UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

PROIECT DE DIPLOMĂ

Distribuția în timp real a datelor de stare pentru procesarea Online a experimentului ALICE

Daniel-Florin Dosaru

Coordonatori științifici:

Sl. Dr. Ing. Mihai Carabaș
Ing. Costin Grigoraș
Prof. Dr. Ing. Nicolae Țăpuș

BUCUREȘTI
2019

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE DEPARTMENT

DIPLOMA PROJECT

Real-time conditions data distribution for the Online data processing of the ALICE experiment

Daniel-Florin Dosaru

Thesis advisors:

Sl. Dr. Ing. Mihai Carabaș
Ing. Costin Grigoraș
Prof. Dr. Ing. Nicolae Țăpuș

BUCHAREST
2019

CONTENTS

SINOPSIS

ALICE (A Large Ion Collider Experiment) este un detector de ioni grei al acceleratorului LHC (Large Hadron Collider) de la CERN, Geneva. A fost proiectat să studieze fizica interacțiunilor tari și a energiei dense extreme unde o formă a materiei numită quark-gluon se formează. Noua facilitate ALICE de reconstrucție sincronă a datelor pentru pentru etapa 3 are nevoie de un sistem de distribuție în timp real al datelor ce descriu condițiile experimentului și al celor de calibrare. Obiectele ce conțin aceste date sunt produse cu o frecvență de până la 50Hz și trebuie să fie transmise la aproximativ 1500 de servere pentru a implementa o buclă de feedback pentru reconstrucția în timp real (online). Pentru o distribuție eficientă în acest mediu soluția propusă de această lucrare folosește un mecanism multicast de expedierea datelor. În plus, sistemul se bazează pe un serviciu de caching distribuit pe toate cele 1500 de servere ce recepționează și reține cea mai recentă versiune a obiectelor în memorie, făcându-le disponibile pentru procesele locale printr-un API de tip REST. Acest API este de asemenea implementat de repository-ul central de obiecte și de instanțele de dezvoltare, permițând conexiuni transparente de recuperare și acces al datelor de stare de la toate joburile ce rulează în infrastructura Grid. În această lucrare vom prezenta detaliile noului framework pentru distribuția datelor de stare și cum am reușit să obținem un sistem sigur de distribuție a datelor bazat pe mecanisme de transport nesigure.

ABSTRACT

ALICE (A Large Ion Collider Experiment) is a heavy-ion detector on the Large Hadron Collider (LHC) ring at CERN, Geneva. It is designed to study the physics of strongly interacting matter at extreme energy densities, where a phase of matter called quark-gluon plasma forms. The new ALICE synchronous data reconstruction facility for Run 3 needs a real-time conditions and calibration data distribution mechanism. New calibration and conditions data objects are produced at up to 50Hz and have to be propagated to about 1500 servers as to implement a feedback loop for the online data reconstruction. For efficient data distribution in this environment, the designed solution by this thesis uses a network multicast delivery mechanism. In addition, the system relies on caching services distributed on all of 1500 servers to receive and keep the most recent object versions in memory, making them available to the localhost running processes via a REST API. The REST API is implemented also by the central object repository and development instances, allowing for transparent connection fallback and access to the conditions data from all jobs running on the Grid infrastructure. In this thesis I show the details of the new experiment conditions data framework and how I managed to have a reliable delivery system based on inherently unreliable transport mechanisms.

Acknowledgments

I would like to express my very great appreciation to SI. Dr. Ing. Mihai Carabaș for his valuable and constructive suggestions during the planning and development of this research work. His positive outlook and confidence in my research inspired me and gave me confidence. His careful editing contributed enormously to the production of this thesis.

I would also like to express my profound appreciation for all my bachelor professors. Without their help this thesis would not be possible. In particular, I wish to thank to Prof. Dr. Nicolae Țăpuș for helping me to choose this project in the first place. I was honored to have such a dedicated professor during my period as a bachelor student at Politehnica University of Bucharest. In addition, thanks to his work over the years, our University has a strong partnership with CERN, the organization for which this work is done.

Finally, I would like to thank my supervisor, Costin Grigoraș (Java Engineer for ALICE@CERN) for the patient guidance, encouragement and advice he has provided throughout my time as a student. I would also like to thank him for being a nice host during my visit at CERN. I have been extremely lucky to have a supervisor who cared so much about my work, and who responded to my questions and queries so promptly.

1 INTRODUCTION

CERN **C**onseil **E**uropéen pour la **R**echerche **N**ucléaire is an international organization that has the largest particle acceleration in the world today. The Large Hadron Collider has a 27 km ring of superconducting elements with accelerating structures to increase the energy of the particles [1]. Physicists around the world can rely on CERN infrastructure to study the basic constituents of matter and high energy physics. The beams inside the LHC are designed to collide at certain positions around the accelerator ring, corresponding to the location of four particle detectors – ATLAS [2], CMS [3], ALICE [4] and LHCb [5].

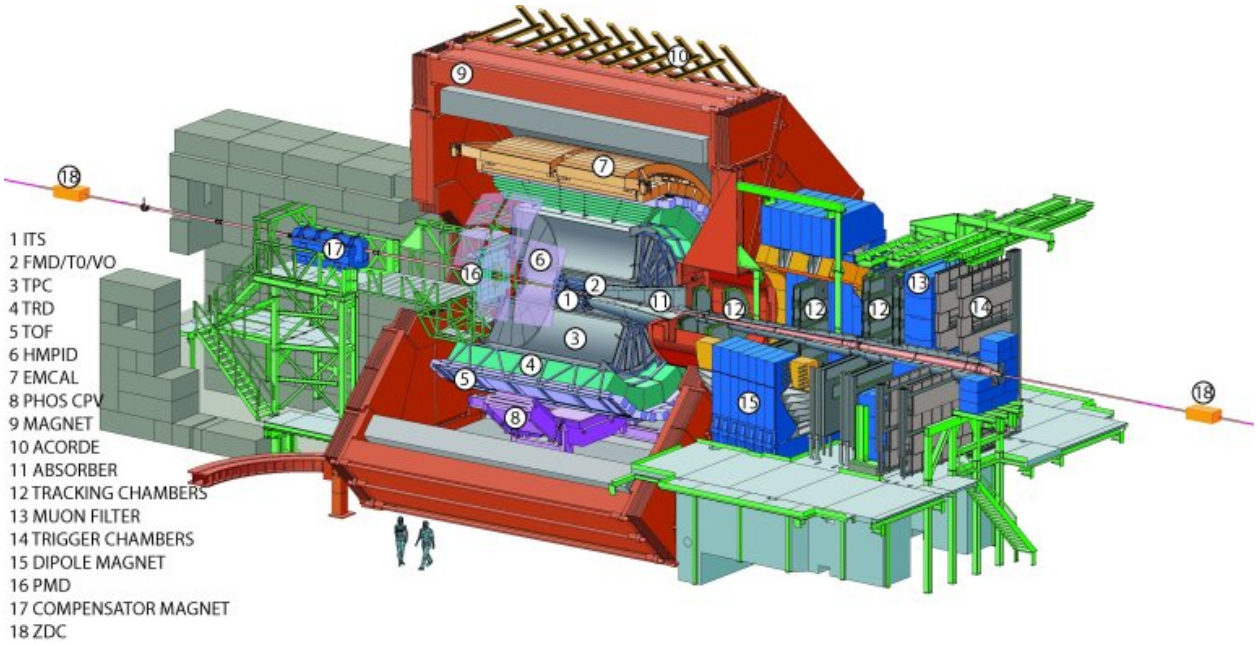

Figure 1: ALICE detector components

ALICE (A Large Ion Collider Experiment) [4] is a heavy-ion detector on the Large Hadron Collider (LHC) ring. It is designed to study the physics of strongly interacting matter at extreme energy densities, where a phase of matter called quark-gluon plasma forms. ALICE infrastructure consist in multiple layers of detectors as shown in Figure 1 [1] . Every individual detector can be considered a pixel matrix. It can be simply described as an advanced cylindrical camera sensor. This system can take 'pictures' at a rate up to 50KHz (with the help of the Long Shutdown 2 upgrade). Scientists are using the reconstruction software to identify and calculated trajectories for the interesting particles in order to study the physics of strongly

[1]Figure 1 is taken from http://aliceinfo.cern.ch/Public [6]

interacting matter at extreme energy densities. Different unique events where made possible until now: for example in 2009 ALICE made the first observation of proton-proton collisions at the LHC [7]; another example is the first observation of proton-lead ion collisions made in 2013, see Figure 2 [2].

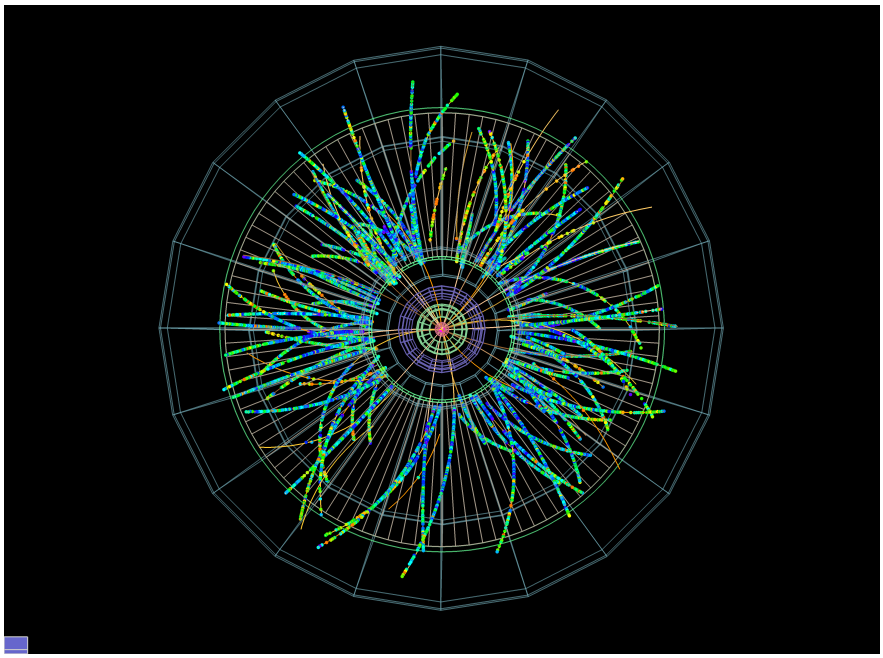

Figure 2: ALICE records first proton-lead collisions at the LHC [8]

1.1 Motivation

ALICE experiment at CERN will change its operation mode in **Run 3** [9] (starting from 2021) from trigger-based to continuous data readout. In terms of data acquisition this translates in a 100x increase in data rates (up to 1.1TB/s) which has very large implications on both the hardware and the software used by the experiment. One important aspect is that data cannot be stored at these rates so it has to be reconstructed on the fly. A cluster of 1500 servers will handle the data stream and the reconstruction process and will need, in addition to the experiment data, **real-time access to the calibration and conditions data**. Conditions data (eg. magnetic field intensity, current levels, operating temperatures etc.) are generated with a frequency of, at most, 50Hz for one parameter and has to be quickly propagated (in the same O(20ms) time frame) to all nodes so that they have access to the latest information when they do the reconstruction. Moreover this data has to be stored persistently to allow for later processing of the same data and bootstrapping nodes with the current detector status when they come online.

