Niko Haapalainen

Multiuser support for an application processing 5G TTI trace data

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor's Thesis

4 May 2021

Abstract

Author:	Niko Haapalainen
Title:	Multiuser support for an application processing 5G TTI trace data
Number of Pages:	27 pages + 2 appendices
Date:	4 May 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Smart Systems
Instructors:	Marko Uusitalo, Senior Lecturer

The goal of this study was to develop multiuser support into an application prebuilt on the Apache Spark framework on a corporate cloud, processing 5G TTI trace signal capture data. Initially, the first version of the prebuilt application was made to support a single user at a time. The study was commissioned by Nokia Corporation, and the application was developed for the testers working in the 5G development sector in the corporation.

This study was conducted by getting acquainted with the Apache NiFi and Apache Zeppelin tools coming along with the Apache Spark framework, in which the prebuilt application was developed. In addition, the properties and capacities of these tools were studied, finding solutions by planning, drafting and testing various kinds of NiFi flow applications and investigating the possibilities of the Scala programming language. The TTI trace data expertise required by this study was gained by interviewing a product architect employee working at Nokia Networks.

A multiuser solution was found by adding several NiFi processors to the prebuilt NiFi flow, modifying the already existing NiFi processor attribute data and adding a quota to one of the NiFi connectors. On the Apache Zeppelin side, changes were done to the existing code, written in the Scala programming language, by adding loops and by altering the file input paths.

As a result, a multiuser support solution was developed on top of the previous single user support. The project conducted may boost the 5G development by encouraging a wider use of the applications developed with the powerful computation engine of Apache Spark in the company, ultimately leading to offering better 5G services to the customers and consumers.

Keywords: 5G, data processing, multiuser support

Tiivistelmä

Tekijä:	Niko Haapalainen
Otsikko:	Monikäyttäjätuen kehitys 5G TTI trace -kaappausdatan prosessointisovellukseen
Sivumäärä:	27 sivua + 2 liitettä
Aika:	4.5.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Smart Systems
Ohjaajat:	Lehtori Marko Uusitalo

Insinöörityön tavoitteena oli kehittää monikäyttäjätuki Apache Spark -ohjelmistokehykseen valmiiksi kehitetylle 5G TTI trace -signaalikaappausdataa prosessoivalle ohjelmistotyökalulle, joka oli kehitetty ensimmäisessä versiossaan tukemaan vain yhtä loppukäyttäjää yhdelle suoritusprosessilleen kerrallaan. Työn tilaajana on Nokia Oyj ja lopputyössä kehitetty työkalu tuli yrityksessä työskentelevien signaalitestaajien käyttöön.

Insinöörityö toteutettiin ensin perehdytyksellä Apache Sparkin ekosysteemiin kuuluvien Apache NiFi ja Apache Zeppelin -ohjelmistojen käyttöön, tutkimalla mainittujen ohjelmistojen ominaisuuksia ja kapasiteettejä, etsien ratkaisuja suunnittelemalla, luonnostelemalla ja testaten eri NiFi-virtausratkaisuja ja tutkien Scala-ohjelmointikielen mahdollisuuksia. Insinöörityön TTI trace -datan tietotaitoa tukee Product Architect Henrik Liljeström, joka työskentelee Nokia Networksilla ja on ekspertti TTI trace -datan analysoinnissa.

Monikäyttäjäratkaisu löydettiin lisäämällä NiFi-prosessoreita jo valmiiksi rakennettuun NiFi-virtaukseen, muuttamalla jo olevien NiFi-prosessoreiden attribuuttitietoja ja lisäämällä yhteen NiFi-liittimen kiintiöön maksimimäärän ylävirtauksesta saapuville FlowFile-tiedostoille. Apache Zeppelin -ohjelmiston puolella lisättiin muutoksia Scala-ohjelmistokielellä kirjoitettuun ohjelmaan lisäämällä lähdekoodiin toistorakenteita ja eri tiedostosyöttöpolut.

Insinöörityön tuloksena saatiin kehitetyksi vakaa monikäyttäjätuki, joka palvelee useaa loppukäyttäjää samanaikaisesti edellisen yksikäyttäjätuen sijaan. Tarkoituksenaan insinöörityö edesauttaa laskentateholtaan nopeaa Apache Spark -ohjelmistokehyksen käyttöönottoa yrityksen kehitystyöhön, monipuolistaen ja tehostaen 5G-kehitystä ja mahdollistaen viime kädessä huippulaatuiset 5G-palvelut yrityksen asiakkaille ja kuluttajille.

Avainsanat: 5G, dataprosessointi, monikäyttäjätuki

Contents

List of Abbreviations

1 Introduction 1

2 Development materials 2

2.1 TTI traces 2
2.2 Apache Spark 3
2.3 Apache NiFi 4
2.4 Apache Zeppelin 5
2.5 Apache Parquet 5
2.6 Amazon S3 6
2.7 Intranet shared storage drive (ISSD) 6

3 The prebuilt application 7

3.1 Overview 7
3.2 File input 7
3.3 The process in Apache NiFi 8
3.3.1 The unpacker processor group 8
3.3.2 The Zeppelin processor group 9
3.4 The process in Apache Zeppelin 10

4 The multiuser support solution 14

4.1 Building the multiuser solution 14
4.2 The multiuser solution in the Apache NiFi side 18
4.3 The multiuser solution in the Apache Zeppelin side 22

5 Results and discussion 23

6 Conclusion 26

References 27

Appendices

Appendix 1: tti_trace_processor.scala

Appendix 2: tti_trace_processor_mu.scala

List of Abbreviations

5G:	Fifth generation telecommunication networks technology
AI/ML/DL:	Artificial intelligence, machine learning and deep learning
API:	Application programming interface
CSV:	Comma separated values
DF:	Dataframe
gNB:	Next generation Node B
ISSD:	Intranet shared storage drive
NR:	New Radio
QoS:	Quality of service
R&D:	Research and development
S3:	Amazon Simple Storage Server
TTI:	Transmission timing interval
UE:	User equipment

1 Introduction

Telecommunications are rapidly shifting to the era of the fifth generation of mobile networks technology standard known as 5G. According to an estimate of the financial comparison site Bankr.nl [1], 53% of the earth's population, 4.14 billion, will have access to 5G services by 2025. The research and development (R&D) sector is challenged and all solutions enabling automation, scalability and performance are highly appreciated in the 5G development in telecom companies.

The concept of transmission timing interval (TTI) traces is at the core of this project. TTI traces are a type of log data of TTI events containing performance messages captured from user equipment (UE) data transmissions. By analyzing TTI trace data, valuable information can be acquired, such as locating the cause for throughput degradation or discovering a combination of parameter settings enabling high and stable performance in the system, ultimately leading to added value in the company.

This thesis describes a project, where the automation for the research of TTI trace data was enhanced by developing further a prebuilt TTI trace processing application running in the Apache Spark environment. After this introduction, a technical review of the concept of TTI traces is provided along with the software tools used for the development. The current single end user solution is explained, continuing to the review of the newly developed multiuser support solution. Finally, the user experience data regarding the new solution and the challenges encountered during the development process are discussed, resulting in drawing conclusions based on the project.

This project was commissioned by Nokia Corporation, which meant that some company internal information could not be disclosed for some parts of this thesis. In addition, the study has been supported by telecom research literature and by an interview with Product Architect Henrik Liljeström working for the Nokia corporation.

2 Development materials

2.1 TTI traces

The application developed and documented in this thesis processes TTI trace data. In networking and telecommunications, *transmission time* or *transmission delay* is the time for the transmitter node to push all of the packet's bits into the outgoing data link [2]. *Transmission Timing Interval*, abbreviated as TTI, defines the time period between new data transmissions over the air interface [3]. For 5G New Radio (NR) networks, the logical next-generation node B (gNB) radio node functions as the transmitter node unit and its TTI depends on transmission numerology [3] as indicated in Table 1. The term of numerology can be defined as the configuration of waveform parameters in telecommunications [4].

Table 1. The supported transmission numerologies and their respective TTIs in 5G NR technology. Data gathered from [3].

Waveform configuration (Subcarrier spacing)	TTI
15kHz	1 ms
30kHz	0.5 ms
60kHz	0.25 ms
120kHz	0.125 ms
240kHz	0.0625 ms

The output of a TTI trace is then a log file captured by monitoring the events during each TTI in the transmitter node [3]. The captured data can be then used for post-analysis. The log files are output into CSV files and then archived and gzip-compressed into tar.gz files for smaller data size and easier portability. Typically, the tar.gz package is of size 300MB to 500MBs. The end user inputs these archive files into the application, and the application requires the archive

files to be in the tar.gz format. Other archive files than tar.gz files will throw an error.

2.2 Apache Spark

The application developed further in this project is based on several Apache Software Foundation projects. The core software used is the analytics engine of Apache Spark (ver. spark2.4.3-hadoop2.7), which enables lightning-fast data processing speeds due to the distributed and cached data in-memory cluster computing system [5]. As indicated in Figure 1, SparkContext invokes the cluster manager, which orchestrates and distributes the workload to the worker nodes, achieving its fast data processing speeds. This makes Apache Spark a worthwhile research platform for big data and AI/ML/DL projects.

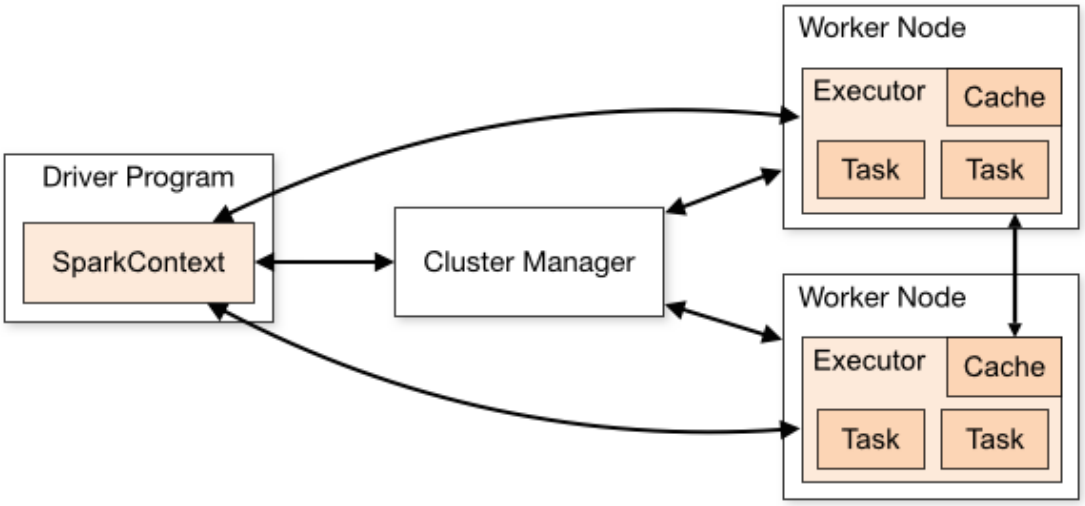

Figure 1. Cluster computing architecture in Apache Spark. Copied from Apache Spark homepage [6].

