists un-handled $\mathrm{Req}_s(X_i, \mathcal{K})$
22:    end if
23: else($\mathcal{K}$ is a cluster of nodes)
24:    for all nodes $M$ in $\mathcal{C}$ do
25:        Compute $F(M)$;
26:    end for
27:    Let $M^*$ be the node with max $F(M)$
28:    $Q_{rest} \leftarrow scheduling_b(M^*, Q)$;    $\triangleright$ Algorithm 9
29:    return $Q_{rest}$;    $\triangleright$ Now, $Q_{rest}$ consists un-handled $\mathrm{Req}_s(X_i, \mathcal{K})$
30: end if
```

4.6.1.2 *scheduling*$_b(M, Q)$ Algorithm

The scheduling procedure in a basic consumer is described in the algorithm *scheduling*$_b(M, Q)$. This procedure depends on the internal structure of a cluster or basic consumer. As an example, the cluster is organized as a Ring 3.5.2.1 topology so a batch Q can be processed at any node in the Ring without interruption. The motivation to select a Ring topology is low cost, easy to set up and maintain [11, 12] than other existing topologies include Tree 3.5.2.1, Mesh

3.5.2.2, Star 3.5.2.3, Bus 3.5.2.4 and Hypercube 3.5.2.6 topologies. The Ring topology has some drawbacks like one node failure causes the entire network failure in the Ring. We partially overcome this problem by allowing communication in two directions of the Ring or by allowing links to the node after the direct neighbor. In this case, the algorithm $scheduling_b(M, Q)$ distributes storage requests to two directions of the node until reaches the failure node. The most free capacities node changes its place for next iteration since its free capacities will be used by the assigned storage requests. Other nodes in the Ring can replace the most free capacities node place for next iteration.

Algorithm 9 Scheduling in a ring - $scheduling_b(M, Q)$

```
1: Let $Q = A_0, \dots, A_\ell$ be a batch of storage requests;
2: Let $M_0, \dots M_s$ be the nodes;
3: Every node $M$ has a flag f(M) which can be “start” or “process”; Initially
   for all M $f(M) =$ “process”;
4: Set flag $f(M) =$ ”start”
5: Split $Q = Q_0 \cup Q_L \cup Q_R$ such that: ;
6:    $D(Q_0) \leq \lambda F(M)$ and
7:    $D(Q_R) \sim D(Q_L)$;
8: handle $Q_0$ at $M$;
9: goto $scheduleL(left(M), Q_L)$;
10: goto $scheduleR(right(M), Q_R)$;
11: ____________________________________________
12: ◁ $scheduleL(M, Q)$ ▷
13: if $f(M) =$ ”start“ then
14:     return $Q$;
15: else
16:     Split $Q = Q_0 \cup Q_L$ such that: ;
17:        $D(Q_0) \leq \lambda F(M)$;
18:     handle $Q_0$ at $M$;
19:     call $scheduleL(left(M), Q_L)$;
20: end if
21: ____________________________________________
22: ◁ $scheduleR(M, Q)$ ▷
23: if $f(M) =$ ”start“ then
24:     return $Q$;
25: else
26:     Split $Q = Q_0 \cup Q_R$ such that: ;
27:        $D(Q_0) \leq \lambda F(M)$;
28:     handle $Q_0$ at $M$;
29:     call $scheduleR(right(M), Q_L)$;
30: end if
```

In the ring, M_i can communicate directly with M_{i-1} and M_{i+1} where the index calculations are performed such that M_0 and M_s are connected. The nodes M_{i-1} and M_{i+1} are called the ***left neighbor***, $left(M_i)$, and ***right neighbor***, $right(M_i)$, of M_i respectively. Note that when $D(Q) \leq F(M)$ then node M can handle all batch Q. The algorithm sends part of the batch left, part right from the starting node. At every node a part of the batch is kept which can be handled at this node. When no storage requests are left the algorithms terminates. Should part of the batch make a full round and return to the starting node, this is returned. Splitting the batch is done greedily, see[41], other method will be considered.

4.6.2 Proportional-share Scheduling

In the proportional-share scheduling 10, a batch of storage requests demand distributes to proportional-share of consumers free capacities. The algorithm 10 visits the consumers inside a DC $\mathcal{K}$ in decreasing order of free capacities. Let $S(\mathcal{C}_i)$ be the sharing capacity of consumer $\mathcal{C}_i$ from total capacity of all consumers inside $\mathcal{K}$. The algorithm 10 selects sub-batch Q_i from batch Q and schedules on $\mathcal{C}_i$, where $D(Q_i) \sim S(\mathcal{C}_i)$. The batch Q can be divided into no.of sub-batches based on no.of consumers inside DC $\mathcal{K}$. If a batch has one request then the first consumer in decreasing order of free capacities will get whole batch. If no consumer is ready to receive a batch then the scheduling algorithm sends the batch to immediately higher-level DC of DC $\mathcal{K}$.