Figure 3 [3] shows the possible hardware pipeline with the estimated number of processing nodes

[2] Figure 2 is taken from [8]

[3] Figure 3 is taken from the ALICE technical Design report [9]

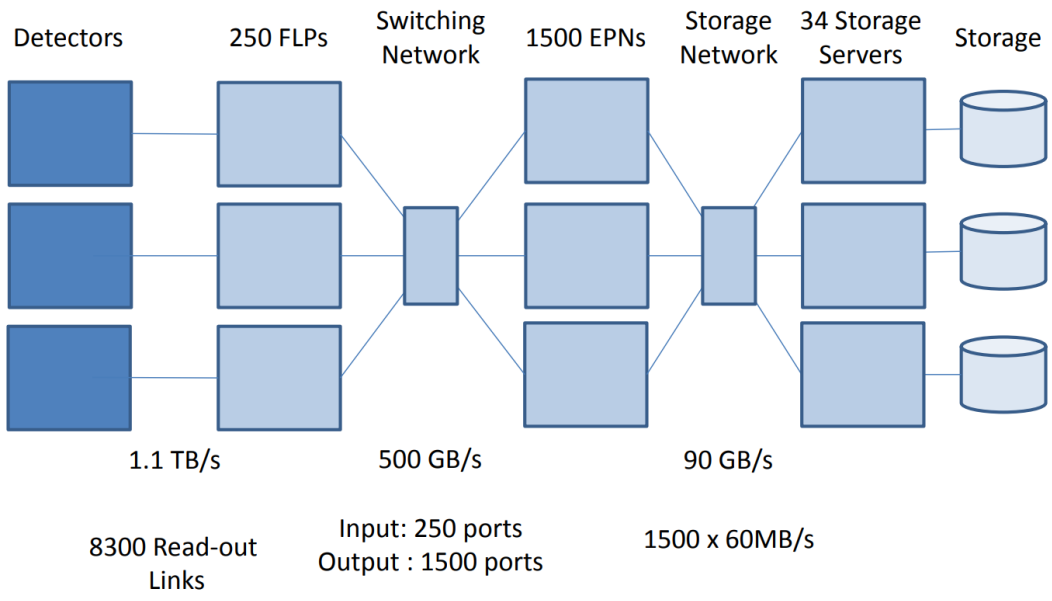

Figure 3: Possible hardware implementation of the logical data flow, as shown by [9]

and throughput at the different online stages. Aproximately 250 FLPs (First Level Processors) and 1500 EPNs (Event Processing Node) will be needed to cope with the data processing. The global detector read-out rate is 1.1TB/s over 8300 read-out links.

1.1.1 Bandwidth problem

Simultaneous unicast delivery of the calibration data to each computing node (EPN) will exhaust the capability of even a high bandwidth network with a powerful server. For example sending an object of 10MB in one second to a cluster of 1500 servers will result in a throughput of 120 Gbps. Considering the fact that the new objects are generated 50 times per second the throughput will be 6000 Gbps (6Tbps). Currently this enormous throughput can not be achieved by a single network card.

1.1.2 Reliability problem

Instead of sending a copy of the packet to each node I can use a multicast mechanism, but this comes with other problems related to transfer reliability. Multicast transmission is a one-to-many system. Reliable TCP requires a one-to-one handshake to synchronize counters (sequence numbers) to ensure trustworthy transport. This cannot be done with many receivers as the sender has no way to know how many receivers there are and how many ACKs to expect, or what to do if one of the receivers disappears during the transfer. A good analogy for the multicast mechanism would be a speaker in a room with people. The speaker does not know how many people are supposed to be there, neither care if some of them are late or leave. Neither does he know (or care) if everyone in the hall is hearing everything he says. So the downside of using the multicast transmission is that I rely on inherently unreliable transport

mechanisms (User Datagram Protocol).

1.1.3 Maximum payload size

In addition, the size of a typical object that has to be transmitted is typically 2 MB, but it can reach up to 10 MB. The maximum UDP payload size is 65507 B (0.065507 MB), far less than 2 MB, as determined by the following formula:

$$MaxSize - (L_{IPH} + L_{UDPH}) = 65535 - (20 + 8) = 65507 \quad (1)$$

where:

$MaxSize$ = Maximum value for length field in UDP header (0xFFFF = 65535)
L_{IPH} = IP Header length (without optional fields)
L_{UDPH} = UDP Header length

For this reason, a fragmentation mechanism should be used in order to send a typical size data object (also called in this thesis **blob**).

1.1.4 Current implementation of CCDB

Another important aspect to consider is that the current implementation [10] (CCDB [4] framework) of the authoritative central service is in Java and is using an embedded Tomcat engine to handle the HTTP requests. The solution has to be build on this framework and it has to add the sender and receiver components to it. In addition to this, the project has to have a mechanism to recover the lost objects or fragmented objects and to check their integrity/validity using checksums/expiration timestamps.

ALICE experiment will search for rare physics phenomena in Run 3 with a very small signal to background ratio. In order to do this scientists have to analyze the data at a much higher rate (from trigger-based to continuous data readout) and to keep large statistics of reconstructed events. Increasing the data rates has large implications on both the hardware and the software used by experiment. My motivation is to solve the problems mentioned above and to create a powerful computing tool for fast, efficient and reliable transmission of calibration and conditions data blobs. This project should be integrated in the current framework before Run 3 starts at LHC.

This thesis will presented the implementation of two major components: a multicast sender and a multicast receiver with a caching service and recovery service. Both components will implement the same REST API as the central repository to make the objects available to the localhost running processes via the REST API.

[4] Condition and Calibration Data Base

1.2 Objectives

In this thesis I introduce a particular transmission mechanism that uses multicast techniques for efficient and fast delivery and the implementation of the same REST API as the central repository for data recovery. The proposed solution will address every aspect presented in the section above: first, it will send it in the network where the cluster of the 1500 servers is located in order to proces the date on the fly. Secondly, this system is using multicast messages for efficient usage of the bandwidth. In addition, this thesis will show how I managed to reliably send blobs using a recovery service based on the REST API implemented by the central service and the multicast sender.

The main objective for this project is use reliable multicast distribution in order to provide real-time access to the calibration and conditions data for every event processing node. Furthermore, each node should implement the same REST API as the central service, a recovery service and a caching service for high better performance.

Real-time access to the calibration and conditions data means that every EPN should receive in less than 20ms (worst case scenario as stated by [9] about the Time Frame Size).

1.2.1 Performance objectives

This implementation succeeded to send with a frequency of 50 Hz a new blob with a payload size of 2MB to a group of 3 machines described in Chapter 5. Using a loss percentage between 2% and 10% in a controlled environment the recovery system managed to recover all the lost packets. More details about the testing environment can be found in Chapter 5.

1.3 Summary

The rest of the thesis is organized as follows:

- Chapter 2 provides a brief overview of the current reliable multicast protocols.
- Chapters 3 and 4 present this system architecture and other implementation details like the algorithms
- Chapter 5 contains the testing method and relevant results for this project
- In Chapter 6 I drawn the conclusions and point ideas for future work.

2 RELATED WORK

Although there are different tools that provide a working proof of concept for reliable multicast transmission, there is currently no tool which can integrate both the caching service and the recovery service based on the REST API already implemented in the current framework [10]. As many other multicast protocols mentioned by [11], this paper presents a multicast protocol that has been designed with a specific application in mind, but it can be easily adapted to different kind of applications since the system does not concern about the content of the Blobs. If needed, incomplete or corrupted blobs can be recovered by the application by simply make use of the recovery system in the receiver component. The tools that come close to send reliable multicast transmission are based either on FEC techniques which means adding extra redundant information to the packet, either on retransmissions using ACKs-NACKs schema as a reliability mechanism (according to [11]).

UFTP - Encrypted UDP based FTP with multicast [12] is file transfer program designed for secure and reliable transfer files to multiple receivers. Giving the fact that the conditions and calibration data will travel only in the ALICE's private network and there is no reason to hide the content of the data; it was agreed with the ALICE team that the overhead of encryption and decryption can be avoided to decrease the processing time of the packets.

Other interesting approaches of solving the problem of reliable transmission are using the ACK reduction techniques or a combination of FEC [1] and ARQ [2]. [13] presented how to reduce the acknowledgment traffic by grouping receivers into local regions and generating a single acknowledgment per local region, and reducing end-to-end latency by performing local recovery. [14] defines a hybrid system to reliable multicast, combining requests for retransmission with transmission of redundant data for error correction and quantifies the advantages and disadvantages of this approach in practice. One major drawback of these codes is that encoding and decoding overhead is significant $O(payloadLength^2)$ on a typical data object sizes (average 2 MB, worst case 10 MB).

Jumbograms IPv6 supports Jumbograms [15] that are much larger compared to the 64KB limit of IPv4 UDP. IPv6 has a hop-by-hop option, also called the Jumbo Payload option, that can carry a 32-bit length field. Having this field means that the protocol can theoretically transport payloads between 65,536 and 4,294,967,295 octets in length. This packets are

[1] Forward error correction

[2] Automatic Repeat Request

referred as Jumbograms. For compatibility reasons with IPv4 protocol I chose to implement a fragmentation and reassembling mechanism in order to send payloads greater than 64 KB using UDP packets. Since the hardware used by ALICE for the data distribution is not established yet, I had to design a adaptable solution. My implementation supports the transmission without fragmentation if the maximum payload limit for a fragmented packet is configured to a greater value than the actual complete Blob payload.

Fragmentation: IPv4 versus IPv6 Unlike Ipv4, IPv6 fragmentation is done only by the source node, not by the routers along the way. In contrast, the IPv6 has a minimum MTU [3] of 1280 B according to [16]. This project adaptable solution of sending big data object will be useful in the future if the IPv6 will be supported.

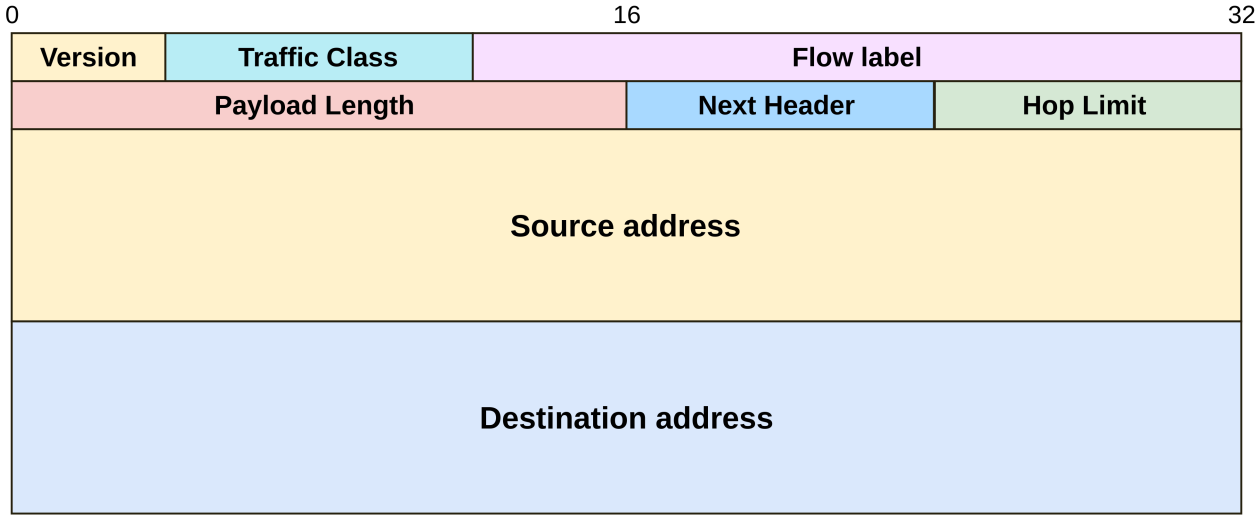

Figure 4: IPv6 Header fields: if the Payload Length field is set to zero than the IPv6 packet can carry a Jumbogram using the Hop-by-Hop Options header

Reliability mechanism This project uses NACKs [4] as a reliability mechanism to recover the lost data and an adaptive repeat transmission method: If the central repository receives a limited number of recovery request (GET request with Range header field active) for a blob it will repeat the transmission of the data request using unicast transmission. If the the number of the request exceeds a certain limit, the central repository will send the data requested using the multicast mechanism.

[3] Maximum Transmission Unit
[4] Negative Acknowledgement

3 DESIGNING THE FRAMEWORK

Overview Real-time access to the calibration and conditions data requires a fast and scalable distribution mechanism. These data are stored in the central repository and changed at about every 20ms (see section 5.2.2. from [9] for more details about the Time Frame Size). For performance reasons the multicast mechanism is better suited for fast and low bandwidth data distribution to a large number of Event Processing Nodes. But this great benefit comes with a major drawback: using UDP as a transport protocol will cause transmission trustworthy issues. Considering the high importance of reliable transmission in the ALICE Online data processing, it was deemed necessary to implement a recovery system. Since the exiting REST API implemented by the central object repository can give access to the condition data from all jobs running on the Grid infrastructure I designed the system recovery based on this API. Further issues related to the maximum payload size were solved using fragmentation in the sender unit and reassembling in the receiver unit.