Apache Spark was chosen for the project due to its fast data processing capabilities and the possibility to bring it into use in 5G development in the corporation. Its computational capabilities are utilized for processing the TTI trace data on the Apache Zeppelin notebook. While Apache Spark offers the

main framework for the project, the tools in Apache Spark's ecosystem are utilized to perform the handling and processing the data as specified below.

The other Apache software and services used in the project are listed below:

- Apache NiFi
- Apache Zeppelin
- Apache Parquet
- Amazon Simple Storage Server (S3)
- Intranet shared storage drive, ISSD.

These tools were chosen for the project simply because the previous prebuilt application prior this thesis, created by another developer, was conducted by using the listed tools. The developer mentioned has exited the corporation already and there is no further information why these tools were chosen. In this project, development was continued from where it was left with the materials at hand.

2.3 Apache NiFi

Apache NiFi is web-based control panel software for visually monitoring data flow processes [7]. The end user builds "NiFi flows" to create workflows from NiFi processors, each of them having some sort of a dedicated task.

In this project, the application running on NiFi (ver. 1.12.1) fetches the input TTI trace tar.gz archive files to its downstream flow from the ISSD intranet shared storage drive, unpacks the archive files and uploads the CSV files extracted from the archive files to the temporary folder on Amazon S3 bucket. Eventually the application in NiFi invokes the notebook application on Apache Zeppelin for further processing the CSV TTI trace data.

The NiFi version 1.12.1 is the version downloaded upon the commissioning of Apache NiFi in the corporation. The latest version 1.13.1 does not have any improvement for the processors that are utilized in the NiFi flow [12] and upgrading would not give any extra value for the project.

2.4 Apache Zeppelin

Apache Zeppelin is web-based notebook programming software, which is used for scientific research involving data visualization and machine learning tasks [8]. It is alternative notebook software to its more popular counterpart, Jupyter Notebook.

In this project, Apache Zeppelin (ver. 0.8.2) was used for importing the CSV files containing TTI trace data from the Amazon S3 bucket, processing the data and finally appending the processed data to the existing Apache Parquet file back on the Amazon S3 bucket. The processing code was written in Scala, which is executed by using the Apache Spark's dedicated interpreter, the Spark interpreter, to create the SparkContext as indicated in Figure 1. When running the Scala program, the Spark interpreter is called with the statement *%spark* at the beginning of the code as seen in appendices 1 and 2.

2.5 Apache Parquet

Apache Parquet is a column-oriented file storage format. Initially it was introduced for the predecessor of Apache Spark, Apache Hadoop, but it is also available for the use of Apache Spark, Spark being a part of Hadoop's ecosystem [9]. Due to its column-oriented data structure, the parquet file format offers fast read-write data speeds for large-scale data processing tasks.

In this project, a parquet file was used as a storage for the processed data of TTI trace files on the Amazon S3 bucket.

2.6 Amazon S3

Amazon S3 is an object-oriented cloud storage service. The basic unit for containing data in S3 is called a “bucket” which contains the desired files in it for storage. [10.]

In this project, Amazon S3’s bucket will host the temporary CSV files waiting for input to the Apache Zeppelin as well as host the output Apache Parquet file for further analysis.

2.7 Intranet shared storage drive (ISSD)

The intranet shared storage drive (ISSD) is a shared cloud storage drive for the end users to host their captured TTI trace file data prior the input to the processing application as presented in the project. The ISSD is the default output target when capturing TTI trace data.

3 The prebuilt application

3.1 Overview

In this project, the task was to develop multiuser support to an application prebuilt on the Apache Spark's ecosystem. The prebuilt application supports a single end user's single tar.gz input file at a time. The objective was then to find a solution to how to make the prebuilt application support multiple file input from multiple end users simultaneously. This chapter presents the workflow of the prebuilt application to explain the starting point of the development as indicated in Figure 2.

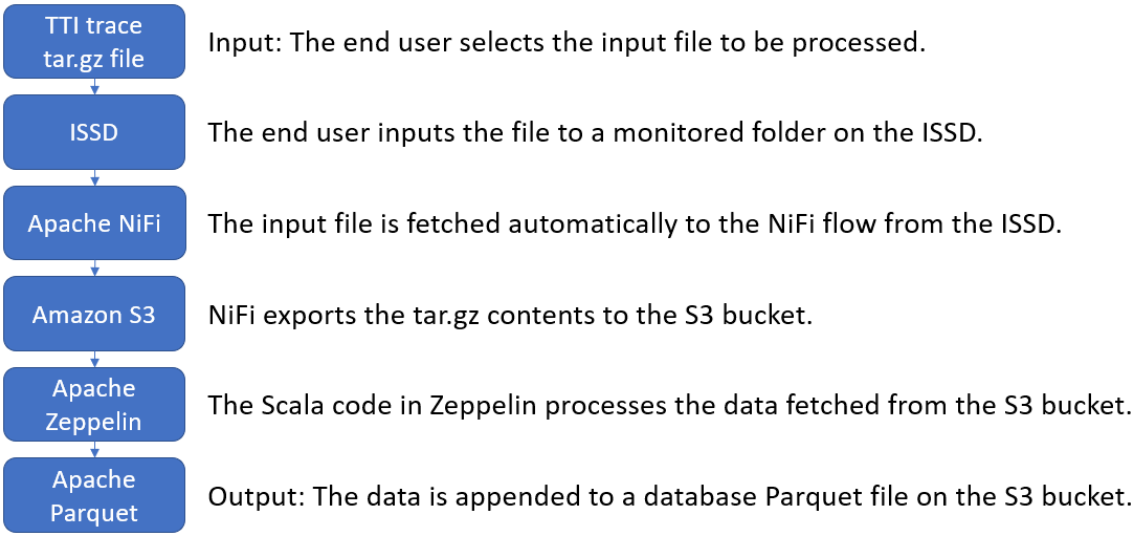

Figure 2. The workflow of the prebuilt application.

3.2 File input

At the beginning of the workflow, as indicated in Figure 2, the end user chooses a TTI trace tar.gz file to be processed and appended to the database parquet file for further analysis. The ISSD contains a folder, and by moving a TTI trace tar.gz file into that folder, the end user invokes the workflow for the NiFi app to fetch those files in the folder for its downstream flow, commencing the project's automated workflow process. There is no need for the end user to transfer the

large-sized tar.gz files out from the ISSD, making the use of the application simple, fast and time-saving for the end user. There are no further actions required from the end user during the workflow process.

3.3 The process in Apache NiFi

Before all, it is necessary to explain the difference between Apache NiFi processors and the NiFi processor groups. The NiFi processors are basic data processor units whereas processor groups may contain many NiFi processors for management reasons. In the prebuilt application, there are two processor groups set. The first processor group functions as the fetcher for the input tar.gz file and unpacks the TTI trace CSV files from the tar.gz file into the temporary “tti_temp” folder located on the Amazon S3 bucket. The second processor group invokes the Zeppelin notebook for further processing of the CSV data. This setup is illustrated in Figure 3.

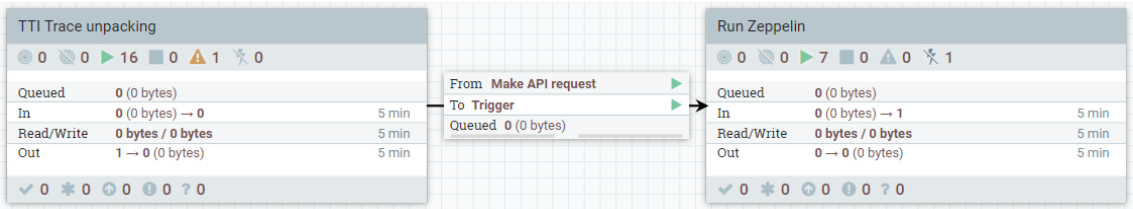

Figure 3. The two NiFi processor groups set for the prebuilt application. Screenshot [13].

3.3.1 The unpacker processor group

For the ease of reading, the “TTI Trace unpacking” processor group as seen in Figure 3 will be noted as the unpacker processor group. The unpacker processor group fetches the input tar.gz file from the ISSD input folder and imports it to the NiFi flow. When successfully imported to the flow, the imported tar.gz file is now of a FlowFile type format, which is the basic data format for the data travelling between the NiFi processors [7].

Subsequently, the input tar.gz file is decompressed and unpacked. As a product, there are now several FlowFiles downstreamed in the flow, each of them representing a singular CSV file extracted from the tar file. Finally, the resulting CSV files are uploaded to Amazon S3 bucket's temporary "tti_temp" folder for further processing. After the upload, the multiple FlowFiles are merged into a single FlowFile in the "MergeContent" processor, which is then forwarded to the output node of the unpacker processor group as indicated in Figure 4.