Algorithm 10 Scheduling in a DC - $ProportionalScheduling(\mathcal{K}, Q)$

```
1: ◁ Require ▷
2: Let $Q = (\text{Req}_s(X_i, \mathcal{K}), i = 0, \ldots, \ell)$ be a batch of storage requests;
3: Compute $D(Q)$;
4:
5: ◁ Procedure ▷
6: if $\mathcal{K}$ is DC then
7:     Let $\mathcal{C}_0, \ldots \mathcal{C}_k$ be the consumers inside $\mathcal{K}$
8:     for $i = 0, \ldots, k$ do
9:         Compute $F(\mathcal{C}_i)$;
10:    end for
11:    Sort consumers according to $F(\mathcal{C}_i)$;
12:    Re-index such that $F(\mathcal{C}_0) \geq \ldots \geq F(\mathcal{C}_k)$;
13:    for $i = 0, \ldots, k$ do
14:        Compute $S(\mathcal{C}_i) = F(\mathcal{C}_i)/\sum_{j=0}^{k} F(\mathcal{C}_j)$;  ▷ Sharing of each consumer's
   free capacities from total free capacities
15:    end for
16:    $i = 0$
17:    while $(Q \neq \emptyset$ and $i \leq k)$ do
18:        Let $Q_i \subseteq Q$ such that $D(Q_i) \sim S(\mathcal{C}_i) \times D(Q)$;
19:        $Q_{return} \leftarrow assignment(\mathcal{C}_i, Q_i)$;                ▷ Algorithm 10;
20:        $Q \leftarrow (Q \setminus Q_i) \cup Q_{return}$;
21:        $i = i + 1$;
22:    end while
23:    if $Q \neq \emptyset$ then
24:        return $Q$;               ▷ Now, $Q$ consists un-handled $\text{Req}_s(X_i, \mathcal{K})$
25:    end if
26: else($\mathcal{K}$ is a cluster of nodes)
27:    for all nodes $M$ in $\mathcal{K}$ do
28:        Compute $F(M)$;
29:    end for
30:    Let $M^*$ be the node with max $F(M)$
31:    $Q_{rest} \leftarrow SchedulingInRing(M^*, Q)$;                ▷ Algorithm 11
32:    return $Q_{rest}$;              ▷ Now, $Q_{rest}$ consists un-handled $\text{Req}_s(X_i, \mathcal{K})$
33: end if
```

4.6.2.1 $SchedulingInRing(M, Q)$ Algorithm

Let $M_0, \ldots, M_s$ be the nodes inside K, if K is a cluster of nodes. In the Ring, M_i can communicate directly with M_{i-1} and M_{i+1} where the index calculations are performed such that M_0 and M_s are connected. The nodes M_{i-1} and M_{i+1}

are called the *left neighbor*, $left(M_i)$, and *right neighbor*, $right(M_i)$, of M_i respectively.

For each node M, let $F(M)$ denote its free capacity. A node M_i can handle a batch Q, if $D(Q) < F(M_i)$ then the process terminates, otherwise the greedy algorithm [41] splits Q into two sets based on their demand. One, Q_0 which M_i keeps itself, one Q_L which it passes left to M_{i-1}. This is done such that Q_0 can be handled in M_i (*i.e.*, $D(Q_0) < F(M_i)$). Note that any of the two sets Q_0 and Q_L can be empty. The pseudo code is shown in algorithm 11.

In general, a node which is receiving a set Q_L from its right neighbor keeps as many storage requests it can handle, and sends the rest on to its left neighbor. If the process returns to the starting node M_i in the ring due to the ring cannot handle all Q, then the starting node returns the rest of the Q. The fact that the starting node is reached can be recognized by flags. To this end before receiving Q all nodes in ring have the flag set to "*new*". When a node receives Q, its flag is set as "*visited*". When the process terminates the flags are reset to "*new*", so all nodes have "*new*" flag for the next schedule. The CM can take care of rest of the Q by running Algorithm 10. While *SchedulingInRing*(Q, M) is running in a ring the next iteration's scheduling will not go into same ring or wait until the process completes in that ring.