3.1 CCDB framework

Current framework Conditions and calibration data have two components that have to be transmitted to the processing nodes: data (also named Blob payload) and metadata. A system called CCDB [1] [10] was designed to address two aspects of condition data handling: real-time data propagation to the cluster via network multicast messages (implemented in this project) and persistent storage / query. The second aspect was implement via a combination of Cassandra [17] database (for metadata) and EOS [18] distributed storage (for the actual data). A REST API [10] hides the actual back end from the clients (experiment software) and offers high level object operations (inserting new objects, searching by time, by object type etc). An authoritative central service will have connections to the metadata database and the blob storage while the processing nodes will expose a similar API to retrieve objects from their in-memory cache of last values.

In Figure 5 [2] there are three different main components outlined, following the data flow from the source to the drain. In the upper left of the figure there are mentioned the calibration data source - the LHC, O2 (Online-Offline) processes, High Level Trigger; on the right, the central service parts: Metadata repository, Blob storage and the persistent store service. In the bottom left part of the diagram there are illustrated the receivers with the main features

[1] Condition and Calibration Data Base

[2] This diagram was taken from [10]

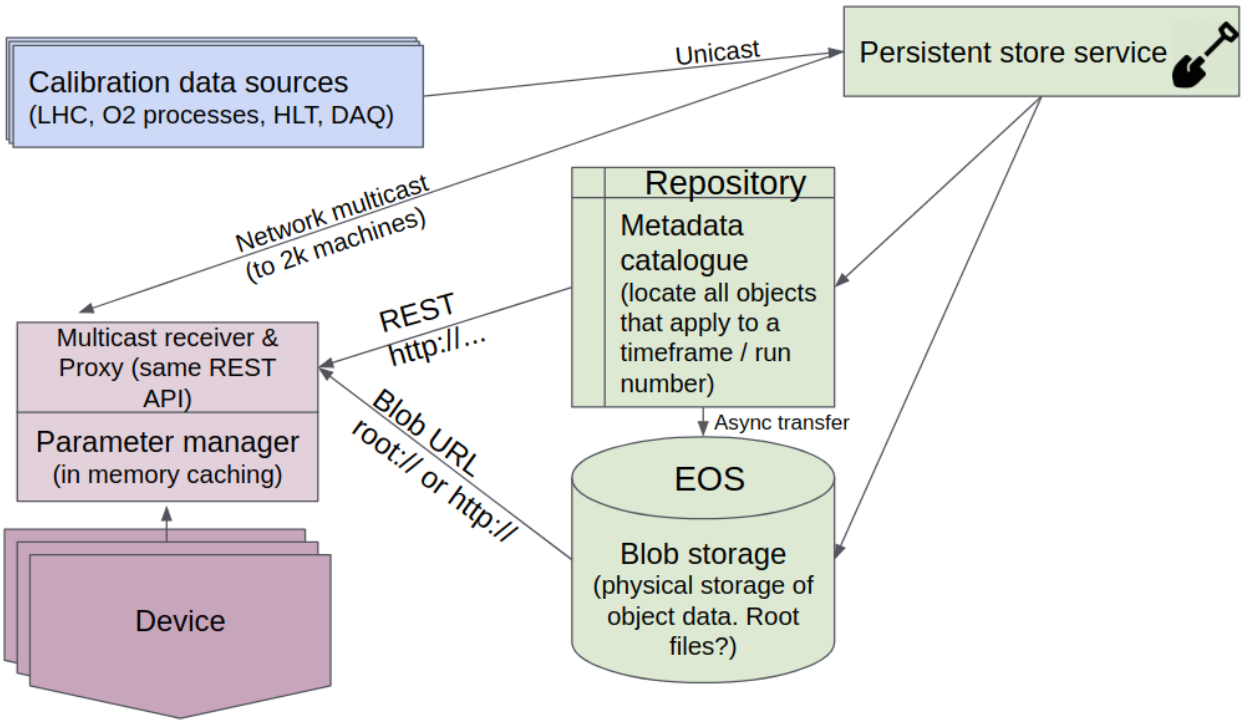

Figure 5: CCDB system overview [10]

(multiple processing nodes with caching service and the same REST API supported).

3.2 Software architecture

This section presents the major software components of this thesis and the services supported by each entity.

3.2.1 Multicast sender and receiver

I have added to the existing framework presented above two main components: a multicast sender and a multicast receiver. The sender is added to the central repository implementation and the receiver is added to the Event Processing Node as shown in Figure 5 and 6.

3.2.2 Main classes

A pair of java classes were developed for this project: the *Sender* and the *MulticastReceiver*. These entities are using the *Blob* class to store a complete data object and *FragmentedBlob* class for a fragmented data object. A *Blob* object has, as it is shown in appendix C, figure 16, two important fields that store useful information: metadata and payload. The *Fragmented-Blob* class stores partial content from a *Blob*: either metadata, either data, depending on the Packet Type field.

3.2.3 Recovery service

All the components implement the same REST API described in appendix A. The Central repository is also using the REST API to register new objects, to response at the recovery request from multicast receivers or to give access to the metadata content. In the *MulticastReceiver* the REST API is an abstraction layer between the memory of the receiver and the local processes that use this data for reconstruction phase.

3.2.4 Caching mechanism

In addition, the receivers have a caching system to store the most recently version of a Blob and a recovery system to request for incomplete or missing Blobs. For more detailed information about the classes methods and fields please consult the class diagrams from appendix C.

3.3 Data flow and system operation

In this section I will present how the conditions data will be distributed in the ALICE network starting from the FLPs to the EPNs and how the recovery system is used to obtain a reliable transmission.

3.3.1 Overview

The Central repository is receiving conditions data (e.g. magnetic field intensity, current levels, operating temperatures etc.) from the sensors every 20ms. Next, the data should be available for a farm of 1500 EPN [3]; 30 seconds is the time frame size in which an EPN should process the brute data slice and it has to obtain the the latest conditions data from the central repository if it was lost during the multicast transmission. The recovery process should end before the actual processing of the raw data slice obtained from the FLPs.

3.3.2 Sending multicast data

The Multicast Sender transfers each new Blob to the EPN cluster via multicast messages like in Figure 6. If the length of the metadata and the payload combined is exceeding certain configurable value (by default 64KB - see equation 1 for details) than the Sender decides to fragment the metadata and the payload into multiple fragmented Blobs. Each fragmented Blob is sent via UDP messages to the Multicast Receivers.

[3] Event Processing Node

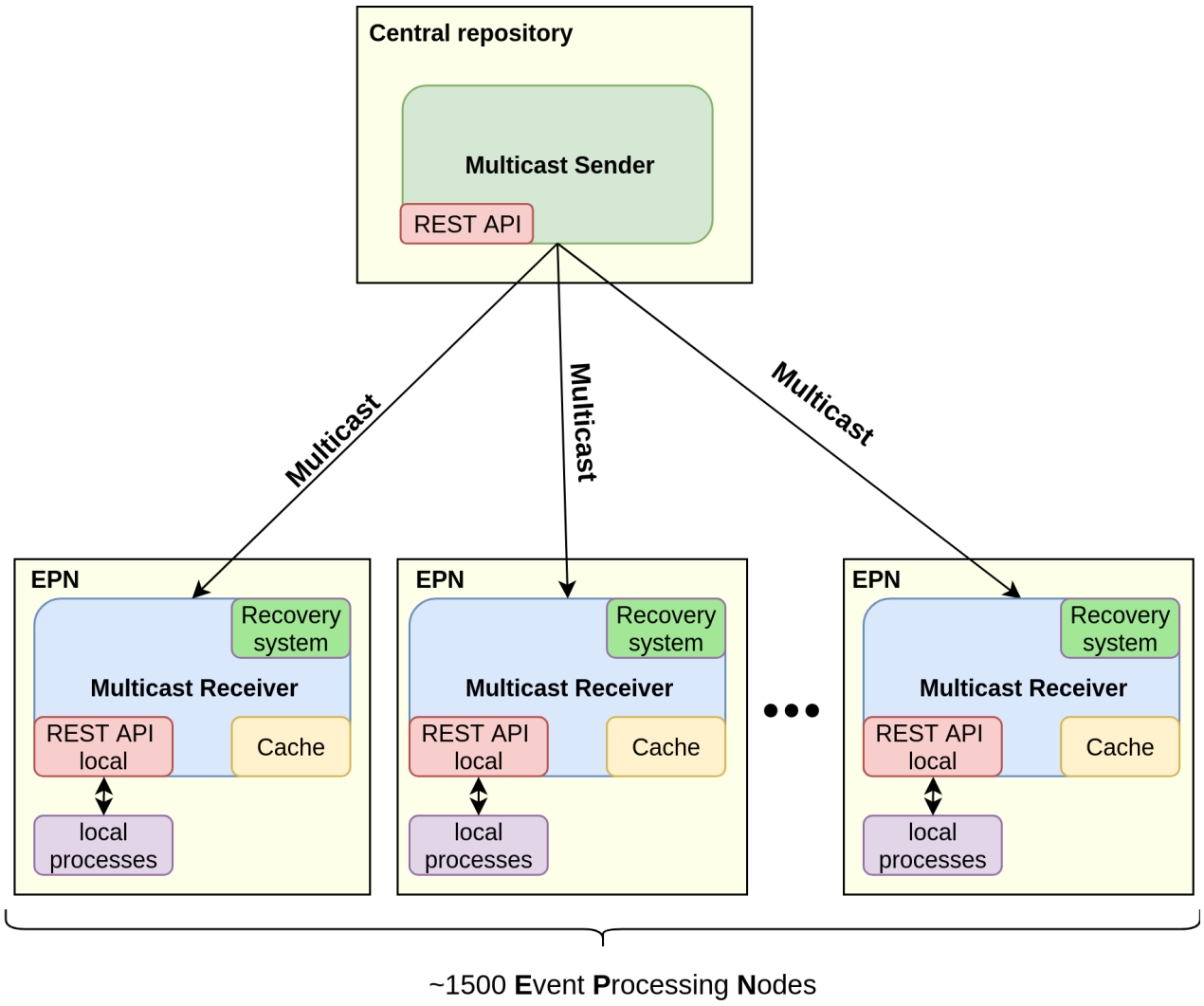

Figure 6: Sending an object via multicast messages

3.3.3 Reassembling process

Receivers are waiting for complete data objects by keeping the incomplete blobs in a ConcurentHashMap data structure named *inFlight* to allow safe multithreading access to the Blobs. The recovery thread should check the timestamp of the incomplete Blobs from the *inFlight* data structure. If the incomplete Blob was not updated recently, the recovery procedure starts. Once a Fragmented Blob was received, the *MulticastReceiver* stores its arrival time and it searches for the corresponding incomplete Blob by its unique identifier and then it adds this fragmented Blob to the incomplete Blob. If the Blob is now complete, the object is moved in the current cache memory. This cache memory content is used by the REST API to respond to the local processes that run on the EPN. In addition, the REST API is also aware of the *inFlight* content. If a local process request matches an incomplete Blob, the REST API can (policy based) either wait for the rest of the Blob to be received or simply redirect the client to the central repository.

3.3.4 Recover incomplete data

If an incomplete Blob was not updated in a configurable time interval (by default 1 second), the recovery thread from the Multicast Receiver makes a GET request with the exact missing byte ranges to the Central repository as it can be observed in Figure 10, to be more precise I have added some recovery data examples request - response in the appendix B.

3.3.5 Reconstruction phase

According to the ALICE Technical Design Report for O2 [9], each EPN will need a buffer to receive the next time frame (10GB), a buffer for current time frame (10GB), buffer for processing the current time frame (1GB per CPU core) and 16GB for the OS, i.e. a total of approximately 100GB. An EPN has 30 seconds to process a 20ms slice data stream from the FLPs [4]. This data consists of partially compressed data or raw data and it has to be processed fast by the 32 core CPUs of the EPNs. During this time the next data slice arrives asynchronously to be ready for the next processing cycle.