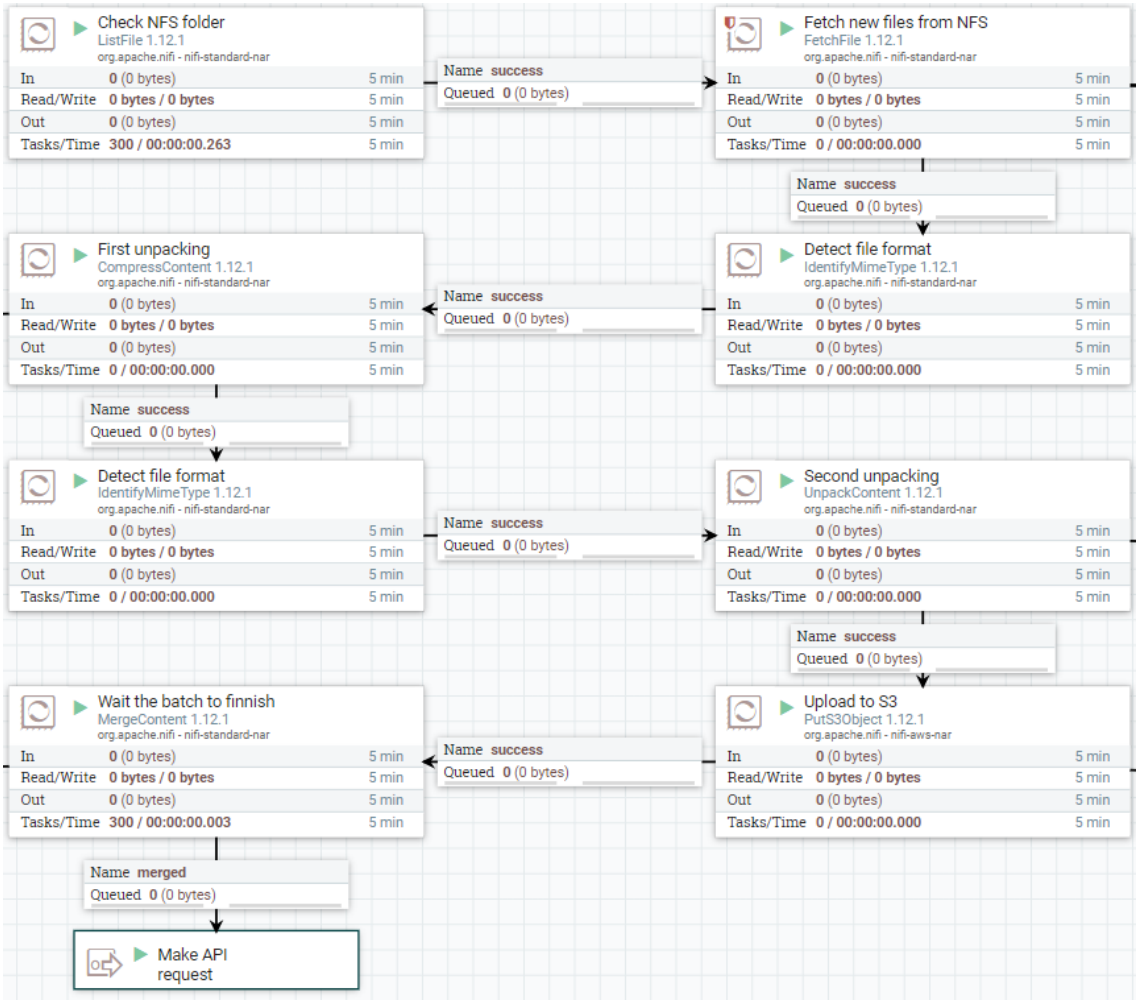

Figure 4. The flow inside the unpacker processor group. Screenshot [13].

3.3.2 The Zeppelin processor group

The "Run Zeppelin" processor group as seen in Figure 3 will be noted as "the Zeppelin processor group". This processor group invokes the Scala code in the Zeppelin notebook. Once the previous unpacker processor group has finished

its task, it triggers the flow in the Zeppelin processor group. The workflow assigns the Zeppelin Notebook credentials to the incoming FlowFile's content, connects to the Zeppelin notebook API, saves the session ID to the FlowFile as an attribute and then invokes the Zeppelin notebook. Ultimately, depending on the response from the API call, the NiFi flow terminates either as a success or as a failure in either of the "LogAttribute" processors. This process is indicated in Figure 5.

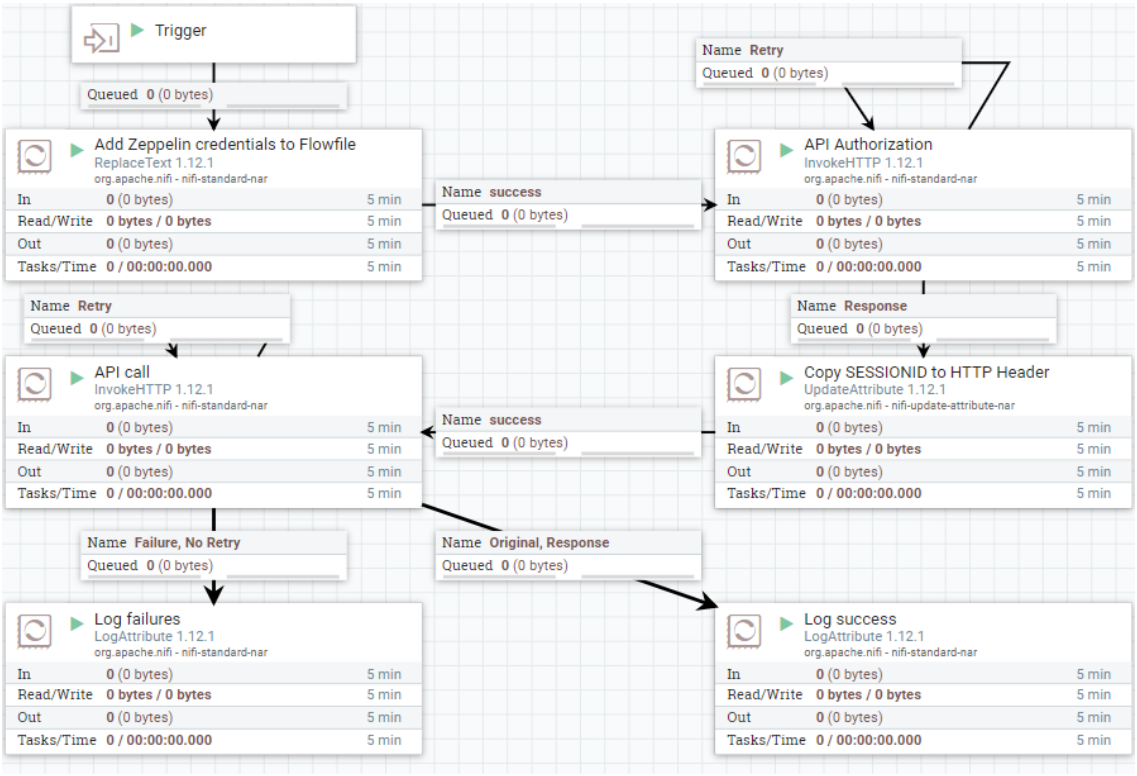

Figure 5. The flow inside the Zeppelin processor group. Screenshot [13].

3.4 The process in Apache Zeppelin

The Scala code application built to the notebook on Apache Zeppelin processes the CSV files previously uploaded on the Amazon S3 bucket's temporary "tti_temp" folder. Eventually the processed data is outputted and appended to an existing parquet file hosted on the S3 bucket, which functions as a database file for storing the processed TTI trace data for further analysis.

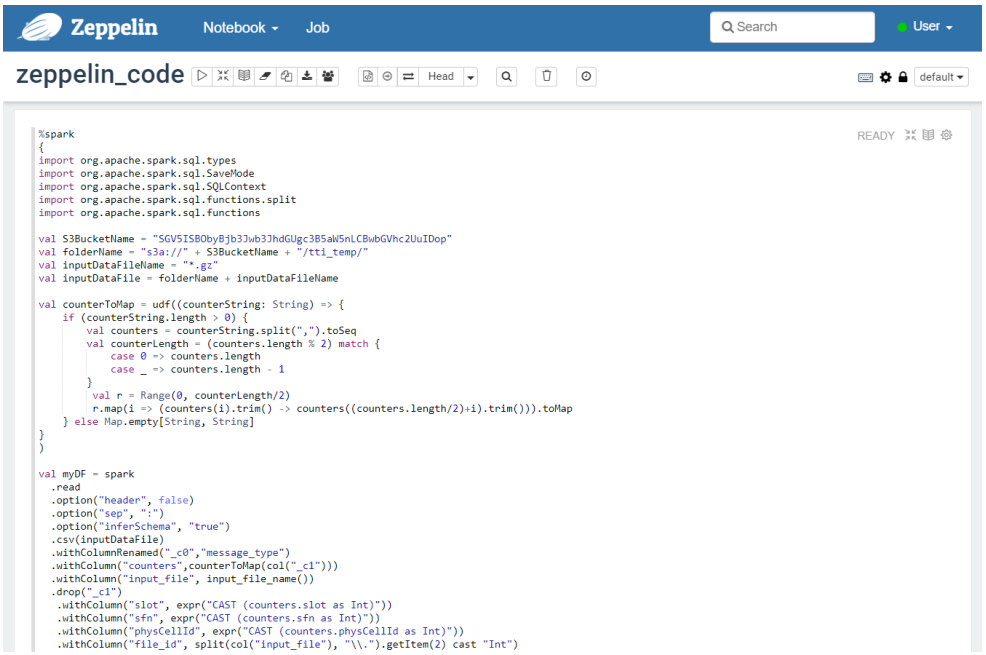

Figure 6. An overview of the Apache Zeppelin notebook interface. Screenshot [14].

The process in the Zeppelin paragraph is straightforward. The Scala code fetches the filenames of the uploaded CSV files on the Amazon S3 bucket's "tti_temp" folder. The fetching is implemented by using Scala's wildcard character (*) as seen in Listing 1.

```
val S3BucketName = "SGV5ISBObyBjb3Jwb3JhdGUgc3B5aW5nLCBwbGVhc2UuIDop"
val folderName = "s3a://" + S3BucketName + "/tti_temp/"
val inputDataFileName = "*.csv"
val inputDataFile = folderName + inputDataFileName
```

Listing 1. The Scala code, which fetches the CSV files from the S3 bucket (see Appendix 1).

After the filenames are fetched for the "inputDataFile" immutable variable, the next part is to start processing them. The dataframe (or DF) "myDF" is initialized and the CSV files are inputted to the dataframe. The modifying of the dataframe begins by renaming and dropping unnecessary columns in the CSV files as indicated in Listing 2.

```
val myDF = spark
  .read
  .option("header", false)
  .option("sep", ":")
  .option("inferSchema", "true")
  .csv(inputDataFile)
  .withColumnRenamed("_c0","message_type")
  .withColumn("counters",counterToMap(col("_c1")))
  .withColumn("input_file", input_file_name())
  .drop("_c1")
  .withColumn("slot", expr("CAST (counters.slot as Int)"))
  .withColumn("sfn", expr("CAST (counters.sfn as Int)"))
  .withColumn("physCellId", expr("CAST (counters.physCellId as Int)"))
  .withColumn("file_id", getFileId(col("input_file")))
  .filter('message_type === "dlFdSchedData" ||
          'message_type === "ulFdSchedData" ||
          'message_type === "ulPuschReceiveRespPsData" ||
          'message_type === "ulUtcTimestampData" ||
          'message_type === "dlBeamData" ||
          'message_type === "ulBeamData")
  .filter('file_id >0)
```

Listing 2. The initialization of the dataframe successful with renaming and dropping of unnecessary columns (see Appendix 1).

Most often, the CSV input data contains missing timestamps. Therefore, these timestamps are generated by computing approximations from adjacent timestamps as seen in Listing 3. These timestamps must be calculated, since a raw timestamp is given every 30 seconds during the capture, which would not be enough to carry out precise post-analysis for the data as such. One cannot distinguish which timestamps are from the capture data and which are generated. One can compare the timestamps between generated timestamps and the timestamps found in the CSV data to locate the original timestamps.

```
val myDfWithTime = myDF.join(myDfWithRefTimeSfn, Seq("file_id"))
        .withColumn("time_add_ms", when('ref_sfn >= 350, when('sfn <  …
            .otherwise(('sfn - 'ref_sfn)*10 + ('slot - 'ref_slot)/2))   …
        .withColumn("time_ms", 'ref_time_ms + 'time_add_ms)
        .withColumn("hours", floor('time_ms / 3600 / 1000).cast(StringType))
        .withColumn("minutes", floor(('time_ms - 'hours * 3600 * 1000 …
        .withColumn("seconds", floor(('time_ms - 'hours * 3600 * 1000 …
        .withColumn("milliseconds", floor(('time_ms - 'hours * 3600 * …
          …
        .drop('milliseconds10)
        .drop('microseconds100)
        .drop('ref_time_ms)
        .orderBy('time_ms)
        .filter('message_type === "dlFdSchedData" ||
                'message_type === "ulFdSchedData" ||
                'message_type === "ulPuschReceiveRespPsData" ||
                'message_type === "dlBeamData" ||
                'message_type === "ulBeamData”)
```

Listing 3. The timestamp approximation process. Cropped. (see Appendix 1).