Algorithm 11 Load scheduling in a basic consumer - *SchedulingInRing*(M, Q)

```
1: Let $Q = (\mathrm{Req}_s(X_i, \mathcal{K}), i = 0, \dots, \ell)$ be a batch of storage requests;
2: Let $M_0, \dots M_s$ be the nodes in $\mathcal{K}$,;
3: Every node $M$ has a flag f(M) which can be "visited" or "new";
4: Initially for all M $f(M)$ = "new";
5: Select some $M$ (for instance, the one with max $F(M)$);
6: SchedulingInRing$(M, Q)$;
7: ◁ Procedure: SchedulingInRing$(M, Q)$ ▷
8: if $f(M)$ = "visited" then
9:     reset all flags;
10:    return $Q$;
11: else
12:    Split $Q = Q_0 \cup Q_L$ such that: ;
13:       $D(Q_0) \leq F(M)$ and
14:    keep $Q_0$ at $M$;
15:    call SchedulingInRing$(left(M), Q_L)$;    ▷ Procedure: SchedulingInRing$(M, Q)$
16:    $f(M) \leftarrow$ "visited";
17: end if
```

4.6.3 Greed Scheduling/Load Balancing

Load Balancing is an approach to distribute a workload across multiple nodes in a basic consumer or consumers in a DC to achieve resource utilization, maximize throughput, and avoid overload than average load of all consumers inside a DC. Load balancing approaches depend on the system topology [42], i.e., status of the nodes since their parameters change dynamically, so the system status changes frequently. The initial load balancing in the system is usually done at the time of a load scheduling, such as the load scheduling use load balancing approach before schedule the incoming storage requests. The primary aim of load scheduling is to achieve high performance of the system storage by

- reducing re-schedules of storage requests by finding greedy consumers in a DC to store complete video streams,
- balance the consumers capacities inside a DC and utilize all consumers capacities instead of putting some of them in idle.
- allowing video streams to store continuously on scheduled nodes without interruption.

The key feature of load scheduling is its ability to quickly and accurately determine heterogeneous consumers capacity in a DC, and their tolerated storage requests demand.

Greedy consumer selection
The load scheduler attempts to schedule a batch Q to consumers inside a DC $\mathcal{K}$ instead of entire system [43]. The consumers can handle the Q demand and are selected greedily. The selection of consumers is based on two metrics,

$G = 1/k \sum_{i=0}^{k} F(\mathcal{C}_i)$ and
$G_{F(\mathcal{C}_i)} = F(\mathcal{C}_i) - G, i = 1, \ldots, k,$
where G is the arithmetic mean of all consumers capacity inside DC $\mathcal{K}$ and $G_{F(\mathcal{C}_i)}$ is a greedy consumer capacity score which can tolerate the Q demand.

For tolerable consumers, $G_{F(\mathcal{C}_i)}$ is always a non-negative number. This implies that there is no negative impact on incoming batch and consumers capacities, so the consumers are perfectly utilized. In practice and to avoid re-scheduling, a good selection of tolerable consumers has large positive $G_{F(\mathcal{C}_i)}$ scores. Negative $G_{F(\mathcal{C}_i)}$ score should be avoided by sorting consumers based on their $G_{F(\mathcal{C}_i)}$ scores since it implies that the selection has poor consumers which cannot handle Q. The load scheduler filters all negative $G_{F(\mathcal{C}_i)}$ scores of consumers. As we filter out tolerable consumers by $G_{F(\mathcal{C}_i)}$ positive score, the load scheduler utilize all consumers capacities at some point and does load balance. At the same time, it is also possible that at some point greedy scheduler selects no single consumer. We handle this situation with sending the batch to immediately above DC of DC $\mathcal{K}$. The process runs concurrently.

Let consider storage requests in descending order of starting time in the Q. One of the possibility to select Q_i from Q is to get the request in FIFO order and another possibility is to select the maximum no.of requests which sum is less than the capacity of consumer $\mathcal{C}_i$.

The psuedocode is described in algorithm 12.

Inside a basic consumer, the load scheduler can use the same Algorithm 11. After scheduling the Q, the real video streams are storing on scheduled nodes of some M_i in basic consumer. After some period the node M_i can full with video streams and there is no more storage space to get the continuous stream. In that situation, we re-run the Algorithm 11 inside basic consumer to re-direct the video streams of the node M_i to neighbor nodes. If all nodes of the basic consumer are full then we re-run the scheduling inside $\mathcal{K}$.