When the EPN starts processing the next time frame it will validate the calibration data objects it has in memory. For that it will request to the local (caching) service each key that is needed for processing the data. The answer is either HTTP 304 (not modified), when the object in memory is still valid, the new Blob content, if it has changed in the mean time, or a redirect to the central service if the cache doesn't hold the (entire) content for that key at the moment.

As the time budget for processing a time frame is extremely constrained (30s), all the above queries are on the critical path of data processing and have thus to be highly optimized for this environment

[4] First Level Processors

4 IMPLEMENTATION DETAILS

This Chapter contains the implementation details of the real-time conditions data distribution for the Online data processing of the ALICE experiment project. First, I will present the Java mechanism that I used to send multicast message over the network, after that I will describe the caching service and the congestion control technique. Next, I will describe the structure of the serialized packets sent via multicast messages, the recovery system work-flow, the algorithms used by the fragmentation and reassembling features and finally the development history of this thesis.

4.1 Multicast IP addresses

To send a multicast message using the IP protocol a certain address range has to be used: 224.0.0.0 - 239.255.255.255. However, not all the address from this range are recommended to be used (see Table 1), but in the private network described in chapter 5 I tested my solution using the reserved address **230.0.0.0** .

Address Range	**Size**	**Designation**
224.0.0.0 - 224.0.0.255	/24	Local Network Control Block
224.0.1.0 - 224.0.1.255	/24	Internetwork Control Block
224.0.2.0 - 224.0.255.255	65024	AD-HOC Block I
224.1.0.0 - 224.1.255.255	/16	RESERVED
224.2.0.0 - 224.2.255.255	/16	SDP/SAP Block
224.3.0.0 - 224.4.255.255	2 * /16s	AD-HOC Block II
224.5.0.0 - 224.255.255.255	251 * /16s	RESERVED
225.0.0.0 - 231.255.255.255	7 * /8s	RESERVED
232.0.0.0 - 232.255.255.255	/8	Source-Specific Multicast Block
233.0.0.0 - 233.251.255.255	16515072	GLOP Block
233.252.0.0 - 233.255.255.255 (/14)	/14	AD-HOC Block III
234.0.0.0 - 238.255.255.255	5 * /8s	RESERVED
239.0.0.0 - 239.255.255.255	/8	Administratively Scoped Block

Table 1: IPv4 multicast address range according to [19]

4.1.1 Java DatagramSocket

Sending a java object via network In order to send an object of the *Blob* class (see Figure 16) via a *DatagramSocket*, the Blob should be serialized into a byte array and fragmented if the size exceeds the maximum UDP payload size. The serialized version of the (fragmented)Blob is the first argument passed in the *sendFragmentMulticast* method (see Figure 7) from *Utils* class - see class diagram shown in Figure 17. This is the main function for sending messages via Multicast messages as shown in Figure 6.

```
public static void sendFragmentMulticast(byte[] packet, String destinationIp, int destinationPort)
        throws IOException, NoSuchAlgorithmException {
    try (DatagramSocket socket = new DatagramSocket()) {
        InetAddress group = InetAddress.getByName(destinationIp);
        DatagramPacket datagramPacket = new DatagramPacket(packet, packet.length, group, destinationPort);
        socket.send(datagramPacket);
    }
}
```

Figure 7: Send a serialized fragmentedBlob object via multicast message

4.2 Caching service

To reduce the number of request to the central repository the multicast receiver uses a caching service. It stores the complete Blobs and incomplete Blobs with their arriving and expiration timestamp. If a local process needs a Blob the EPN will search for it in its cache memory. A request to the central repository is not necessary if the Blob from the cache is still valid.

4.2.1 Types of cache

The multicast receiver has two types of data structures where it keep Blobs:

- *inFlight* cache - uses a ConcurentHashMap$< UUID, Blob >$ for storing the incomplete Blobs
- *currentCacheContent* - uses a ConcurentHashMap$< String, Blob >$ to associate fully received Blob objects to a particular key

4.2.2 Congestion Control

The basic operating mode for this project requires multiple threads for packet processing in the receiver. The performance testing sender is configured to run on its maximum rate limit, so it will send the new generated Blobs as fast as it can (see chapter 5 for details). Receiving a large number of Blobs per second means an even greater number of fragmented Blobs packets per second. In order to process and reassemble these fragments I used the following java mechanism for multithreading:

- **ExecutorService** - an asynchronous execution mechanism
  - with a *FixedThreadPool* of size equals to the number of available processors
  - each receiving thread runs the *processPacket* method which reassemble Blobs
- **ConcurentHashMap** - an efficient thread-safe map to keep data objects
  - with atomic methods to add one entry, like: to replace a Blob: *computeIfAbsent*

4.3 Packet structure

Since the *DatagramSocket* accepts only a byte array as the data to be sent, I had to implement my own serializing tool to convert a *fragmentedBlob* object to a byte array using the structure presented below. For performance reasons, I implemented a hybrid mechanism that fragments the Blobs and serializes the fragments. In the following sections I will present the common protocols used by the sender and receiver to communicate the fragmented data object. These protocol describes the structure of a fragmented packet that contains data or metadata. I choose to avoid the Java Serialization [20] because I need to have the same deserialization mechanism for all the receivers even though they use a different language.

4.3.1 FragmentedBlob

Because the UDP maximum payload size is less than 64 KB, below the expected Blob size (2 MB), this project proposes a fragmentation mechanism to send big data object using multicast messages. Each fragment will be identified by its key and UUID to establish the Blob in which the fragment should be placed and by its offset to decide where in the Blob's payload should the fragment content should be copied.

4.3.2 Header fields description

Packets sent via a *DatagramSocket* uses the Fragmented Blob structure presented in Figure 8. The fields have the following meaning:

$FragmentOffset$	= start index of this fragment payload in the Blob
$PacketType(Flags)$	= Indicates what kind of payload does this fragment carry
$UUID$	= Universally Unique IDentifier, also used as ETag in the REST API
$BlobPayloadLength$	= the total length of the Blob's payload or Blob's metadata
$KeyLength$	= the length of the key associated with the current Blob
$BlobPayloadchecksum$	= the checksum of the payload or metadata
Key	= the key content with size x
$Payload$	= the Blob's (partial) metadata or payload with size y

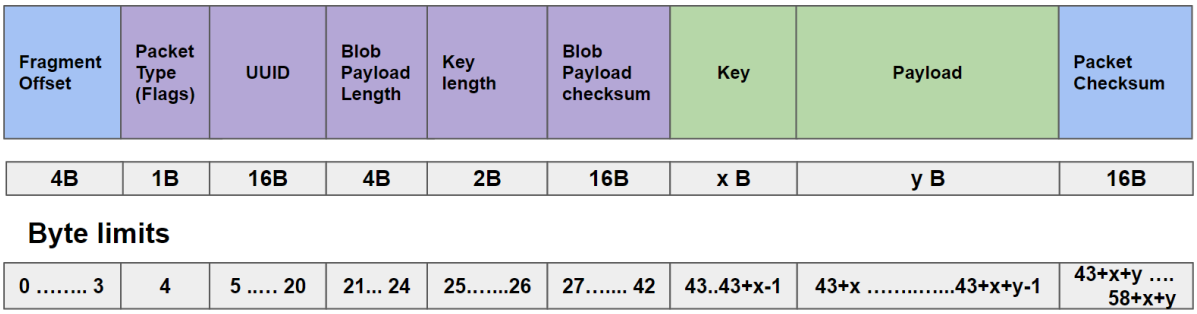

Figure 8: Serialized Fragmented Blob structure

As per the above, some of the fields have multiple roles depending on the type of payload carried by the packet (Packet Type): Blob actual data or metadata. The Packet Type field can carry even more information in the future since 6 bits are not used yet. As it is showed in Figure 8, one fragmented Blob packet has several fields used by the receiver unit to reassemble the fragments into the right Blob identified by its key and its UUID. The fragment Offset field is used to place the payload of the fragmented Blob at the right position in the Blob. This field is variable for different fragments of the same Blob (colored with blue).

4.3.3 Fragmentation - reusing constant value header fields

For performance reasons I chose to reuse the common part of the Fragmented Blob header, there is no need to reallocate this fields for every packet since the content of these field will not change. When fragmenting a Blob, each packet will have the same values in the following fields:

- Packet Type (only the least significant two bits are used)
  - 00 : the fragmentedBlob carries the Blob's (partial) metadata
  - 01 : the fragmentedBlob carries the Blob's (partial) payload
  - 10 : the fragmentedBlob carries an entire Blob
  - 11 : unused
- Universally Unique IDentifier
- Blob Payload Length (or Blob's serialized metadata length)
- Key Length
- Blob Payload checksum (or Blob's metadata checksum)

The fragments of a Blob have the following variable content fields:

- Fragment Offset - an int of size 4 which will be useful for the reassembling process at the Receiver
- Payload - the useful content carried by the packet
- Packet checksum - the md5 value for the first 8 fields of the packet

4.3.4 Metadata serialization

As the ALICE O2 Upgrade Technical Design Report [9] mentions, metadata will be correlated to the data objects to describe their properties, like "the data taking period to which they refer, the calibration type, the source etc.". This information will be stored in a Map and be serialized for network transport as follows:

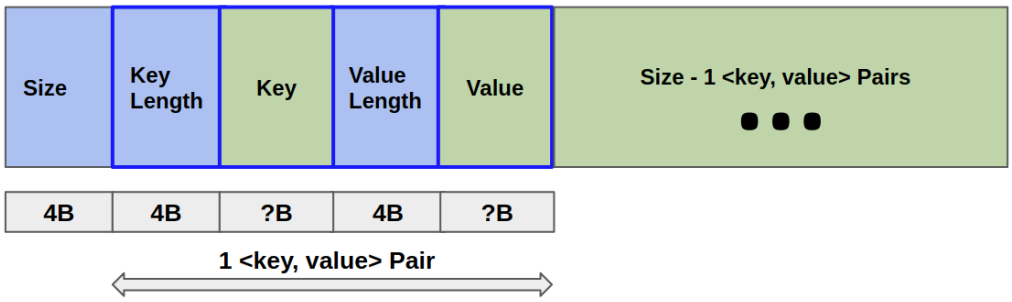

Figure 9: Serialized metadata structure

$Size$ = Total nunmber of $< key, value >$ pairs
$KeyLength$ = Length of the Key in bytes
Key = A String with the actual content of the Key
$ValueLength$ = Length of the Value in bytes
$Value$ = A String with the actual content of the Value

4.4 Recovery service - a REST client

If some fragments do not reach the destination in due time, the multicast receivers will detect this behaviour using their dedicated *incompleteBlobRecovery* thread. This thread runs with an adaptable frequency (depending on the *inFlight* map size) and checks if the last modified timestamp of a Blob is too old (default 1 second), if so, it will create a GET request to recover the missing data Blocks.

4.4.1 Overview

Even though the network infrastructure in the target environment is built using state of the art components (100 Gbps Ethernet or Infiniband cards), it will be continuously used by the raw data acquisition of the experiment. Packet loss events are thus expected to happen with much higher probability than in a test environment of similar characteristics. The test scenarios for this feature simulate arbitrary high packet loss by simply not sending random packets, in the same statistic amount as the simulated packet loss. UDP uses an unreliable best-effort delivery so I designed the following fallback system using the existing REST API implemented on the central repository.

4.4.2 Workflow - using asynchronous execution

Each EPN, has a working thread that checks periodically (the frequency depends on the map size) the incomplete Blobs received stored in a *ConcurrentHashMap* named *inFlight*. Since each EPN will process one full Time Frame every 30 seconds according to [9], than it can independently decide a time interval in which it can wait the arrival of the most recent data associated with a key. If the incomplete Blob was not updated in the last second (default value), the incompleteBlobRecovery thread initiate the recovery procedure. This mechanism is based on the REST API implemented by the central service. It uses GET requests to recover an entire Blob or a certain part of the Blob using GET Byte Range request [21]. The recovery request - response flow can be analysed in Appendix B.