After the timestamp generation process, the processed data in the dataframe will be output and appended to the parquet file hosted on the Amazon S3 bucket. The CSV files located in the S3 bucket's temporary "tt_temp" folder are deleted as seen in Listing 4. The whole process for the prebuilt application is thereby completed.

```
myDfWithTime
  .repartition(1)
  .write
  .mode("append")
  .parquet(s"s3a://$S3BucketName/parquet.parquet")

val delfile = FileSystemUtil(spark).delete(folderName)
```

Listing 4. The output process of the Scala code (see Appendix 1).

The output parquet file can be then used for further TTI trace analysis by visualizing the parquet file's data in the data visualization software of Apache Superset for locating anomalies and errors, for example. This supports the testers in the 5G development to debug the performance of the 5G signals in order to fix them for future 5G releases, resulting in quality 5G services for the customers.

On the performance side, the Scala code performs relatively fast through the Apache Spark's Spark interpreter. When Zeppelin invokes the code, it takes approximately from one to three minutes to process the multiple CSV files contained in a single input tar.gz file. This is a relatively well performing result, as a single CSV file may contain hundreds of thousands of lines of captured TTI event data. Running the process on the cluster computing engine of Apache Spark on the cloud is more optimal than running the process on each end user's workstation locally, where the process time could be expected to be manifold.

4 The multiuser support solution

In order to develop the prebuilt application, as presented in chapter 3, into a multiuser support version, there were two major problems to solve:

1. As a convention, each CSV file inside the TTI trace capture tar.gz package had the same filename, which was distinct from each other by using an index number as a suffix in the filename. As a result, an important question was how to distinct each CSV file from each other and prevent the files from getting mixed up when there are multiple tar.gz packages and their CSV files being uploaded to the “tti_temp” folder.

2. When should the workflow invoke the processing code of the Zeppelin notebook if multiple users are using the application simultaneously?

To find a solution to each problem, each tool taking part in the workflow had to be modified. In Apache NiFi, the two processor groups were to be merged for better flow monitoring and new processors were to be added to the NiFi flow. In addition, the NiFi connector prior to the processor that invoked the Zeppelin notebook was to be set with a quota, to limit the invoking of the Zeppelin notebook. In Apache Zeppelin, the directory paths were to be edited and loops were to be added to the Scala code. In addition, the filename of each input tar.gz file was to be inserted to the dataframe as a column, to clearly mark and separate the respective input file data from one another when the data has been appended to the database parquet file. In case of inputting an archive file with a filename already existing in the database, the system would throw an error. This error can be overcome by guiding the end user not to input the same files twice to the system.

4.1 Building the multiuser solution

At the initial planning phases on how to implement multiuser support in the prebuilt application, implementing a queueing system was introduced. Apache

NiFi would queue all incoming tar.gz files into their respective folders by the name of their original tar.gz filename on the Amazon S3 bucket's temporary "tti_temp" folder. These folders then contained the tar.gz packages' respective CSV files in them, each folder waiting to be fetched to the Apache Zeppelin code one by one. This solution satisfies case number 1 as noted in the problems marked at the beginning of this chapter. The queueing system seemed to be easy to implement and a logical solution to take forward.

During the planning of setting up the queueing system, there was an intention of keeping the Scala code intact, meaning that no changes would be applied to the Apache Zeppelin side. At this stage, Apache NiFi seemed to be capable enough for executing the queueing system due to its diverse support for different kinds of functions in its processors as indicated in Figure 7. Therefore, the focus of the development turned to the prebuilt application's NiFi flow and the queueing system was decided to be integrated into the workflow.

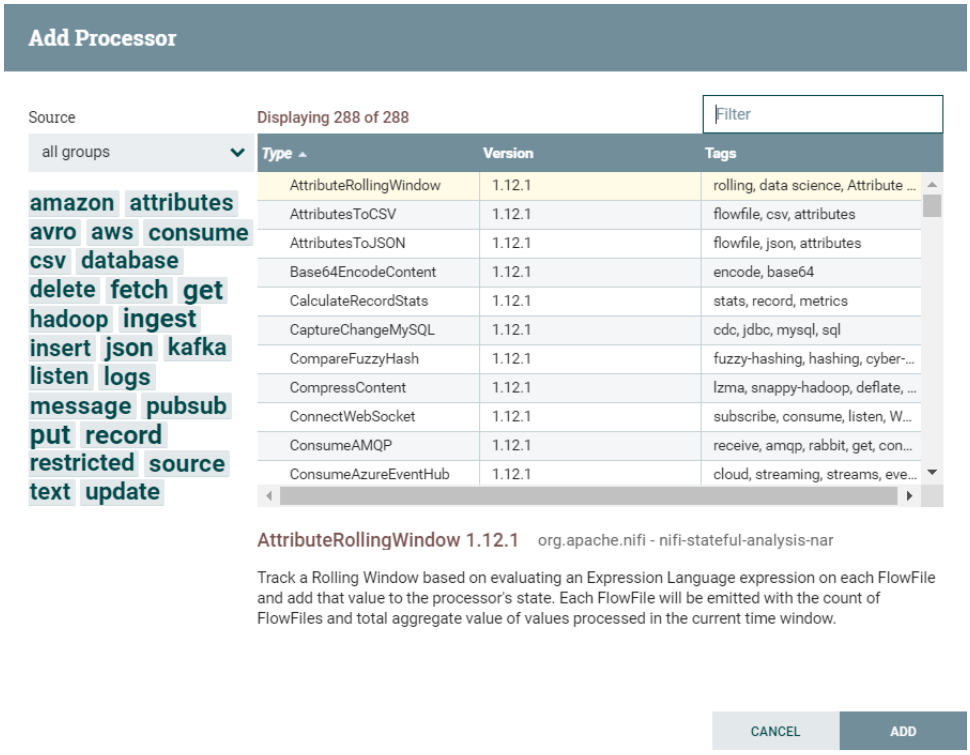

Figure 7. The "Add Processor" menu for selecting and adding new NiFi processors to the flow. Screenshot [13].

However, after several weeks of experimenting with Apache NiFi and its processors, the mentioned plan was discarded due to limited NiFi processor support for S3 operations (see Figure 8). These operations were needed the most for implementing the queue into the prebuilt application. In addition, the NiFi's FlowFile system [11] blocked the possibility of setting up a queueing system efficiently. Apache NiFi is strictly built to the concept of its basic FlowFile file unit, which is not the most flexible option when wishing to create multiuser based simultaneous action pipeline applications. As a result, the focus turned away from Apache NiFi to Apache Zeppelin for finding solutions.

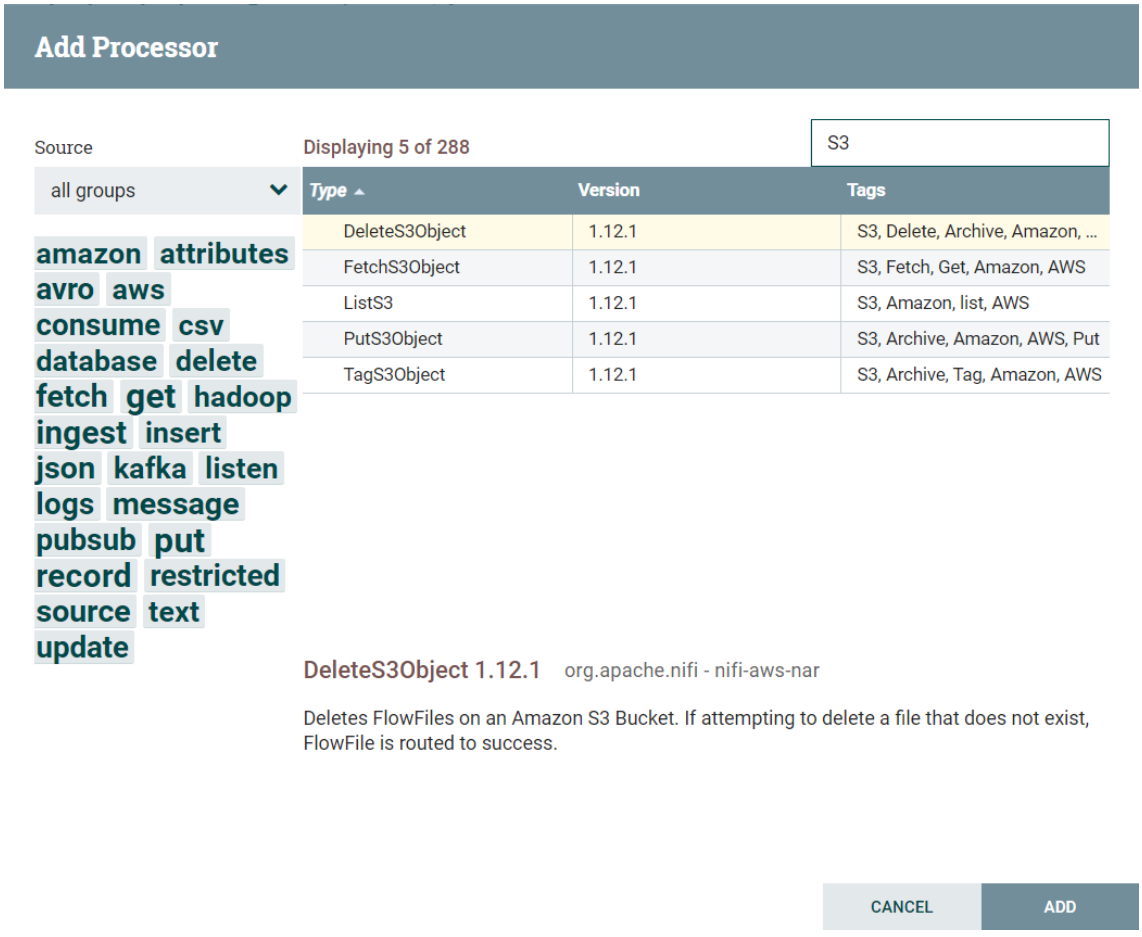

Figure 8. The listing of available NiFi (version 1.2.1) processors for Amazon S3 operations. Screenshot [13].

The experimenting continued in the Zeppelin notebook side. After some research on the prebuilt application's Scala code application, it was realized that a foreach loop was seen to safely iterate through all the detected input files uploaded to the Amazon S3 bucket's temporary "tti_temp" folder in one-by-one action as indicated in Listing 5. This way of working also solved problem number 2 as seen in the problem list at the beginning of this chapter.