Algorithm 12 Scheduling in a DC - *GreedyScheduling*$(\mathcal{K}, Q)$

```
1: ◁ Require ▷
2: Let $Q = (\mathrm{Req}_s(X_i, \mathcal{K}), i = 0, \ldots, \ell)$ be a batch of storage requests are
   non-decreasing starting time;
3: Compute $D(Q)$;
4:
5: ◁ Procedure ▷
6: if $\mathcal{K}$ is DC then
7:     Let $\mathcal{C}_0, \ldots \mathcal{C}_k$ be the consumers inside $\mathcal{K}$
8:     for $i = 0, \ldots, k$ do
9:         Compute $F(\mathcal{C}_i)$;
10:    end for
11:    Compute $G = 1/k \sum_{i=0}^{k} F(\mathcal{C}_i)$
12:    for $i = 0, \ldots, k$ do
13:        Compute $G_{F(\mathcal{C}_i)} = F(\mathcal{C}_i) - G$;
14:    end for
15:    Sort consumers and filter negative scores according to $G_{F(\mathcal{C}_i)}$;
16:    Re-index such that $G_{F(\mathcal{C}_0)} \geq \ldots \geq G_{F(\mathcal{C}_k)}$;
17:    $i = 0$
18:    while $(Q \neq \emptyset$ and $i \leq k)$ do
19:        Let $Q_i = 0$;
20:        for $j = 0, \ldots, \ell$ do
21:            if $D(\mathrm{Req}_s(X_j, \mathcal{K}) \leq F(\mathcal{C}_i)$ then
22:                $Q_i \leftarrow \mathrm{Req}_s(X_j, \mathcal{K})$;
23:            end if
24:        end for
25:        $Q_i \subseteq Q$ such that $D(Q_i) \leq \lambda F(\mathcal{C}_i)$
26:        $Q_{return} \leftarrow assignment(\mathcal{C}_i, Q_i)$;                ▷ Algorithm 12;
27:        $Q \leftarrow (Q \setminus Q_i) \cup Q_{return}$;
28:        $i = i + 1$;
29:    end while
30:    if $Q \neq \emptyset$ then
31:        return $Q$;          ▷ Now, $Q$ consists un-handled $\mathrm{Req}_s(X_i, \mathcal{K})$
32:    end if
33: else($\mathcal{K}$ is a cluster of nodes)
34:    for all nodes $M$ in $\mathcal{K}$ do
35:        Compute $F(M)$;
36:    end for
37:    Let $M^*$ be the node with max $F(M)$
38:    $Q_{rest} \leftarrow assignment_b(M^*, Q)$;                ▷ Algorithm 11
39:    return $Q_{rest}$;          ▷ Now, $Q_{rest}$ consists un-handled $\mathrm{Req}_s(X_i, \mathcal{K})$
40: end if
```

CHAPTER 5

Simulation

5.1 Motivation

The performance of the scheduling algorithms depends on the underlying architecture and the system parameters. Traditionally, the performance of distributed systems can evaluate by different models, i.e., analytic modeling, experimental measurements, and simulation modeling. Analytical techniques use to evaluate multiprocessor system performance. However, the complexity of multiprocessor systems limits the applicability of these techniques [44]. Experimental measurement is another alternative technique for parallel and distributed system performance analysis. Measurements can be gathered on existing systems by means of benchmark applications that stress specific aspects of the multiprocessor system [44]. The simulation model is the most straightforward way to evaluate scheduling algorithms without building a full-scale implementation. Simulation models can help to determine performance bottlenecks inherent in the architecture and provide a basis for refining the system configuration [44].

Distributed simulation systems can provide substantial benefits in several ways:

- Simulations useful in very fast executions since some simulations results are produced in seconds to make important decisions [45].

- Simulations used for virtual environments must execute in real time [46].
- Distributed simulation techniques can be used to create virtual environments that are geographically distributed, which have benefits in terms of convenience and reduced travel costs [44].
- Distributed simulation can simplify integrating simulators that execute on heterogeneous nodes [46].
- Another potential benefit of utilizing multiple nodes is increased tolerance to failures [46]. If one node fails in a cluster, it may be possible for other nodes to continue the simulation provided critical elements do not reside on the failed nodes.

Therefore, the MVaaS system selects the simulation to test the performance of scheduling algorithms and for analysis of it underlying the described MVaaS architecture in Section 3.2 and its parameters in Section 3.6.

5.2 Simulator design

We run a simulation to evaluate scheduling algorithms results on proposed architecture. The simulation is using different batches and their various demands, various topology sizes in basic consumers, and different consumers parameters settings. The simulation simulates a single DC $\mathcal{K}$. The $\mathcal{K}$ can have any number of consumers (either DCs or Basic consumers or both). All nodes inside basic consumers have a single non-negative number, specifying the capacities and all nodes have their original storage spaces. The design of a simulator is promising to examine the performance of scheduling algorithms.

To run simulation, we built a simulator which can use dynamic system settings. For instance,

```
"DC,0,1;(DC,1,100;(BCR,5,8;BCR,6,8;DC,7,100;(BCR,13,12;BCR,14,10;)
BCR,8,10;)DC,2,100;(BCR,9,5;BCR,10,5;)BCR,3,8;DC,4,100;(DC,11,100;
(BCR,15,8;BCR,16,4;)DC,12,100;(BCR,17,10;)))"
```

where the structure describes 3 arguments like

```
For a DC: "Name of a DC, Identification of the DC, No. of producers associated
with the DC;".