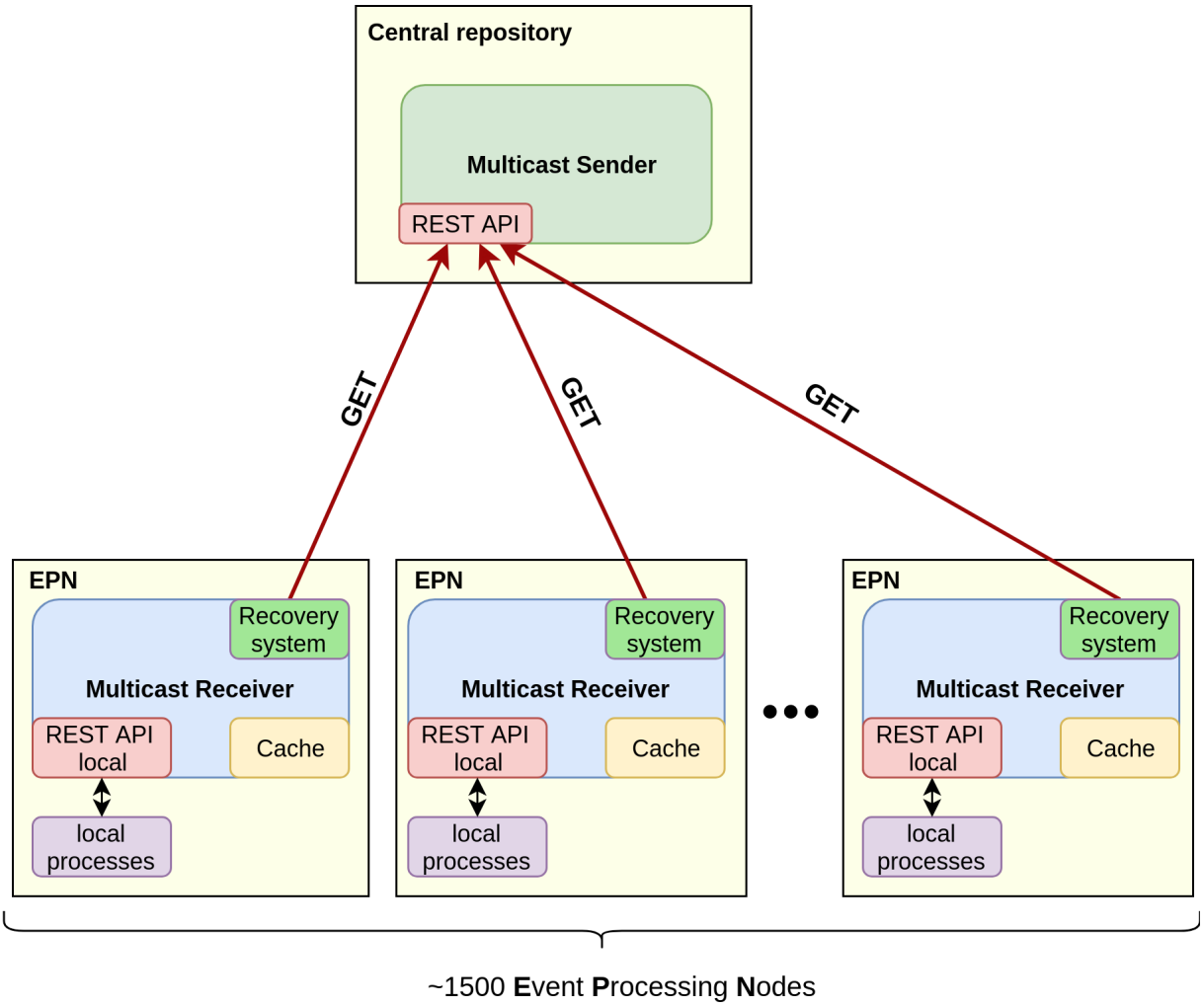

Figure 10: All the processing nodes are requesting lost data (worst case scenario)

4.5 Reassembling process - algorithms

The receiver receives the data fragments and it uses the fragment offset and the payload length to reassemble the Blob content. In the following sections I will present the data structures and algorithms used for this process to design a fast method for checking if a Blob is complete.

4.5.1 Reducing byteRange size

In order to reduce the computing cost of the repeatedly checking $isComplete$ for a Blob (a Blob is considered complete is it has received its payload and metadata correctly via fragmented Blobs) I choose to implement an additional method to merge the existing byteRanges intervals (see Algorithm 2).

4.5.2 Adding new fragments to Blobs

Adding a new fragment to a Blob requires two actions: first copy the content of the fragment's payload to the Blob's payload or metadata (depending on packet type) starting from fragmentOffset and second add a a Pair object to the byteRange with the following values: [fragmentOffset, fragmentOffset + fragment payload size] (see Algorithm 1).

Algorithm 1 Adding a new interval to byteRanges

```
1:  procedure ADDFRAGMENT([a,b])
2:      index ← -1
3:      for i ← 1, payloadRanges.size do
4:          if payloadRanges[i].second = a then
5:              payloadRanges[i] ← [payloadRanges[i].first, b]
6:              index ← i
7:              break
8:          else
9:              if payloadRanges[i].first = b then
10:                 payloadRanges[i] ← [a, payloadRanges[i].second]
11:                 index ← i
12:                 break
```

In addition, after adding the interval to the byte ranges data structures, algorithm 2 is executed. The $index$ parameter of the mergeByteRange is the most recently modified pair from the byteRanges. This is the element that is most likely to be joined with another element as it is done at line number 6 or 10 from the mergeByteRanges method 2. Doing so will decrease the size of the byteRanges data structure which will allowed a faster implementation for the isComplete method 3.

Algorithm 2 Merge existing byteRange intervals

```
1: procedure MERGEBYTERANGES(index)
2:     Pair pair ← payloadRanges[index]
3:     for i ← 1, payloadRanges.size do
4:         if i ≠ index then
5:             if payloadRanges[i].first = pair.second then
6:                 payloadRanges[i] ← [pair.first, payloadRanges[i].second]
7:                 delete payloadByteRanges[index]
8:                 break
9:             if payloadRanges[i].second = pair.first then
10:                payloadRanges[i] ← [payloadRanges[i].first, pair.second]
11:                delete payloadByteRanges[index]
12:                break
```

4.5.3 isComplete Blob verification

Recovery thread is frequently checking if a partially received Blob from the *inFlight* data structure is complete. In addition, *isComplete* method is executed by the packet processing threads. Calling this method every time a packet is received involves an implementation with a special design and fast algorithms. I have managed to achieve this by keeping track of the byte ranges received in an *ArrayList* of intervals (*Pair* - see *Pair* class in appendix C). Every time a packet is received the system will search the corresponding Blob by using its key, will add the fragment to it and also add the interval to the *byteRange* data structure as it is presented in Algorithm 1. After this, the *isComplete* method is called (see Algorithm 3).

Algorithm 3 isComplete

```
1: procedure ISCOMPLETE
2:     if metadata = null or payload = null then
3:         return false
4:     if payloadByteRanges.size ≠ 1 or metadataByteRanges.size ≠ 1 then
5:         return false
6:     if metadataByteRanges[0] ≠ (0 , metadata.length) then
7:         return false
8:     if payloadByteRanges[0] ≠ (0 , payload.length) then
9:         return false
10:    if metadata checksum fails then
11:        return false
12:    if payload checksum fails then
13:        return false
14:    return true
```

4.6 Development

The timeline of this project started with the research of similar technologies listed in chapter 2. After this, I have decided to design a pair of two java programs - a sender and a receiver - in order to validate the idea that sending multicast packets with a low less percentage in the testing environment at CERN will be the most efficient solution for fast and reliable data delivery. This idea was validated by the tests presented in the next chapter. The code source for this project is public available on github [22].

5 TESTING AND EVALUATION

This chapter evaluates the performance of three main features of this solution:

- Multicast loss percentage
- Fragmentation and reassembling mechanism efficiency
- Recovery system success rate

5.1 Testing approach

I will present the results of sending burst multicast messages to analyze the multicast loss percentage in the network topology described below. Next, I will evaluate the efficiency of the fragmentation and reassembling mechanism and finally the recovery system success rate. In addition, the throughput and Blob send rate where analyzed to meet the project objectives presented in Chapter 1.2.

5.2 Hardware configuration

The testing was conducted using an Ubuntu 16.04.6 or 18.04.2 x86_64 system, equipped with Broadcom Corporation NetXtreme II BCM57810 10 Gigabit Ethernet Network Cards and the following CPU configuration:

Hostname	**Model name**	**Number of cores**
pcalienstorage	Intel(R) Xeon(R) CPU E5-2640 0 @ 2.50GHz	24 cores (2 threads / core)
aliendb06h	Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz	32 cores (2 threads / core)
aliendb06f	Intel(R) Xeon(R) CPU E5-2687W v4 @ 3.00GHz	48 cores (2 threads / core)
aliendb06g	Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz	40 cores (2 threads / core)

Table 2: Hardware configuration

5.3 Network topology

The evaluation for this project was made using four machines with the hardware configuration specified in table 2. One of the machines sends random content Blobs via DatagramSockets to a multicast address, the other three machines are using the MulticastSocket to joing the

multicast group. Two of the machines *aliendb06h* and *aliendb06g* belong to the same subnet (137.138.99.128/26 according to [23]), the other two belong to 188.184.2.0/26 (*aliendb06f*) and 137.138.47.192/26 (*pcalienstorage*) respectively. I have chosen this topology because these fours from ALICE infrastructure at CERN were equipped with high bandwidth network cards and because they are monitored using the MonALISA[1] Grid Monitoring tool. The EPN hardware is not available yet and ALICE will repeat the tests in a few months when the first servers will be delivered and installed in the cluster.

5.4 Tests

In this section I will present the results for the tests. I have evaluated the three features mentioned at the beginning of this Chapter. The multicast loss percentage was calculated for complete Blobs, but also for simple UDP packets in order to analyze the fragmentation and reassembling mechanism efficiency. Depending on the loss percentage, I will discuss and evaluate the recovery system operation and success rate.

5.4.1 Loss percentage

It is important to have a low percentage of lost Blobs during a transmission. Many recovery requests might be initiated at the same time by the receivers, all pointing to the single Central repository. This scenario might lead to bandwidth exhaust due to the fact that an HTTP GET request uses a TCP connection.

5.4.2 Blob loss percentage

The percentage of complete Blob lost using different values for *Max Payload Size* for a fragmented Blob. This test shows that a small value for the fragment's payload size will decrease the total number of lost Blobs because the small fragments have a high probability to reach the destination as it can be observed in appendix D. In addition, losing a random number of fragments will lead to an even larger number of lost Blob (lost Blob = incomplete or absent Blob).

5.4.3 Basic packet loss testing

To analyze the results from Figure 11, I needed to run a similar test, but without the fragmentation/reassembling, so I created a BurstSender class that would send an empty packet with the same packet size as in Figure 12. As it can be observed, some lost fragment increased the lost complete Blob percentage.