However, more optimization and development was required, as there was a possibility for Apache NiFi to invoke the Zeppelin notebook code process ahead of time, when uploading to the S3 bucket's "tti_temp" folder was still in progress, leading to an error. To patch this problem, an NiFi processor was added to the flow (see Figure 11) and another nested while-loop was added to the code (see Listing 5) to force the code to wait until the uploading process had finished before starting the processing of the CSV TTI trace data.

```
for (folderName <- folderNames) {

    println("Starting processing for: " + folderName)
    var waitCount = 0

    breakable {

        while (waitCount < 12) {

            println("Inside while loop...")
            val folderNameContentsStr = FileSystemUtil(spark).list(folderName…

            if (folderNameContentsStr.contains("/ok")) {

                println("Contains string '/ok' -> Now inside if-condition …

                val inputDataFileName = "/*.gz"  // The input csv file (also …
                val inputDataFile = folderName + inputDataFileName

                println(s"Starting: $inputDataFile")
                …
                …
                …
```

Listing 5. The Scala code in Apache Zeppelin containing the newly added loops (see Appendix 2). Cropped (see Appendix 2).

The resulting work was to separate the TTI trace CSV files that were to be uploaded to their own folders. This was implemented by adding a couple of NiFi processors to the prebuilt implementation's NiFi flow, which adds a subpath to the Amazon S3 bucket output folder and the queue line logic was built. The research and development for the multiuser solution was then finished.

4.2 The multiuser solution in the Apache NiFi side

In this subchapter, the developed multiuser solution is presented for the NiFi side. The two separate processor groups are merged to a single processor group, making it easier for the developer to monitor the events in the flow. In the unpacking section of the merged flow, previously noted as the unpacker processor group in the prebuilt application, an "UpdateAttribute" processor is added to the flow. The processor saves the original filename of the input tar.gz file and takes a timestamp of the current time as an attribute for the downstreamed FlowFile. The filename and the timestamp will be used for separating the uploaded files on the S3 bucket's temporary "tti_temp" folder. It is necessary to save the original filename at this point, since the original filename will be lost once the FlowFile is processed through the "CompressContent" processor. The updated unpacker flow section can be seen in Figure 9.

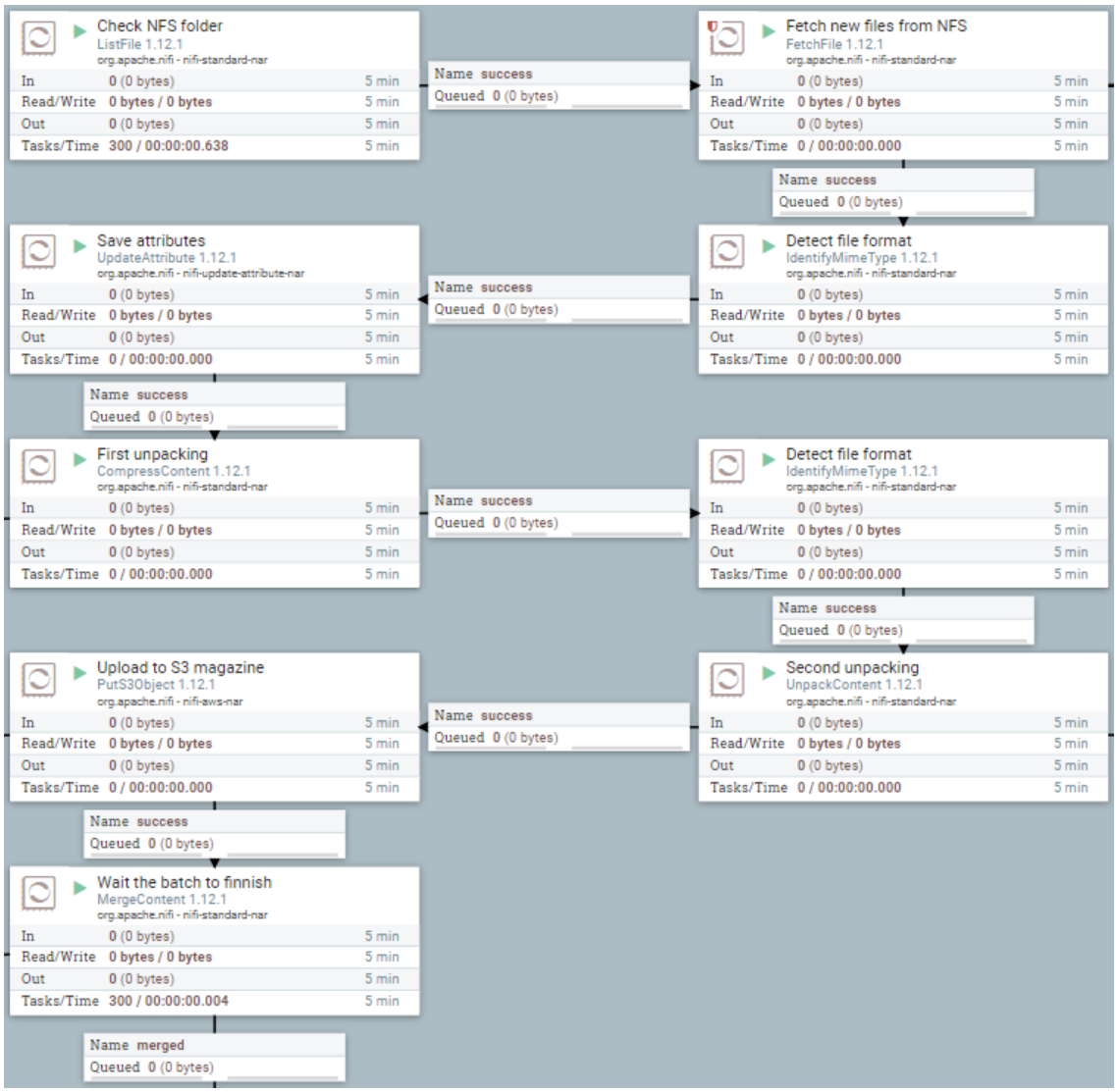

Figure 9. The updated solution in the unpacking section of the NiFi flow. The section that invokes the Zeppelin notebook has been cropped from this figure. Screenshot [13].

As for the Zeppelin notebook invoker flow, previously noted as the Zeppelin invoker processor group, there are two changes committed. The connector preceding the “API call” titled processor is modified by limiting the connector’s quota to take only one FlowFile inside it at once. This is done by editing the value “Back Pressure Object Threshold” in the connector’s settings and selecting the prioritizer option “FirstInFirstOutPrioritizer” in order to serve the newest FlowFile in the queue as seen in Figure 10. This configuration assures that the Zeppelin Notebook will invoke only once per an incoming batch and the

latest end user's input tar.gz file will be processed first rather than the longer ones in queue. This way of working is chosen by presuming that the latest end user is the busiest of all other waiting end users, making the newest end user's input file to be served and processed first.

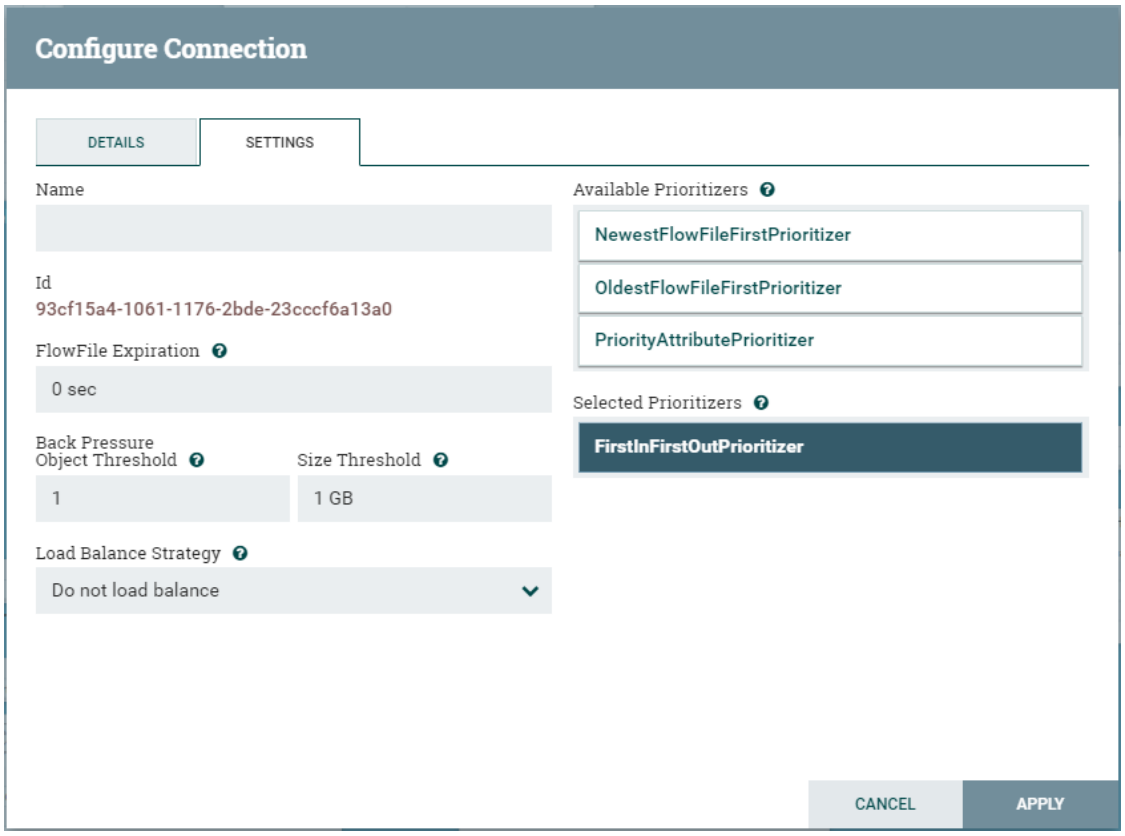

Figure 10. The configuration for applying FlowFile back pressure for the connector preceding the "API call" in the Zeppelin invoker section (see Figure 11). Screenshot [13].

In addition, the Zeppelin invoker section flow also contains a newly added "PutS3Object" processor, which uploads an empty folder titled as "ok" to the temporary folder on the S3 bucket (see Figure 11), functioning as a confirmation for the Apache notebook that the uploading process has finished successfully.