```

For a Basic consumer (BCR): "Name of a BCR, Identification of the BCR, No. of nodes in BCR;".

The instance of the system (DC $\mathcal{K}$) presented here has 6 consumers (DCs), and 11 basic consumers similar to shown in Figure 3.1. The basic consumers have heterogeneous capacities, i.e., the units arranged in hierarchies with different topologies. The number of producers associated with each DC is varied. We assume that each node has at least 14500 capacity as a number at the starting of the simulation. At the moment, we are assuming that each node has the same number of storage space. The simulation of the proposed system is carried out in Sun Java 8 on a 64-bit Windows machine has an Intel Core i7-3520M with 16GB Ram and 2.9GHz speed. The simulation runs entirely on a single core.

For the first example, we assume that the storage requests are generating by an external source which is implemented in SimDriver class in the simulator. Each producer generates storage requests regularly, and most of them are associated with video streams of that producer. The producers which are associated with DC $\mathcal{K}$ can send requests to unique CM of DC $\mathcal{K}$. The CM of $\mathcal{K}$ can establish a connection with higher-level DC of DC $\mathcal{K}$.

For test cases, we run the simulation with different input granularities, for instance, batches with low and high demand as compared to the capacities of consumers inside $\mathcal{K}$. The load scheduling algorithm schedules the batches and returns un-handled batches, if any and re-schedule them in higher-level DCs. We give a slack margin λ between 0 and 1 for each scheduling decision.

The implemented simulator code has been uploaded to GitHub [1] and available on it with limited access.

5.2.1 Implemented java classes in simulator

1. SimDriver
   The SimDriver class has the main(() method, so this class is the starting point of whole simulation. This class starts to build the system structure, starts initial scheduling and then starts actual simulation. In real time, SimDriver could be an external source to generate storage requests with their demands, i.e., from this class, we can order to generate storage requests from any source, including producers, consumers, users, etc.

   Initial scheduling is triggered from Simdriver where the storage requests are generated and passing to CenterManager class.

[1] `https://github.com/karuncs/MVaaS_Simulator.git`

2. SystemBuilder
   The whole system can build based on given constants (Constants class) in this class. This class can identify a DC's children type like either a DC or a Basic consumer(BCR). The SystemBuilder can add children to DCs and can build dynamic system which may have any no.of DCs, and BCRs, for the reference you can see the simulator design instance5.2. The UML diagram of a complete dynamic system structure is depicted in Figure 5.1.

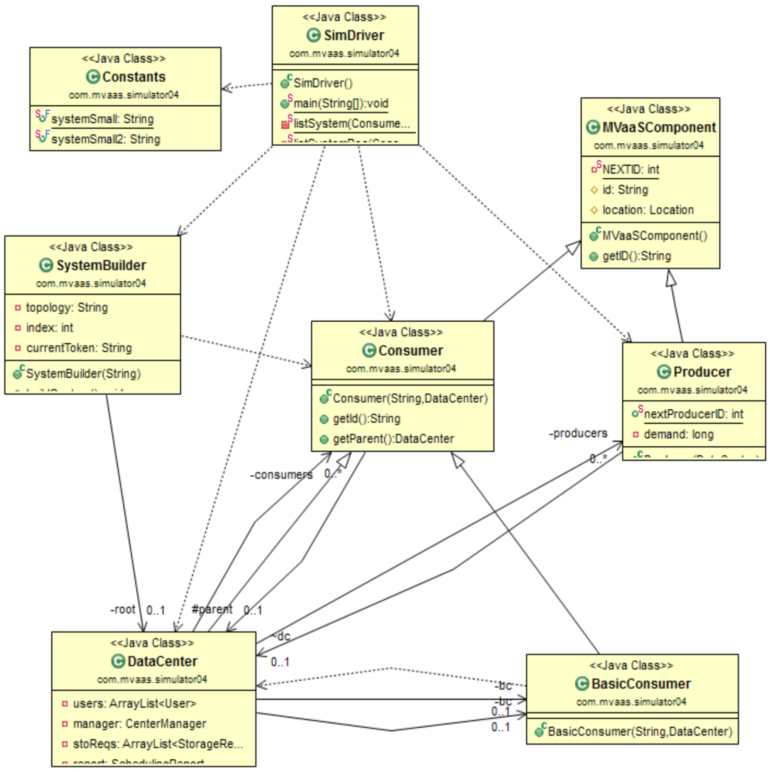

Figure 5.1: The UML diagram of a complete dynamic system structure

3. Constants
   In the Constants class, we can give the fixed number of DCs, associated producers of DCs, BCRs and their topologies for system design. Simulation duration is also can be decided here. This class can maintain slack margin for scheduling decisions, maintains nodes capacity and storage space before simulation starts.