[1] MONitoring Agents using a Large Integrated Services Architecture

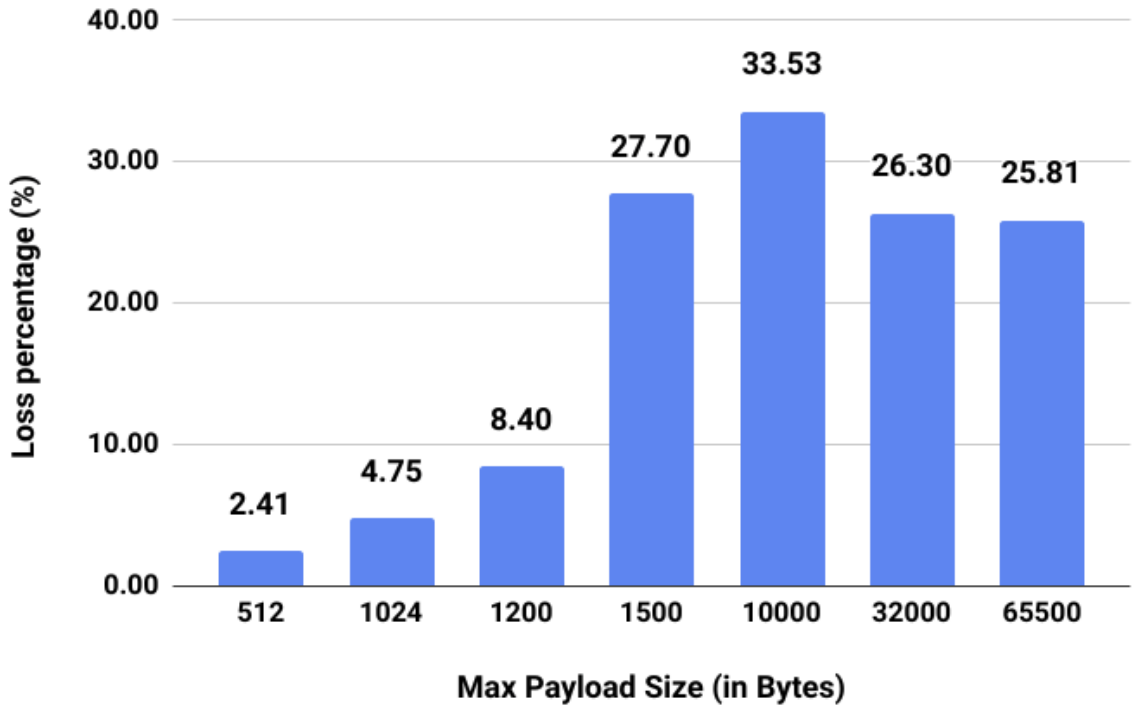

Figure 11: Complete Blob lost percentage in relation with the fragmented Blob payload size

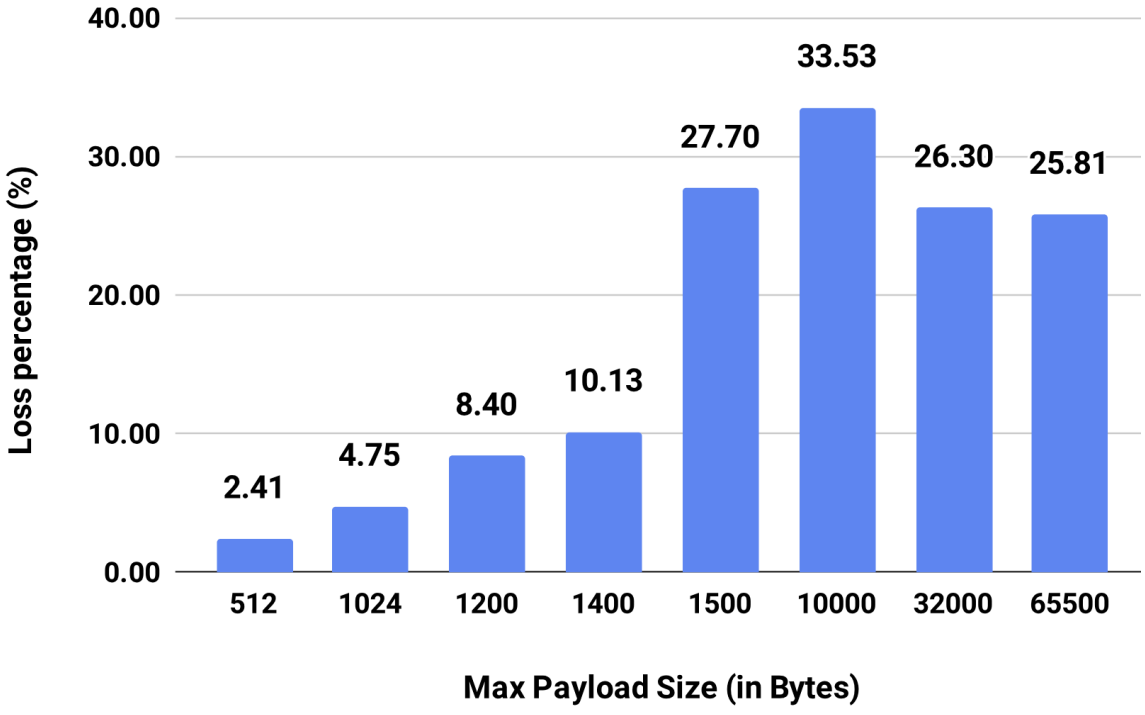

Figure 12: Simple UDP packets lost percentage in relation with the their payload size

5.5 Throughput

I have measured the network capability of the EPN to receive Blobs in Figure 13. The maximum throughput with a low loss percentage was achieved using the payload size of 1400 B (658 MB/s). Considering the fact that the target to achieve was 50Hz x 2MB = 100 MB/s, I consider that this throughput is good enough to ensure a good performance for my project. The packets with payload size equal to 1400 B are not fragmented by the routers, so this size is preferred for transport of the Blobs. A 1500 B payload size will exceed the 1500 B MTU for IPv4 networks because in addition to the actual useful data the packet needs to carry header fields, so there will be an IP level fragmentation sending two packets instead of one.

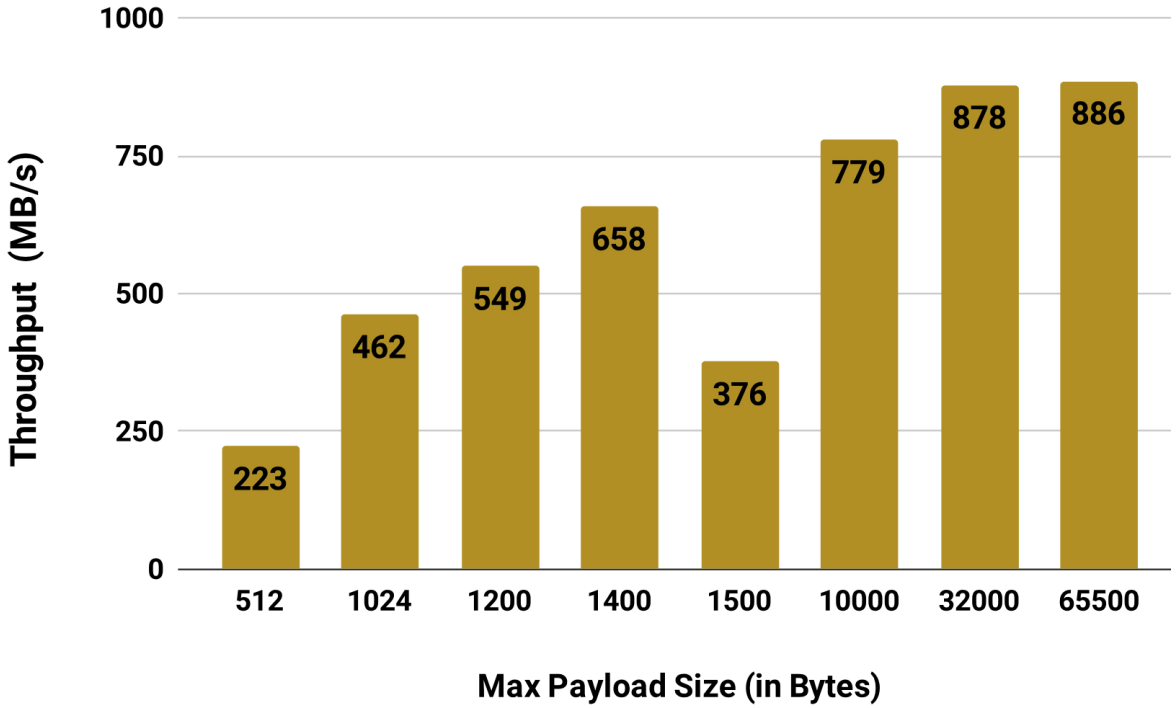

*Figure 13: Network throughput - Number of fragments arrived on MulticastReceiver per second * payload size*

Recovery system simulation In order to test the recovery system feature, I have conducted several tests with a configurable loss percentage of the fragmented Blob on the *pcalienstorage* machine using one Sender process and three other receiver processes. I have chosen this option because I needed a controlled environment to test the recovery of fragmented Blobs. I have managed to send Blobs at 50 Blobs per second using the local network stack of the machine. I have introduced a loss percentage between 2% and 10% and my recovery system succeed to recover all the fragmented Blob and to complete the remaining Blobs from *inFlight* map.

5.6 Network monitoring

To measure the throughput in my setup, I have used dedicated counter threads in the *Sender* and the *MulticastReceiver*. These threads print the number of complete Blobs received every second. In addition to this quantitative counting, I have also used a quantitative way of checking the packet rate using the MonALISA network traffic monitoring tool.

5.6.1 MonALISA network traffic monitoring

By using a secondary monitoring system [24], it can be observed that the **OUT** traffic from the Sender unit is approximately equal with the **IN** traffic of the receiver divided by the number of receivers. See the appendix D. This confirms the fact that sending a high number of Blobs per second will not lead to many complete Blob loss.

6 CONCLUSION AND FUTURE WORK

This Chapter will draw the conclusions and propose new improvement ideas in the future. First, it is summarized the current state of the project. next, the sender and receiver implementations will need to increase their response time by improving their parallel execution. Finally, the integration with the current framework is outlined.

In this thesis I introduced a real-time distribution mechanism for conditions data based on a reliable multicast protocol. I was able to send data objects of 2 MB from the Central repository to a few *MulticastReceiver* entities with a reasonable loss percentage (less than 2%) and recover all the missed fragments in the time frame given.

6.1 Current status

The current implementation status has solved the problems listed in section 1.1 for a small number of receivers. Nevertheless, more tests should be done on a larger scale after the final hardware implementation and network topology is determined. In the following months this project should be able to send data object of 2 MB to approximately 1500 receivers before it can be fully integrated in the first stable release of the framework.

6.2 Recovery system

The recovery system proposed by this thesis is feasible to use in this framework. The recovery system is efficient for a small number of request, but more tests should be done in order analyze the impact of a high number of GET request in the network.

6.3 The Sender

Currently, the Sender has a sequential method to send the fragments, but it can process in parallel multiple recovery request since it uses an *Apache Tomcat Servlet*. This tool improves the response time of the Central repository to the recovery request, but more investigations should be made to see the impact of this implementation regarding the network capacity with a high number of recovery request.

6.4 The Receiver

The receiver uses multiple threads for processing (deserialization) and reassembling the packets. Also, a dedicated thread is working to recover the lost fragments from the incomplete Blobs. Further work should be done at the Receiver implementation to increase the level of parallelism of the work and to solve the concurrency problems.

The reassembling of the received packets can be made by a *MulticastReceiver* implemented in a different language than Java using the Fragmented Blob serialized version structure proposed by this thesis.

6.5 Integration with current framework

This thesis proposes an efficient reliable multicast distribution built on top of the current framework. The recovery system uses the *servlet* container named *Tomcat* with the same REST API already implemented by the Central repository.

In conclusion I propose a hybrid data delivery mechanism: a network multicast-based data propagation in a large distributed environment that uses a REST API for recovering data and and an adaptable fragmentation mechanism to send large data object that exceed the maximum UDP payload size.

BIBLIOGRAPHY

[1] The large hadron collider. https://home.cern/science/accelerators/large-hadron-collider. Accessed: June 2019.

[2] ATLAS Collaboration et al. The atlas experiment at the cern large hadron collider, 2008.

[3] CMS Collaboration et al. The cms experiment at the cern lhc, 2008.

[4] Kenneth Aamodt, A Abrahantes Quintana, R Achenbach, S Acounis, D Adamová, C Adler, M Aggarwal, F Agnese, G Aglieri Rinella, Z Ahammed, et al. The alice experiment at the cern lhc. *Journal of Instrumentation*, 3(08):S08002, 2008.

[5] A Augusto Alves Jr, LM Andrade Filho, AF Barbosa, I Bediaga, G Cernicchiaro, G Guerrer, HP Lima Jr, AA Machado, J Magnin, F Marujo, et al. The lhcb detector at the lhc. *Journal of instrumentation*, 3(08):S08005, 2008.

[6] Alice detector components. http://aliceinfo.cern.ch/Public/Objects/Chapter2/DetectorComponents/alice2.jpg. Accessed: June 2019.

[7] Kenneth Aamodt, N Abel, U Abeysekara, A Abrahantes Quintana, A Acero, D Adamova, MM Aggarwal, G Aglieri Rinella, AG Agocs, S Aguilar Salazar, et al. First proton–proton collisions at the lhc as observed with the alice detector. *The European Physical Journal C*, 65(1-2):111, 2010.

[8] Alice records first proton-lead collisions at the lhc. https://commons.wikimedia.org/wiki/File:Test8.png. Accessed: June 2019.