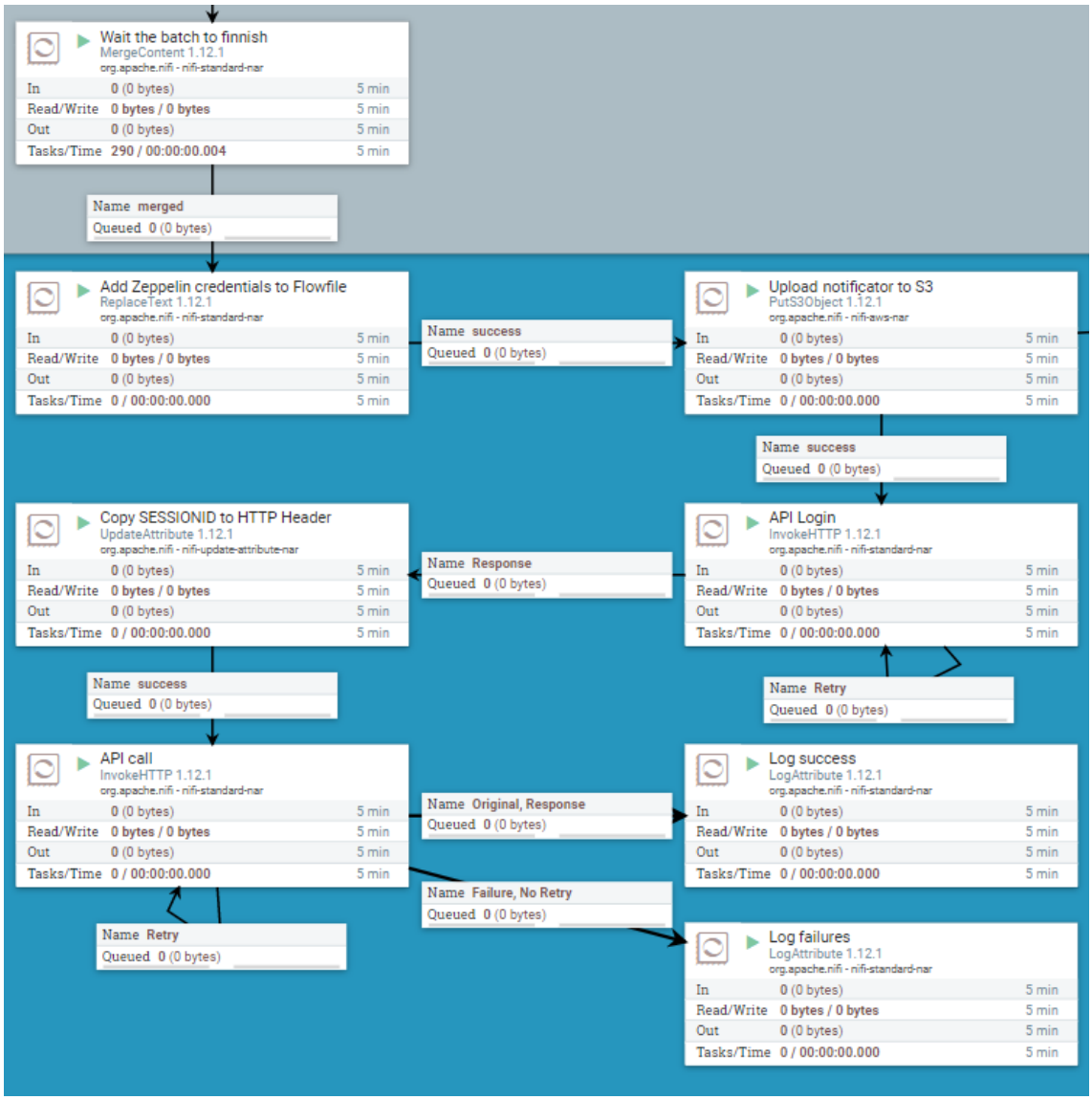

Figure 11. The updated solution in the Zeppelin invoker section of the NiFi flow. This section is a continuation to the unpacking section as seen in Figure 9. Screenshot [13].

These are the changes applied to NiFi. By adding several new processors to the flow and applying several changes to the existing processors, their attributes and their connectors, the simultaneous multiuser capability is enabled.

4.3 The multiuser solution in the Apache Zeppelin side

In this subchapter, the changes committed to the multiuser solution are presented for Apache Zeppelin. The outlook of the Scala code has changed, as the processing code of the prebuilt application described in the Appendix 1 now lies inside the added loops as seen in Listing 5 and new functions have been added to the code. The two new functions have been placed before the start of the foreach loop and they handle the access to the filenames as seen in Listings 6 and 7.

```
//get the file_id
val getFileId = udf((filePath: String) => {
    filePath.split("\\.").reverse(3).toInt
})
```

Listing 6. Retrieving the file_id(s), the unique index number from each uploaded CSV file (see Appendix 2).

Immediately after the “getFileId” variable initialization, the “folderNames” variable is initialized. It contains a list of the detected files on the temporary “tti_temp” folder of the Amazon S3 bucket as indicated in Listing 7.

```
// list of folders (input files)
val folderNames = FileSystemUtil(spark)
    .list("s3a://SGV5ISBObyBjb3Jwb3JhdGUgc3B5aW5nLCBwbGVhc2UuIDop/tti_temp/")
    .toList
```

Listing 7. The newly detected files inside the “tti_temp” folder are assigned to a list to the “folderNames” immutable variable. The hash code is the Amazon S3 bucket’s name (see Appendix 2).

Immediately after the “folderNames” variable initialization, the looping section commences (see Listing 5 and Appendix 2). The foreach loop takes one element from the “folderNames” list for each iteration and processes the data accordingly. The inner while-loop inside the foreach-loop waits and secures until the “ok” folder has been uploaded to the respective input folder on the S3 bucket, confirming that the uploading of all CSVs has been fully finished in the respective input folder on the S3 bucket before continuing the process.

With these additions to Apache NiFi and Apache Zeppelin, the developed application provides a stable multiuser solution to the end users.

5 Results and discussion

Upon finishing the project and taking the multiuser solution in use, an added value was created to TTI trace research and development capabilities in the corporation. In addition, the commissioning of Apache Spark as a part of Nokia's 5G R&D was advanced. For the author of this thesis, a lot of new knowledge and many skills were gained in terms of the Apache family software and how to utilize it for extensive projects.

As the result of the project, the multiuser solution satisfies the objective of expanding the prebuilt application's single-user support into simultaneous multi-user support and the application functions as planned. However, the improved solution may not be the most optimal, as there is a chance for empty iterations in the foreach loop in the Scala code as indicated in Figure 11. This might cause unnecessary stress to the Spark interpreter by activating the interpreter excessively many times, causing the Spark interpreter to be unavailable to other Apache Spark end users most of the time, hence weakening the quality of service (QoS) of Apache Spark for the end users.

It is expected that there will be two to five end users upon the commissioning of the multiuser solution, meaning that there is no immediate need for fixing the optimization problems before the commissioning. The application may be taken into use at the stage it is finished.

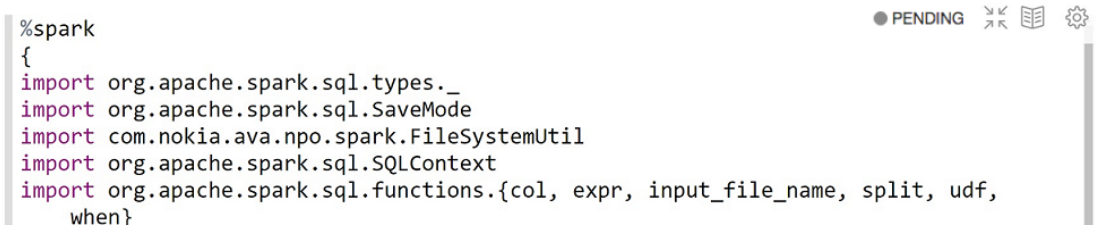

Figure 11. Unnecessary invoking of the Spark interpreter might cause the Zeppelin notebook paragraph to be constantly in "PENDING" mode, waiting for the Spark interpreter to finish its ongoing tasks first. Screenshot [14].

The follow-up discussion is to review the reasons for why the optimization problems occurred. The key challenges were in Apache NiFi. Setting up NiFi flows in Apache NiFi can be regarded as graphical programming, but NiFi programming does not have similar logic as the other traditional programming languages, such as Python or C. This caused difficulties to find proper solutions during the NiFi section development. A lot of criticism can be given to NiFi itself and to its non-flexible concept of FlowFiles travelling between the NiFi processors, which limits its development possibilities. In conclusion, NiFi is a powerful tool, but it may not be the most powerful one for creating pipeline action applications.

The end users' feedback for the multiuser solution tool was positive. The development of the multiuser support was seen to be recompensing. The end users encountered some problems in the ISSD input folder poller "ListFile" NiFi processor, which does not seem to be quite robust to detect the newly placed input tar.gz files in the folder, failing to start the workflow in NiFi. This is a cause of the "ListFile" processor's state-saving functionality as seen in Figure 12. The problem can be solved by guiding the end user to clear the "ListFile" processor manually in the NiFi interface. Despite this, no other problems were perceived, and the developed solution was taken into use.

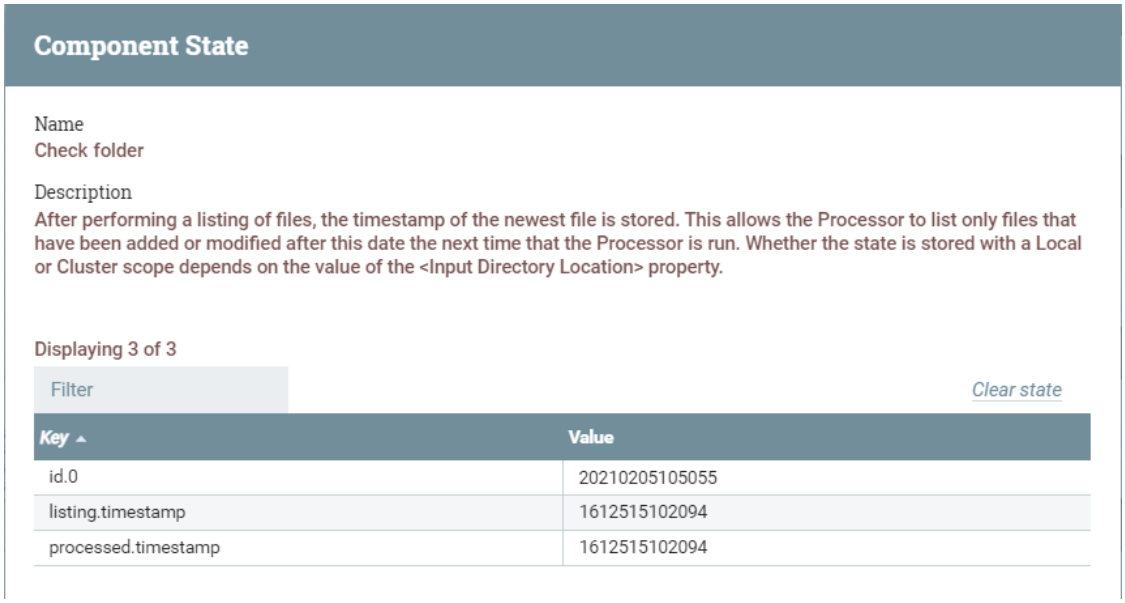

Figure 12. The component state view of the "ListFile" processor. Screenshot [13].