4. CenterManager
   The Center manager (CM) is the heart of a DC, i.e., every DC has owned

CM to manage all tasks of the DC, for instance, executes a scheduling algorithm to schedule storage requests and assign a video stream to node and observe the stream until it is stored.

This class collects storage requests and schedules them through SchedulingAlgo class, i.e., it executes the scheduling algorithm. The CM maintains a scheduling report which has routing information of scheduled storage requests and un-handled storage requests. For the scheduled storage requests, it runs connection request and for the un-handled requests it runs re-assignment request which can be sent to higher-level CM of DC for re-scheduling. This class can handle all types of the requests, including storage requests, connection requests, assignment requests, re-assignment requests, health check request.

5. MVaaSComponent
   A component could be a producer or a consumer. Every component has Unique identification. This class is the Super class of Producer and Consumer classes.

   (a) Producer
   A Producer class maintains the same id which it got from Super class. Every producer is associated with a unique DC at once. The DC id is maintained in Super class and DataCenter class.

   (b) Consumer
   A consumer can be a DC or BCR, so this class maintains the ids of both of them where it got from Super class (MVaaSComponent)

   - DataCenter
     A DataCenter class has the following elements: DCs or BCRs or both, Center manager, associated producers and users. This class can create/add producers, DCs, BCRs in DCs, and users. It can also calculates capacity of each DC, which can use SchedulingAlgo class.
   - BasicConsumer
     BCR does not have a CM. It is a real storage facility in the system. This class maintains same unique id which it got from Super class (BasicConsumer <– Consumer <– MVaaSComponent).
   - BasicConsumerRing
     A BCR can form any topology. Current system considers it formed as a Ring, so all the nodes can form in a Ring. In the Ring, each node can communicate with both sides of immediate neighbours, so this class defines neighbours and start node of the Ring for scheduling. The start node always changes its position since node capacity calculations are performed for max capacity node in the Ring and total capacity of all nodes of the Ring.

6. Node
   A node consumes a real video stream. To get the video stream from producer, a connection is established between a producer and a node by the corresponding CM. Every Node maintains its id and list of video streams, which are consuming. The Node class implements Runnable threads, so it has run method. The run method works like daemon. This class can print streaming data in local computer for testing a purpose. The Node class maintains each node total and consumed storage space.

   If a node has no storage space to consume streams, then the node stops consuming and throws a message like 'Node has no storage space'. In this case the streams have to be redirected to its neighbours.i.e., it calls the scheduling algorithm inside BCR. The Node class is a Super class of RingNode class.

   (a) RingNode
   It gives neighbours of each node information to detect each node free storage space.
   (b) NodeElements
   This class calculates all parameters of nodes, including each node capacity and storage space.

7. Stream
   This is the real video stream which can be produced by a producer and assigned to a scheduled node. This class has video stream's spead and its quality variables.

8. Location
   It gives the location of producers and consumers.

9. Routing
   This class has routing information of a node which can be consumed stream. This node identified by a scheduling algorithm.

10. Scheduling
    Three different scheduling algorithms are scheduling batches as of their own styles which we described in 4.6.1.1, 4.6.2 and 4.6.3

    - SchedulingAlgo
      The CM of each DC executes the scheduling algorithm to schedule a collected batch of storage requests. A batch can have a single storage request. This class can decide where the storage requests can be scheduled like on a DC or inside a BCR. The scheduling works for both cases, for instance, scheduling on DC with schedulingOnDC() method and scheduling inside a BCR with loadSchedulingInRing() method. This class throws scheduling report with handled batches and un-handled batches, if any.

The CM class sorts the consumers with HashMap with their capacities for scheduling.

- PrportionalScheduling
  This class can work like SchedulingAlgo but its own style of scheduling as described in 4.6.2.
- GreedyScheduling
  This class can work like SchedulingAlgo but its own style of scheduling as described in 4.6.3.

11. SchedulingReport
    It comes from scheduling algorithm and contains the unscheduled and scheduled storage requests information.

12. ReportInfo
    This class maintains scheduled storage requests information, which can generate by the scheduling algorithm to assign video stream, i.e, connection request variables are available here.

13. User
    From this class user can retrieve video data.

14. Request
    As we said MVaaS system itself a request driven system, so all following requests classes use to maintain specific tasks. The Request class is the Super class of all types of request classes. This class generates id for each request in the system.