[9] P Buncic, M Krzewicki, and P Vande Vyvre. Technical design report for the upgrade of the online-offline computing system. Technical report, 2015.

[10] Ccdb project last status report. https://docs.google.com/presentation/d/14z0-jQOaesjPl3_2RjWGwcx75jk2J24M7GHxnmtBuUo/edit?usp=sharing. Accessed: June 2019.

[11] Katia Obraczka. Multicast transport protocols: a survey and taxonomy. *IEEE Communications magazine*, 36(1):94–102, 1998.

[12] Uftp - encrypted udp based ftp with multicast. https://tools.ietf.org/html/rfc768. Accessed: June 2019.

[13] Sanjoy Paul, Krishan K. Sabnani, JC-H Lin, and Supratik Bhattacharyya. Reliable multicast transport protocol (rmtp). *IEEE Journal on Selected Areas in Communications*, 15 (3):407–421, 1997.

[14] Jörg Nonnenmacher, Ernst W Biersack, and Don Towsley. Parity-based loss recovery for reliable multicast transmission. *IEEE/ACM Transactions on networking*, 6(4):349–361, 1998.

[15] David Borman, Steve Deering, and Robert Hinden. Ipv6 jumbograms. Technical report, 1999.

[16] Steve Deering and Robert Hinden. Internet protocol, version 6 (ipv6) specification. Technical report, 2017.

[17] The apache cassandra database. `http://cassandra.apache.org/`. Accessed: June 2019.

[18] Eos is a disk-based, low-latency storage service. `http://information-technology.web.cern.ch/services/eos-service`. Accessed: June 2019.

[19] Iana guidelines for ipv4 multicast address assignments. `https://tools.ietf.org/html/rfc5771`. Accessed: June 2019.

[20] Java serialization oracle documentation. `https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html`. Accessed: June 2019.

[21] Hypertext transfer protocol (http/1.1): Range requests. `https://tools.ietf.org/html/rfc7233`. Accessed: June 2019.

[22] Code source on github. `https://github.com/dosarudaniel/ReliableMulticastForALICE`. Accessed: June 2019.

[23] Alice network topology: the switches. `https://docs.google.com/presentation/d/1WLaDx7KQg7EWs1i6B286EVk2Nt7yBkIoU2sJdNcEFgs/edit?usp=sharing`. Accessed: June 2019.

[24] Monalisa grid monitoring network in/out example. `http://alimonitor.cern.ch/display?imgsize=1024x600&interval.max=0&interval.min=3600000&modules=machines%2Fnet%2Fmachines_in&modules=machines%2Fnet%2Fmachines_out&page=machines%2Fnet%2Fmachines&plot_series=aliendb06f&plot_series=aliendb06g&plot_series=aliendb06h&plot_series=pcalienstorage`. Accessed: June 2019.

A REST API

The REST API implemented by the central repository

```
GET:
task name / detector name / start time / <UUID>
    - return the content of that UUID, or
task name / detector name / [ / time [ / key = value]* ]
POST:
task name/detector name/start time[/end time][/UUID][/key = value]*
binary blob as multipart parameter called 'blob'
PUT:
task name/detector name/start time[/new end time][/UUID][?(key=newvalue&)*]
DELETE:
task name/detector name/start time/UUID
or any other selection string, the matching object will be deleted

Usage of /browse/* or /latest/*:
GET:
task name / detector name / [start time, default = now] [/key=value]*
```

B RECOVERY EXAMPLE

Request an entire Blob

```
GET /Task/Detector/1/183fe150-921c-11e9-9a8b-7f0000015566 HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.58.0
Accept: */*
```

Response:

```
HTTP/1.1 200
Date: Tue, 18 Jun 2019 23:06:14 GMT
Valid-Until: 100000
Valid-From: 1
Created: 1560898497509
ETag: "183fe150-921c-11e9-9a8b-7f0000015566"
Last-Modified: Tue, 18 Jun 2019 22:54:57 GMT
Accept-Ranges: bytes
Content-Disposition: inline;filename="passwd"
Content-MD5: 87f83417335e3dd1a1d27183bac21fa4
Content-Type: application/octet-stream
Content-Length: 2407

<Complete Blob content>
```

Request the first 100 byte from a Blob

```
GET /Task/Detector/1/183fe150-921c-11e9-9a8b-7f0000015566 HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.58.0
Accept: */*
Range: bytes=0-100
```

Response:

```
HTTP/1.1 206
```

```
Date: Tue, 18 Jun 2019 23:06:18 GMT
Valid-Until: 100000
Valid-From: 1
Created: 1560898497509
ETag: "183fe150-921c-11e9-9a8b-7f0000015566"
Last-Modified: Tue, 18 Jun 2019 22:54:57 GMT
Content-Range: bytes 0-100/2407
Content-Disposition: inline;filename="passwd"
Content-Type: application/octet-stream
Content-Length: 101

<Partial Blob content>
```