The outcome of this project, the multiuser supported TTI trace processor tool, is meant for the use of the 5G testers inside the corporation. There are no plans to give the tool to the customers or let the operators use it at this stage. On the development side, the tool may be developed even further by writing the data alternatively into an Apache Cassandra file (see Listing 8) instead of the current parquet file for comparing the performance between the two data formats. Otherwise, further development will focus on tool optimization and strengthening the functionality of the tool in all situations.

```
import com.datastorage.Cassandra
val cassandra = Cassandra.withKeyspace("data")

cassandra.save(myDfWithTime, "ttitable",
    Option(Seq("message_type")), Option(Seq("physCellId","time_stamp")))
```

Listing 8. A Scala code draft for writing the processed data into an alternative Apache Cassandra table format. For later development.

As for the data confidentiality of this project, the application was built inside the corporation's intranet behind multiple firewalls, preventing and protecting from attacks from outer networks. Whether the inner and outer security measures are set properly in the corporation, the chances for the TTI trace data hosted on the cloud services leaking to the unauthorized personnel is minimal.

6 Conclusion

The goal of this project was to develop multiuser support into a prebuilt single-user application processing 5G TTI trace data running in the corporate cloud in the Apache Spark ecosystem.

The project was a success. A large investigation was done about the Apache NiFi and Apache Zeppelin tools, resulting in finding a solution that enabled developing a multiuser support capability for the prebuilt application. The solution may not be the most optimal one due to possible unnecessary invokes in the Spark interpreter when utilizing the multiuser support tool. Ultimately, the target was reached, and the end users are satisfied with the outcome. Value was added into the TTI trace data research and development capabilities of the commissioning company, Nokia Corporation.

I thank my colleagues at the Nokia Field Verification organization for co-operation and for reviewing my thesis. Special thanks go to Letizia Iannucci, Working Student, for assisting with the Scala code development. The outcome of this project would not be the same without the support from the helpful colleagues working at Nokia Corporation.

The thesis offers support to developers, researchers and hobbyists who are studying the capabilities, limits and applications developed by using the Apache Spark software and the Apache Foundation projects in Apache Spark’s ecosystem, such as Apache NiFi and Apache Zeppelin.

References

1 Global 5G Coverage to Grow 253% by 2025 and Reach 53% of Population. Online. Bankr.nl. <https://bankr.nl/global-5g-coverage-to-grow-253-by-2025-and-reach-53-of- population/> Accessed March 14, 2021.

2 Kurose, James F.; Ross, Keith W. 2013. Computer Networking: A Top-Down Approach. 6th ed. Brooklyn, NY: Addison-Wesley.

3 Liljeström, Henrik. 2021. Product Architect, Nokia Corporation, Online interview, March 25, 2021.

4 Beginners: 5G Numerology. Online. Uploaded to YouTube by user "3G4G". <https://www.youtube.com/watch?v=7p23PeddK-4> Accessed March 25, 2021.

5 Pointer, Ian. What Is Apache Spark? The Big Data Platform That Crushed Hadoop. Online. InfoWorld. <https://www.infoworld.com/article/3236869/what-is-apache-spark-the-big-data-platform-that-crushed-hadoop.html> Accessed March 26, 2021.

6 Cluster Mode Overview. Online. Apache Spark. <https://spark.apache.org/docs/latest/cluster-overview.html> Accessed March 16, 2021.

7 Apache NiFi Overview. Online. Apache NiFi. <https://nifi.apache.org/docs/nifi-docs/html/overview.html> Accessed March 26, 2021.

8 Apache Zeppelin. Online. Apache Zeppelin. <https://zeppelin.apache.org/> Accessed March 26, 2021.

9 Apache Parquet Documentation. Online. Apache Parquet. <http://parquet.apache.org/documentation/latest/> Accessed March 26, 2021.

10 What Is Amazon S3? Online. Amazon Web Services. <https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html> Accessed March 26, 2021.

11 Apache NiFi In Depth. Online. Apache NiFi. <https://nifi.apache.org/docs/nifi-docs/html/nifi-in-depth.html> Accessed March 27, 2021.

12 NiFi Version 1.13.1 Release Notes. Online. Apache Nifi. <https://issues.apache.org/jira/secure/ReleaseNote.jspa?projectId=12316020&version=12349733> Accessed April 6, 2021.

13 Apache NiFi [software]. Version 1.12.1. Apache NiFi team. March 27, 2021.

14 Apache Zeppelin [software]. Version 0.8.2. Apache Zeppelin team. March 27, 2021.

tti_trace_processor.scala

```
%spark
{
import org.apache.spark.sql.types
import org.apache.spark.sql.SaveMode
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.functions.split
import org.apache.spark.sql.functions

val S3BucketName = "SGV5ISBObyBjb3Jwb3JhdGUgc3B5aW5nLCBwbGVhc2UuIDop"
val folderName = "s3a://" + S3BucketName + "/tti_temp/"
val inputDataFileName = "*.gz"
val inputDataFile = folderName + inputDataFileName

val counterToMap = udf((counterString: String) => {
    if (counterString.length > 0) {
        val counters = counterString.split(",").toSeq
        val counterLength = (counters.length % 2) match {
            case 0 => counters.length
            case _ => counters.length - 1
        }
         val r = Range(0, counterLength/2)
         r.map(i => (counters(i).trim() ->
counters((counters.length/2)+i).trim())).toMap
    } else Map.empty[String, String]
}
)

val myDF = spark
  .read
  .option("header", false)
  .option("sep", ":")
  .option("inferSchema", "true")
  .csv(inputDataFile)
  .withColumnRenamed("_c0","message_type")
  .withColumn("counters",counterToMap(col("_c1")))
  .withColumn("input_file", input_file_name())
  .drop("_c1")
   .withColumn("slot", expr("CAST (counters.slot as Int)"))
   .withColumn("sfn", expr("CAST (counters.sfn as Int)"))
   .withColumn("physCellId", expr("CAST (counters.physCellId as Int)"))
   .withColumn("file_id", split(col("input_file"), "\\.").getItem(2) cast
"Int")
   .filter('message_type === "dlFdSchedData" ||  'message_type ===
"ulFdSchedData" || 'message_type === "ulPuschReceiveRespPsData" ||
'message_type === "ulUtcTimestampData" || 'message_type === "dlBeamData" ||
'message_type === "ulBeamData")
    .filter('file_id >0)

//calculate ref time in ms from utc time stamps - starting from time 00:00.00
val myDfWithRefTime = myDF.filter('message_type === "ulUtcTimestampData")
    .withColumn("ref_time_ms", expr("CAST (counters.hours as
Double)")*3600*1000 +  expr("CAST (counters.minutes as Double)")*60*1000 +
```

```
expr("CAST (counters.seconds as Double)")*1000 + expr("CAST
(counters.microSeconds as Double)")/1000)
    .withColumn("ref_day", expr("CAST (counters.days as Int)"))
    .withColumn("microSeconds", expr("CAST (counters.microSeconds as Int)"))
    .drop('counters)
    .drop('input_file)
    .drop('message_type)
    .drop('physCellId)

//find smallest ref_time per fila and add sfn and slot with join
val myDfWithRefTimeSfn= myDfWithRefTime
    .groupBy('file_id).agg(min('ref_time_ms).as("ref_time_ms1"),
max('ref_time_ms).as("ref_time_ms2"))
    .withColumn("ref_time_ms", when('ref_time_ms1+3000 < 'ref_time_ms2,
'ref_time_ms2).otherwise('ref_time_ms1))
    .join(myDfWithRefTime, Seq("file_id", "ref_time_ms"))
    .withColumnRenamed("sfn", "ref_sfn")
    .withColumnRenamed("slot", "ref_slot")

val startTime =
myDfWithRefTime.orderBy('ref_time_ms).select('ref_time_ms).first().getDouble(0
)

val myDfWithTime = myDF.join(myDfWithRefTimeSfn, Seq("file_id")
    .withColumn("time_add_ms", when('ref_sfn >= 350, when('sfn < ('ref_sfn -
350), ('sfn - 'ref_sfn)*10 + ('slot - 'ref_slot)/2 + 10240).otherwise(('sfn -
'ref_sfn)*10 + ('slot - 'ref_slot)/2))
        .otherwise(when('sfn > ('ref_sfn + 674), ('sfn - 'ref_sfn)*10 + ('slot
- 'ref_slot)/2 - 10240).otherwise(('sfn - 'ref_sfn)*10 + ('slot -
'ref_slot)/2)))
    .withColumn("time_ms", 'ref_time_ms + 'time_add_ms)
    .withColumn("hours", floor('time_ms / 3600 / 1000).cast(StringType))
    .withColumn("minutes", floor(('time_ms - 'hours * 3600 * 1000) / 60 /
1000).cast(StringType))
    .withColumn("seconds", floor(('time_ms - 'hours * 3600 * 1000 - 'minutes
*60 *1000) / 1000).cast(StringType))
    .withColumn("milliseconds", floor(('time_ms - 'hours * 3600 * 1000 -
'minutes *60 *1000 - 'seconds *1000)).cast(StringType))
    .withColumn("milliseconds100", floor(('time_ms - 'hours * 3600 * 1000 -
'minutes *60 *1000 - 'seconds *1000)/100).cast(StringType))
    .withColumn("milliseconds10", floor(('time_ms - 'hours * 3600 * 1000 -
'minutes *60 *1000 - 'seconds *1000 -
'milliseconds100*100)/10).cast(StringType))
    .withColumn("milliseconds1", floor(('time_ms - 'hours * 3600 * 1000 -
'minutes *60 *1000 - 'seconds *1000 -'milliseconds100*100 -
'milliseconds10*10)).cast(StringType))
    .withColumn("microseconds100", floor((('time_ms - 'hours * 3600 * 1000 -
'minutes *60 *1000 - 'seconds *1000 - 'milliseconds)*10)).cast(StringType))
//with to numbers
    .withColumn("time_stamp", to_timestamp(concat(date_format(current_date,
"YYYY-MM"), lit("-") // tinmeStamp CURRENT YEAR, MONTH + MEASUREMENT DAY +
TIME
    , 'ref_day.cast(StringType), lit(" ")
    , 'hours, lit(":")
    , 'minutes, lit(":")
    , 'seconds, lit(".")
    , 'milliseconds100
    , 'milliseconds10
    , 'milliseconds1
    , 'microseconds100, lit("1")  //"1" added as zeros are dropped from the
end for some reason
    )))
    .withColumn("ref_seconds", (floor(('time_ms -
lit(startTime))/1000)).cast(IntegerType) +1) //starting from the first utc
timestamp - 1second
    //.drop('ref_sfn)
    .drop('ref_slot)
```

```
    .drop('ref_day)
    .drop('time_add_ms)
    .drop('hours)
    .drop('minutes)
    .drop('seconds)
    .drop('milliseconds)
    .drop('milliseconds100)
    .drop('milliseconds10)
    .drop('milliseconds1)
    .drop('microseconds100)
    .drop('ref_time_ms)
    .orderBy('time_ms)
    .filter('message_type === "dlFdSchedData" ||  'message_type ===
"ulFdSchedData" || 'message_type === "ulPuschReceiveRespPsData" ||
'message_type === "dlBeamData" || 'message_type === "ulBeamData") // leave
myDfWithTime
    .repartition(1)
    .write.mode("append").parquet(s"s3a://$S3BucketName/parquet.parquet")
//overwrite append

val delfile = FileSystemUtil(spark).delete(folderName)

}
```