    (a) AssignmentRequest
    This class assigns a component to a DC. The component could be a producer or consumer or user or external source. This class always maintains two parameters MVaaSComponent and DataCenter.

    (b) StorageRequest
    This class activates by SimDriver class to generate storage requests. This class has two parameters MVaaSComponent and DataCenter.

    (c) ConnectionRequest
    It connects a producer and a node in a basic consumer of a DC based on scheduling report. This class always maintains four parameters to route the stream to scheduled node from associated producer of the DC. The parameters are MVaaSComponent, Consumer, Stream, and Routing.

    (d) ReAssignmentRequest
    This class re-assigns a component to higher or parent DC of a current DC, if the current DC is not able to handle storage requests.

(e) HealthCheckRequest
It checks the health status of all components in a DC and in the system.

15. FailuresDetection
This class can detect failures of the system components continuously through the HealthCheckRequest class. The system components could be producers, and consumers. This class is partially implemented since it's part of fault-tolerance.

5.2.2 Simulation Events

In the simulation, events can fire internally in the system. The events can trigger various components status messages, for instance, if a consumer is full, then the system can print that the consumer is full. If the whole system is full and had no more capacity to receive any storage requests, then the system fires an event to print that the 'Whole system is full'. The producers can also fail at the time of the system building , however, we try to fail other components randomly to make events for further development.

5.3 Performance Metrics

5.4 Evaluation of Results

CHAPTER 6

Glossary

BigTable

Sharding

Cluster

Producer

Consumer

Key-value pairs : It is the most common data model for storage of video data. It has three subcategories:

Node or Server

Transactional application

Hashing

Consistency hashing

Master-slave

Master-less

Fail-over

Gossip protocol

Bibliography

[1] B. Veeravalli and Z. Zeng, “Large-scale object-based multimedia storage systems,” in *Encyclopedia of Multimedia*, B. Furht, Ed., 2008.

[2] I. Riakotakis, F. Ciorba, T. Andronikos, and G. PapaKonstantinou, “Self-adapting scheduling for tasks with dependencies in stochastic environments,” in *Cluster Computing, 2006 IEEE International Conference on*, 2006, pp. 1–8.

[3] C. Kruskal and A. Weiss, “Allocating independent subtasks on parallel processors,” *Software Engineering, IEEE Transactions on*, vol. SE-11, no. 10, pp. 1001–1016, Oct 1985.

[4] C. D. Polychronopoulos and D. J. Kuck, “Guided self-scheduling: A practical scheduling scheme for parallel supercomputers,” *IEEE Trans. Comput.*, vol. 36, no. 12, pp. 1425–1439, Dec. 1987.

[5] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, Aug. 2001. [Online]. Available: http://doi.acm.org/10.1145/964723.383071

[6] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, “Ivy: A read/write peer-to-peer file system,” 2002, pp. 31–44.

[7] J. Zhao, H. Wang, J. Dong, and S. Cheng, “A reliable and high-performance distributed storage system for p2p-vod service,” in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, 2012, pp. 614–617.

[8] T. Klumpp, “File sharing, network architecture, and copyright enforcement: An overview,” *Managerial and Decision Economics*.

[9] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, “An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 14, no. 3, pp. 236–247, 2003.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’03. New York, NY, USA: ACM, 2003, pp. 29–43.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[12] A. Lakshman and P. Malik, “Cassandra: A decentralized structured storage system,” *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[13] M. Vora, “Hadoop-hbase for large-scale data,” in *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*, vol. 1, Dec 2011, pp. 601–605.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, “Oceanstore: An architecture for global-scale persistent storage,” *SIGPLAN Not.*, vol. 35, no. 11, pp. 190–201, Nov. 2000.

[15] M. Satyanarayanan, “The evolution of coda,” *ACM Trans. Comput. Syst.*, vol. 20, no. 2, pp. 85–124, May 2002.

[16] C. T. Timothy, T. Mann, and E. K. Lee, “Frangipani: A scalable distributed file system,” in *In Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997, pp. 224–237.

[17] I. Riakiotakis, F. M. Ciorba, T. Andronikos, and G. Papakonstantinou, “Distributed dynamic load balancing for pipelined computations on heterogeneous systems,” *Parallel Comput.*, vol. 37, no. 10-11, pp. 713–729, Oct. 2011.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[19] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, “Availability in globally distributed storage systems,” in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[20] C.-F. Chan and S.-H. Chan, “Distributed hash tables: Design and applications,” in *Handbook of Peer-to-Peer Networking*, X. Shen, H. Yu, J. Buford, and M. Akon, Eds. Springer US, 2010, pp. 257–280. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-09751-0_10

[21] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska, “Building a database on s3,” in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’08, 2008, pp. 251–264.