C CLASS DIAGRAMS

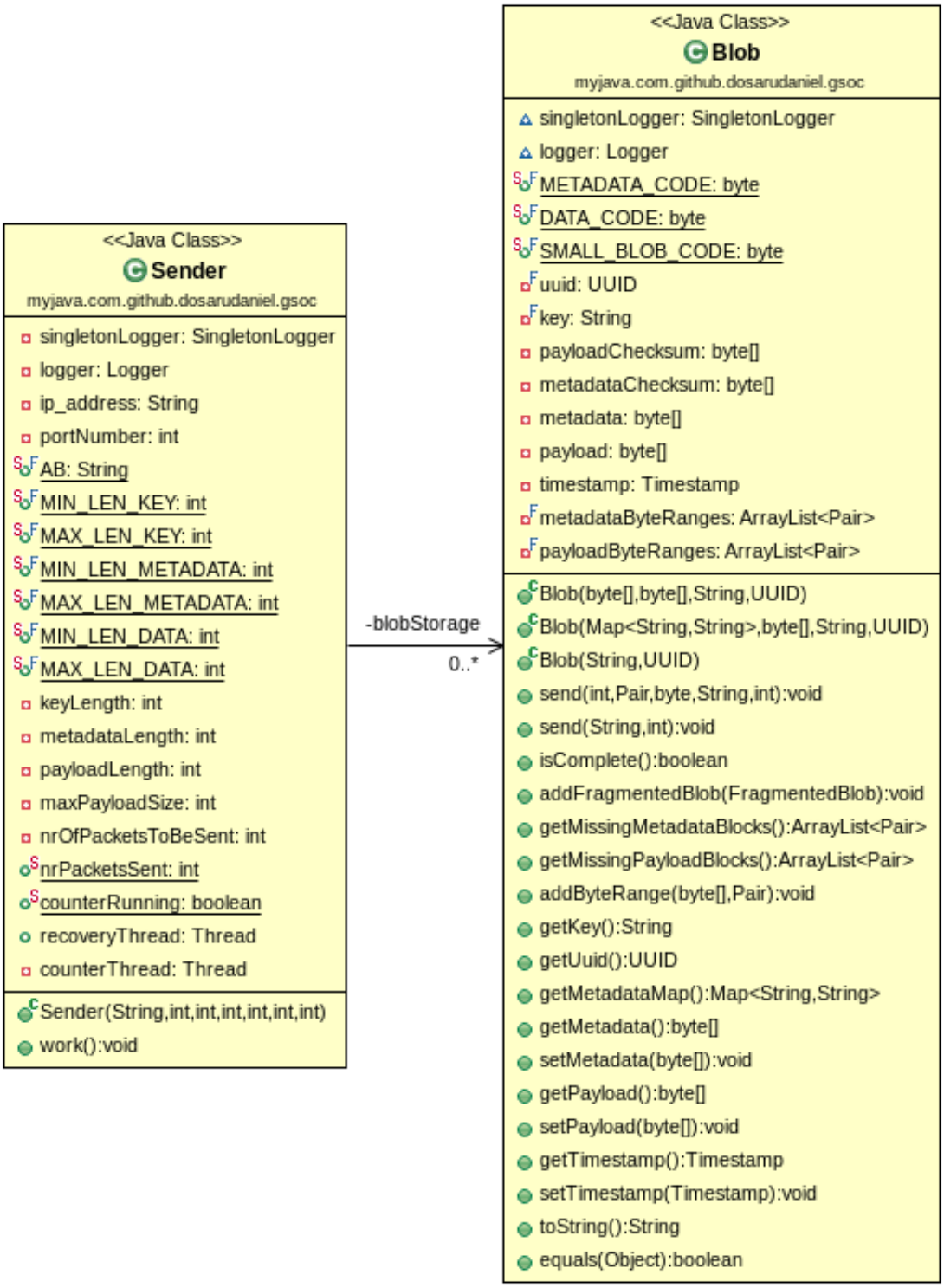

Figure 14: Sender and Blob class diagram

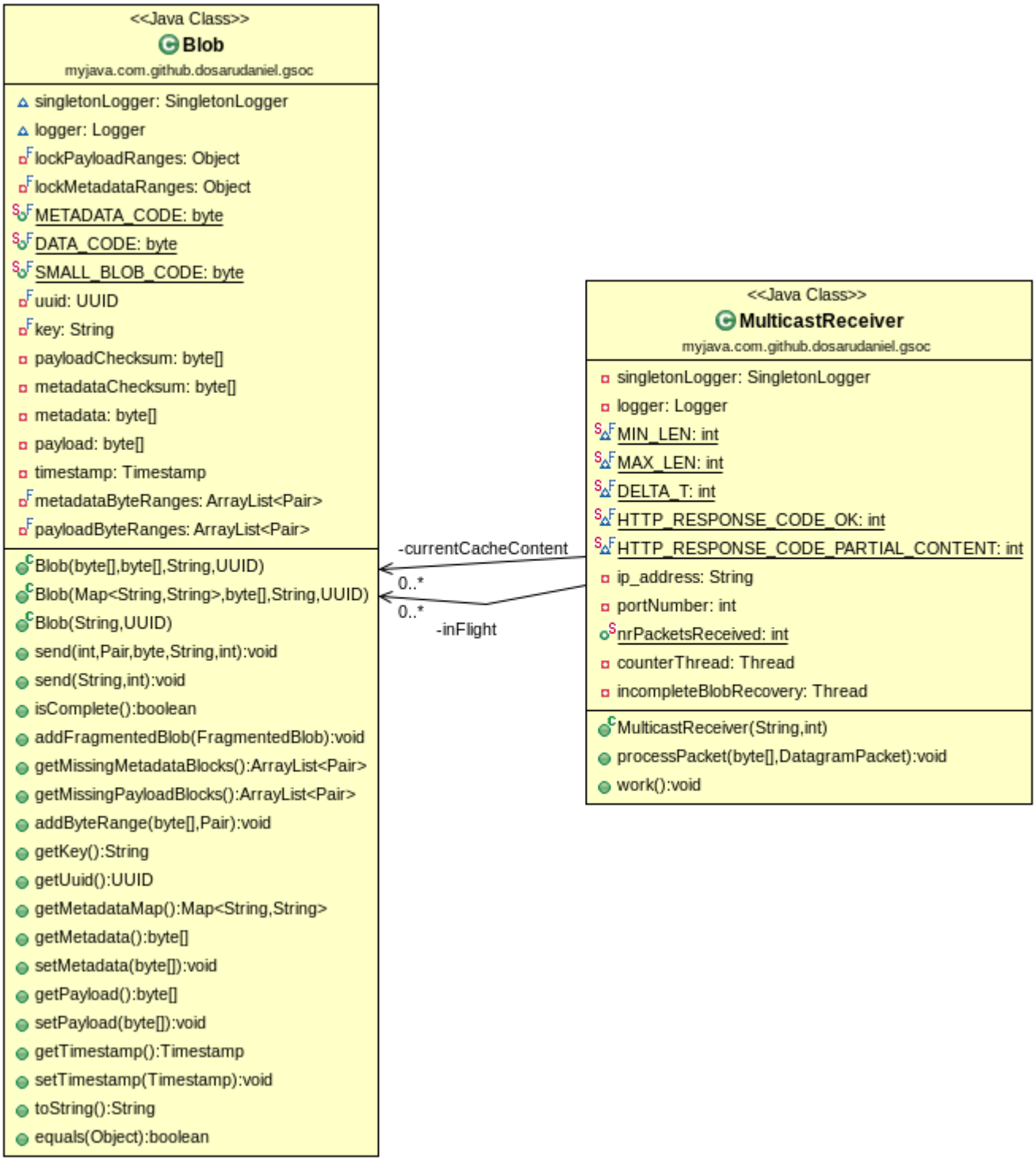

Figure 15: Multicast Receiver and Blob classes

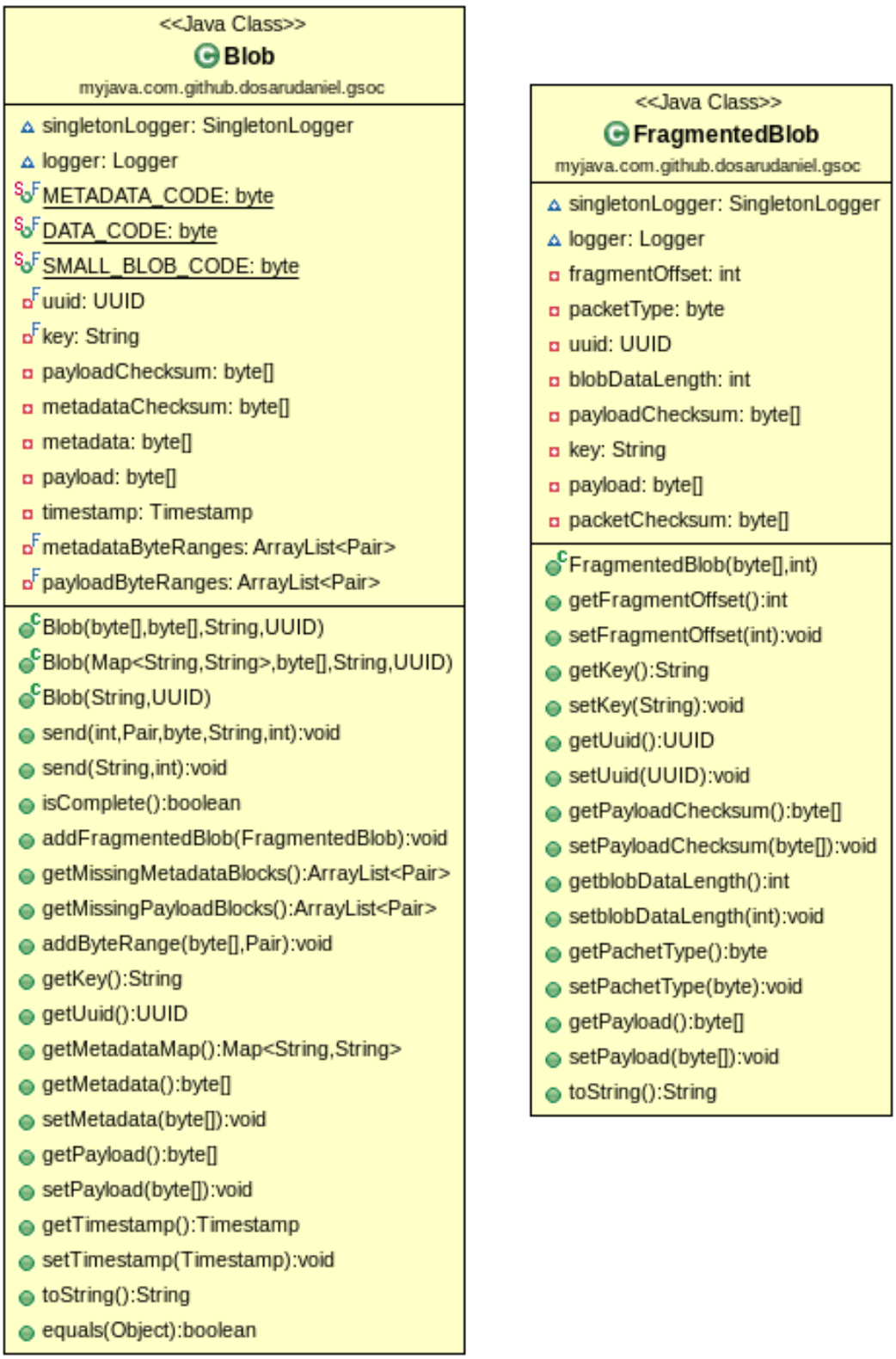

Figure 16: Blob and Fragmented Blob classes has no inheritance relation because the FragmentedBlob class is use only for serialization and deserialization

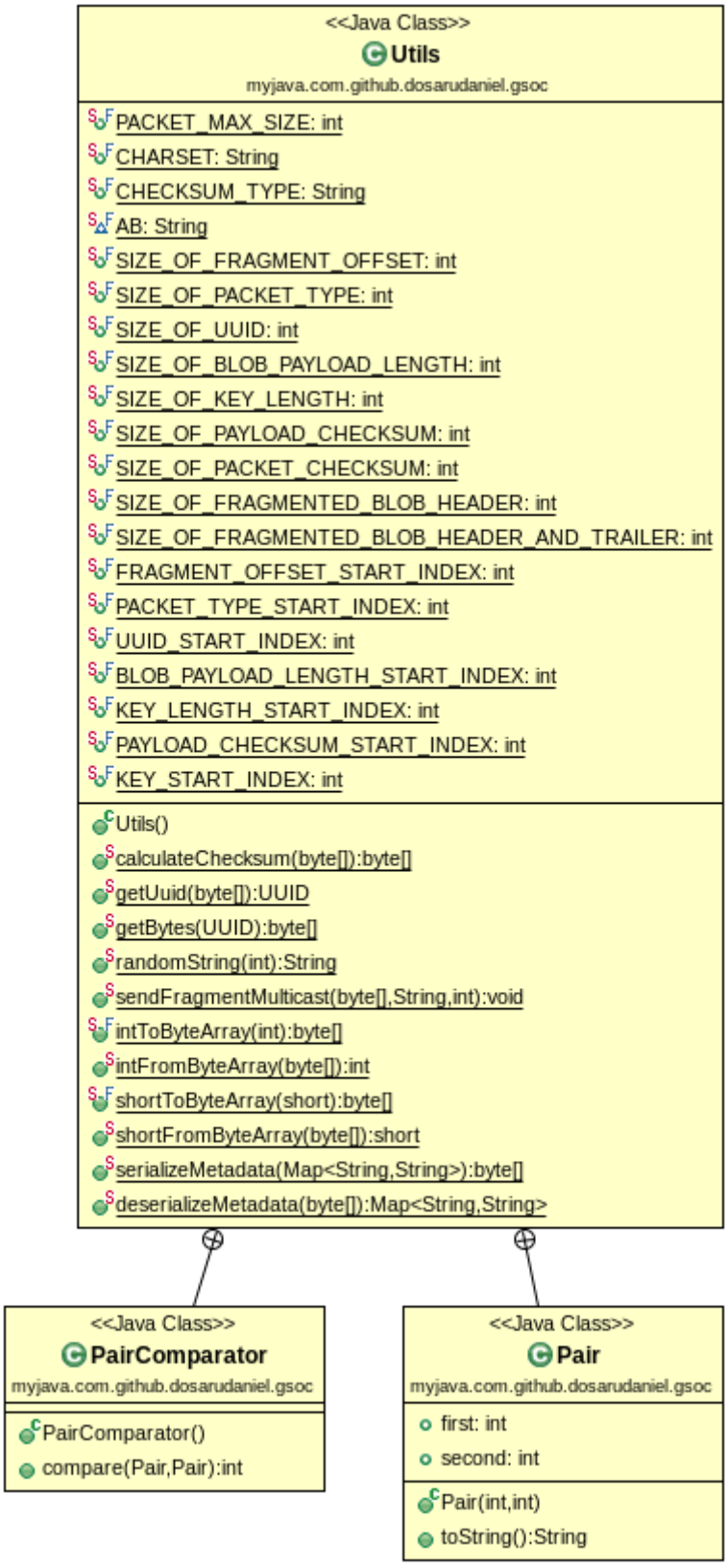

Figure 17: Utilities functions and constants

D MONALISA - MONITORING NETWORK TRAFFIC

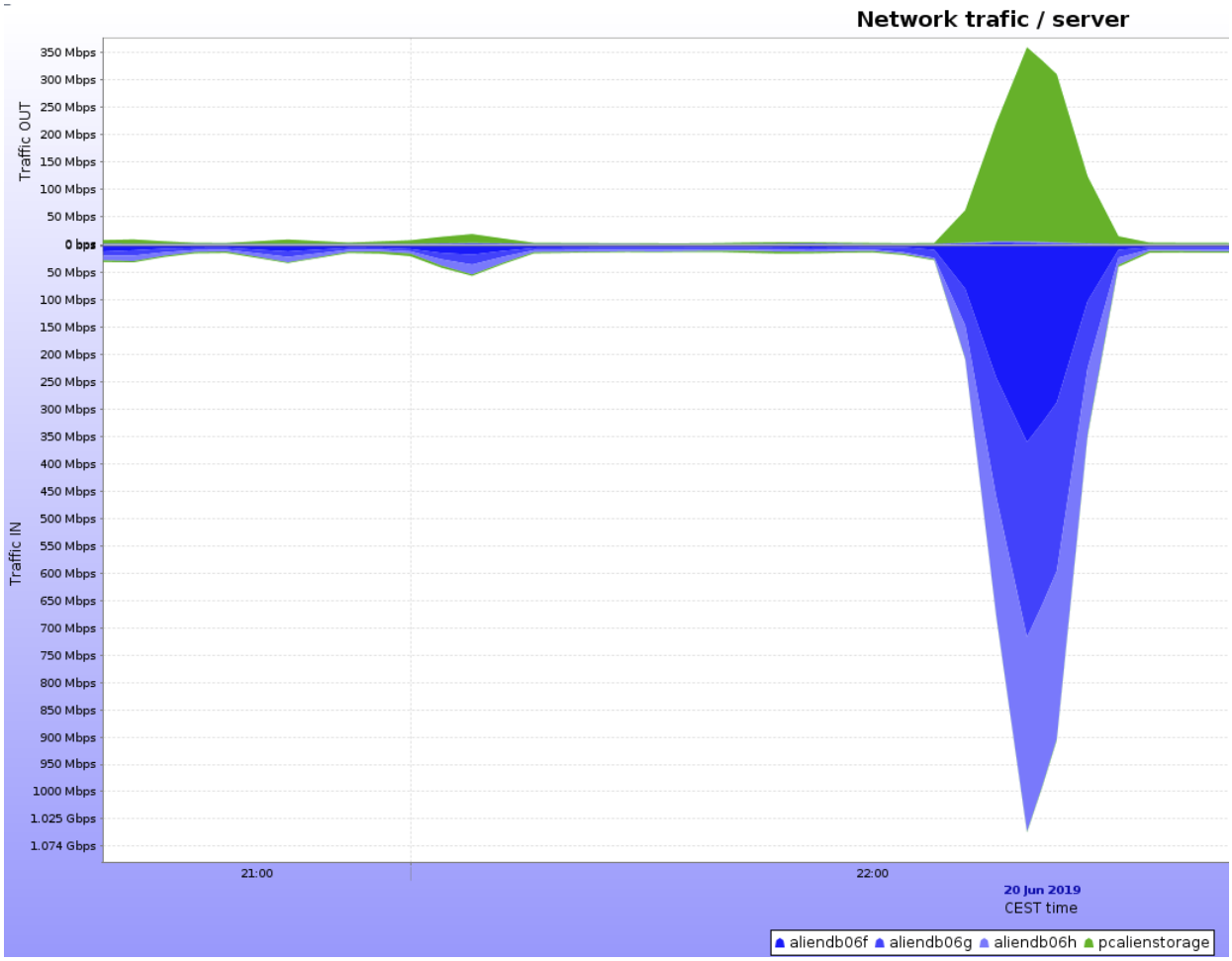

Figure 18: A short burst of random content 2 MB Blobs sent from **pcalienstorage** *to* **aliendb06h**, **aliendb06g** *and* **aliendb06f**: *Traffic IN = 3 * Traffic OUT*

. . .