tti_trace_processor_mu.scala

```
%spark
{
import org.apache.spark.sql.types._
import org.apache.spark.sql.SaveMode
import com.nokia.ava.npo.spark.FileSystemUtil
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.functions.{col, expr, input_file_name, split, udf,
when}
import org.apache.commons.io.FilenameUtils
import scala.util.control.Breaks._

println("Process started!")

// counterToMap
val counterToMap = udf((counterString: String) => {
    if (counterString.length > 0) {
        val counters = counterString.split(",").toSeq
        val counterLength = (counters.length % 2) match {
            case 0 => counters.length
            case _ => counters.length - 1
        }
         val r = Range(0, counterLength/2)
         r.map(i => (counters(i).trim() ->
counters((counters.length/2)+i).trim())).toMap
    } else Map.empty[String, String]
}
)

// get the file_id
val getFileId = udf((filePath: String) => {
    filePath.split("\\.").reverse(3).toInt
})

// list of folders (input files)
val folderNames = FileSystemUtil(spark)
    .list("s3a://SGV5ISBObyBjb3Jwb3JhdGUgc3B5aW5nLCBwbGVhc2UuIDop/tti_temp/")
    .toList

println("Starting loop!")

for (folderName <- folderNames) {

    println("Starting processing for: " + folderName)

    var waitCount = 0

    breakable {

        while (waitCount < 12) {

            println("Inside while loop...")

            val folderNameContentsStr =
FileSystemUtil(spark).list(folderName).toList.toString

            if (folderNameContentsStr.contains("/ok")) {

                println("Contains string '/ok' -> Now inside if-condition...")

                val inputDataFileName = "/*.gz"  // The input csv file (also
gzip compression supported) for cell frequency info
                val inputDataFile = folderName + inputDataFileName
```

```
println(s"Starting: $inputDataFile")

println("1")

val myDF = spark
  .read
  .option("header", false)
  .option("sep", ":")
  .option("inferSchema", "true")
  .csv(inputDataFile)
  .withColumnRenamed("_c0","message_type")
  .withColumn("counters", counterToMap(col("_c1")))
  .withColumn("input_file", input_file_name())
  .drop("_c1")
   .withColumn("slot", expr("CAST (counters.slot as Int)"))
   .withColumn("sfn", expr("CAST (counters.sfn as Int)"))
   .withColumn("physCellId", expr("CAST (counters.physCellId
as Int)"))
   .withColumn("file_id", getFileId(col("input_file")))
   .filter('message_type === "dlFdSchedData" ||  'message_type
=== "ulFdSchedData" || 'message_type === "ulPuschReceiveRespPsData" ||
'message_type === "ulUtcTimestampData" || 'message_type === "dlBeamData" ||
'message_type === "ulBeamData")
    .filter('file_id >0) //skip first file, as it is having
not valid data


//calculate ref time in ms from utc time stamps - starting
from time 00:00.00
val myDfWithRefTime = myDF.filter('message_type ===
"ulUtcTimestampData")
    .withColumn("ref_time_ms", expr("CAST (counters.hours as
Double)")*3600*1000 +  expr("CAST (counters.minutes as Double)")*60*1000 +
```

```
expr("CAST (counters.seconds as Double)")*1000 + expr("CAST
(counters.microSeconds as Double)")/1000)
                    .withColumn("ref_day", expr("CAST (counters.days as
Int)"))
                    .withColumn("microSeconds", expr("CAST
(counters.microSeconds as Int)"))
                    .drop('counters)
                    .drop('input_file)
                    .drop('message_type)
                    .drop('physCellId)

                println("2")

                //find smallest ref_time per fila and add sfn and slot with
join
                val myDfWithRefTimeSfn= myDfWithRefTime

.groupBy('file_id).agg(min('ref_time_ms).as("ref_time_ms1"),
max('ref_time_ms).as("ref_time_ms2"))
                    .withColumn("ref_time_ms", when('ref_time_ms1+3000 <
'ref_time_ms2, 'ref_time_ms2).otherwise('ref_time_ms1))
                    .join(myDfWithRefTime, Seq("file_id", "ref_time_ms"))
                    .withColumnRenamed("sfn", "ref_sfn")
                    .withColumnRenamed("slot", "ref_slot")

                println("3")
                println(myDfWithRefTime.show())

                val startTime =
myDfWithRefTime.orderBy('ref_time_ms).select('ref_time_ms).first().getDouble(0
)

                val archiveFileName = FilenameUtils.getName(folderName)

                println("4")

                val myDfWithTime = myDF.join(myDfWithRefTimeSfn,
Seq("file_id"))  //add TimeStamp per file id from ulUtcTimestampData smallest
time -- MIN SLOT NBR REFERENCE FOR UTC TIME STAMP (always same???)
                  /* .withColumn("time_add_ms", when('sfn - 'ref_sfn < 500 &&
'sfn - 'ref_sfn >= -500, ('sfn - 'ref_sfn)*10 + ('slot - 'ref_slot)/2)
                      .when('sfn - 'ref_sfn < -500, ('sfn - 'ref_sfn)*10 +
('slot - 'ref_slot)/2 + 10240)
                      .otherwise(('sfn - 'ref_sfn)*10 + ('slot -
'ref_slot)/2 - 10240))
                   */
                    .withColumn("time_add_ms", when('ref_sfn >= 350, when('sfn
< ('ref_sfn - 350), ('sfn - 'ref_sfn)*10 + ('slot - 'ref_slot)/2 +
10240).otherwise(('sfn - 'ref_sfn)*10 + ('slot - 'ref_slot)/2))
                      .otherwise(when('sfn > ('ref_sfn + 674), ('sfn -
'ref_sfn)*10 + ('slot - 'ref_slot)/2 - 10240).otherwise(('sfn - 'ref_sfn)*10 +
('slot - 'ref_slot)/2)))
                    .withColumn("time_ms", 'ref_time_ms + 'time_add_ms)
                    .withColumn("hours", floor('time_ms / 3600 /
1000).cast(StringType))
                    .withColumn("minutes", floor(('time_ms - 'hours * 3600 *
1000) / 60 / 1000).cast(StringType))
                    .withColumn("seconds", floor(('time_ms - 'hours * 3600 *
1000 - 'minutes *60 *1000) / 1000).cast(StringType))
                    .withColumn("milliseconds", floor(('time_ms - 'hours *
3600 * 1000 - 'minutes *60 *1000 - 'seconds *1000)).cast(StringType))
                    .withColumn("milliseconds100", floor(('time_ms - 'hours *
3600 * 1000 - 'minutes *60 *1000 - 'seconds *1000)/100).cast(StringType))
                    .withColumn("milliseconds10", floor(('time_ms - 'hours *
3600 * 1000 - 'minutes *60 *1000 - 'seconds *1000 -
'milliseconds100*100)/10).cast(StringType))
```

```
                    .withColumn("milliseconds1", floor(('time_ms - 'hours *
3600 * 1000 - 'minutes *60 *1000 - 'seconds *1000 -'milliseconds100*100 -
'milliseconds10*10)).cast(StringType))
                    .withColumn("microseconds100", floor((('time_ms - 'hours *
3600 * 1000 - 'minutes *60 *1000 - 'seconds *1000 -
'milliseconds)*10)).cast(StringType)) //with to numbers
                    .withColumn("time_stamp",
to_timestamp(concat(date_format(current_date, "YYYY-MM"), lit("-") //
tinmeStamp CURRENT YEAR, MONTH + MEASUREMENT DAY + TIME
                    , 'ref_day.cast(StringType), lit(" ")
                    , 'hours, lit(":")
                    , 'minutes, lit(":")
                    , 'seconds, lit(".")
                    , 'milliseconds100
                    , 'milliseconds10
                    , 'milliseconds1
                    , 'microseconds100, lit("1")  //"1" added as zeros are
dropped from the end for some reason
                    )))
                    .withColumn("ref_seconds", (floor(('time_ms -
lit(startTime))/1000)).cast(IntegerType) +1) //starting from the first utc
timestamp - 1second
                    //.drop('ref_sfn)
                    .drop('ref_slot)
                    .drop('ref_day)
                    .drop('time_add_ms)
                    .drop('hours)
                    .drop('minutes)
                    .drop('seconds)
                    .drop('milliseconds)
                    .drop('milliseconds100)
                    .drop('milliseconds10)
                    .drop('milliseconds1)
                    .drop('microseconds100)
                    .drop('ref_time_ms)
                   //.filter('message_type === "dlFdSchedData")
                    //.filter('ref_seconds === 300)
                    //.drop('counters)
                    .orderBy('time_ms)
                    .filter('message_type === "dlFdSchedData" ||
'message_type === "ulFdSchedData" || 'message_type ===
```

```
"ulPuschReceiveRespPsData" || 'message_type === "dlBeamData" || 'message_type
=== "ulBeamData") // leave timestamp message out
                    .withColumn(archiveFileName, lit(0)) // save the input
filename as a column

                  //  z.show(myDfWithTime)

                println("5")

                val outputTarget = s"s3a://
SGV5ISBObyBjb3Jwb3JhdGUgc3B5aW5nLCBwbGVhc2UuIDop/mu_parquet.parquet"

                // output as parquet
                myDfWithTime
                    .repartition(1)
                    .write.mode("append").parquet(outputTarget)

                println("6")

                /*
                // output as cassandra
                import com.nokia.ava.npo.datastorage.Cassandra
                val cassandra = Cassandra.withKeyspace("data") // A Cassandra
keyspace must created beforehand

                cassandra.save(myDfWithTime, "ttitable",
                    Option(Seq("message_type")),
Option(Seq("physCellId","time_stamp")))   //  partition, clustering keys
                */

                //Delete processed input folder
                val delfile = FileSystemUtil(spark).delete(folderName)

                println(s"OK. Parquet appended to: $outputTarget \n")
                println("Breaking while loop...")

                break

            } else {

                println("Upload not ready yet for " + folderName + ". Waiting
10 secs...")
                Thread.sleep(10000)
                waitCount = waitCount + 1

            } // end of if-else condition
        } // end of while loop bracket
    } // end of loop.breakable bracket
} // end of for loop bracket

println("End of loop! (If empty, then there were no files to process.)")

// end of spark interpreter bracket
}
```