[22] S. Sivasubramanian, “Amazon dynamodb: A seamlessly scalable non-relational database service,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’12, 2012, pp. 729–730.

[23] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[24] S.-H. G. Chan and F. A. Tobagi, “Modeling and dimensioning hierarchical storage systems for low-delay video services,” *IEEE Trans. Comput.*, vol. 52, no. 7, pp. 907–919, Jul. 2003.

[25] D. Feitelson and L. Rudolph, “Distributed hierarchical control for parallel processing,” *Computer*, vol. 23, no. 5, pp. 65–77, May 1990.

[26] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal, “A simple load balancing scheme for task allocation in parallel machines,” in *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA ’91, 1991, pp. 237–245.

[27] D. Abts and J. Kim, *High Performance Datacenter Networks: Architectures, Algorithms, and Opportunities*, ser. Synthesis lectures in computer architecture. Morgan & Claypool Publishers, 2011. [Online]. Available: http://books.google.dk/books?id=1_TUzIaKkdsC

[28] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web,” in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC ’97. New York, NY, USA: ACM, 1997, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/258533.258660

[29] D. S. Lee and J. L. Kalb, “Network topology analysis.”

[30] S. Das, F. Sarkar, and M. Pinotti, “Distributed priority queues on hypercube architectures,” in *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, May 1996, pp. 620–627.

[31] R. Jayaraman, D. o. E. SRC-CMU Center for Computer-Aided Design, and C.-M. U. Computer Engineering, *Floorplanning by Annealing on a Hypercube Architecture*, ser. Research Report. Carnegie-Mellon University. CMUCAD. Pittsburgh, Pa., 1987. [Online]. Available: http://books.google.dk/books?id=3HWVYgEACAAJ

[32] E. I. Cohen, G. M. King, and J. T. Brady, "Storage hierarchies," *IBM Syst. J.*, vol. 28, no. 1, pp. 62–76, Mar. 1989.

[33] M. Comuzzi, C. Kotsokalis, C. Rathfelder, W. Theilmann, U. Winkler, and G. Zacco, "A framework for multi-level sla management," in *Proceedings of the 2009 International Conference on Service-oriented Computing*, ser. ICSOC/ServiceWave'09, 2009, pp. 187–196.

[34] "A reference architecture for multi-level sla management," in *Service Level Agreements for Cloud Computing*, P. Wieder, J. M. Butler, W. Theilmann, and R. Yahyapour, Eds., 2011.

[35] B. Hu, K. L. Yeung, and Z. Zhang, "An efficient single-iteration single-bit request scheduling algorithm for input-queued switches," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 187 – 194, 2013.

[36] Y. Wang and A. Merchant, "Proportional-share scheduling for distributed storage systems," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, ser. FAST '07, 2007, pp. 4–4.

[37] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz, "Maintenance-free global data storage," *IEEE Internet Computing*, vol. 5, no. 5, pp. 40–49, Sep. 2001. [Online]. Available: http://dx.doi.org/10.1109/4236.957894

[38] G. Zheng, E. Meneses, A. Bhatele, and L. V. Kale, "Hierarchical load balancing for charm++ applications on large supercomputers," in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ser. ICPPW '10, 2010, pp. 436–444.

[39] M. Lo and S. P. Dandamudi, "Performance of hierarchical load sharing in heterogeneous distributed systems," in *International conference on parallel and distributed computing systems*, 1996, pp. 370–377.

[40] K. Xiong and H. Perros, "Service performance and analysis in cloud computing," in *Services - I, 2009 World Conference on*, July 2009, pp. 693–700.

[41] "On generalized greedy splitting algorithms for multiway partition problems," *Discrete Applied Mathematics*, vol. 143, no. 1–3, pp. 130 – 143, 2004.

[42] N. Sran and N. Kaur, “Article: Comparative analysis of existing dynamic load balancing techniques,” *International Journal of Computer Applications*, vol. 70, no. 26, pp. 25–29, May 2013, full text available.

[43] C. Delimitrou and C. Kozyrakis, “Paragon: Qos-aware scheduling for heterogeneous datacenters,” *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 77–88, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2490301.2451125

[44] H. D. Karatza, “Applied system simulation,” M. S. Obaidat and G. I. Papadimitriou, Eds., 2003, ch. Simulation of Parallel and Distributed Systems Scheduling, pp. 61–80.

[45] A. Law and W. Kelton, *Simulation modeling and analysis*, ser. McGraw-Hill series in industrial engineering and management science.

[46] R. M. Fujimoto, *Parallel and Distribution Simulation Systems*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1999.