Data Processing

Architectures and Techniques Applied to CubeSat Missions

Anatoly Ilin

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

Data Processing

Architectures and Techniques Applied to CubeSat Missions

by

Anatoly Ilin

in partial fulfillment of the requirements for the degree of

Master of Science
in Aerospace Engineering

at the Delft University of Technology,
to be defended publicly on Tuesday July 17, 2018 at 13:00 AM.

Supervisor:	Dr. ir. S. Sperettta	
Thesis committee:	Dr. ir. A. Cervone,	TU Delft
	Dr. ir. B. C. Root,	TU Delft

TU Delft Delft University of Technology

Contents

Nomenclature

2PC	Two-Phase Commit
A	Availability
AC	Availability-Consistency
ACK	Acknowledgment
ADC	Analog to Digital Conversion
AP	Availability-Partition Tolerance
API	Application Programming Interface
AVG	Average
AX.25	AX.25 protocol
BER	Bit Error Rate
C	Consistency
CAP	Consistency, Availability, Partition Tolerance
COTS	Commercial off-the-shelf
CP	Consistency-Partition Tolerance
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CRUD	Create, Read, Update, Delete operations
DAO	Data access object
DAS	Directed Attached Storage
DB	Database
DDOS	Distributed Denial of Service
DF	Data Frame and Telemetry Frame
DGS	Delfi Ground Station
DLSAP	Data-Link Service Access Point
DOM	Document Object Model
DOS	Denial of Service
ECA	External Client Application
ELK	ElasticSearch , LogStach, Kibana stack
ES	ElasticSearch
FlexTLM	Flexible telemetry frame defintion (standard)
GC	Galera Cluster
GDS	Graph data store
GENSO	Global Educational Network for Satellite Operations
GS	Ground Station
HA	High-Availability
HDFS	Hadoop Distributed File System
HK	House-keeping Transceiver mode
I	AX.25 Information Frame
ISR	In-Sync Replica
ITRX	Secondary, ISIS Transceiver
JVM	Java Virtual Machine
KS	Kaitai Struct
KV	Key-value
LoC	Lines of Code
LSB	Least significant bit
MM	Multi-Master
MS	Master-Slave
NAS	Network Attached Storage
NDC	Near-data computing
NIC	Networking Interface Controller

OBC	On-board Computer
OOP	Object-Oriented Programming
OOS	Out of Scope
OS	Operating system
OSI	Open Systems Interconnection
P	Partition Tolerance
PIF	Partially incorrect frames
PL	Payload Transceiver mode
PoC	Proof of Concept
PTRX	Primary Transceiver
R	Read Operations
RA	Radio Amateur
RAM	Radom Access Memory
RCE	Remote code execution
RDBMS	Relational DataBase Management Systems
RDD	resilient distributed dataset
RF	Replication Factor
RPI	Raspberry Pi
RW	Read-Write Operations
S	AX.25 Supervisory Frame
S/C	Spacecraft
SAN	Storage Area Network
SDR	Software Defined Radio
SE	Software Engineering
SOA	Service-Oriented Architecture
SoC	Separation of Concerns
SQL	Structured Query Language
STD	Standard Deviation
STX	S-band Transceiver
TBD	To be determined
TLE	Two Line Elements
TPS	Telemetry Processing System
TU	Technische Universiteit
TUD	Delft Univerisity of Technology
U	AX.25 Unnumbered Frame
UHF	Ultra High Frequency
UI	AX.25 Unnumbered Information Frame
UI	User Interface
VHF	Very High Frequency
VPN	Virtual Private Network
W	Write Operations
WOD	Whole-orbit data
XID	AX.25 Exchange Information Frame
XML	eXtensible Markup Language
XTCE	Telemetry and Command Exchange Protocol
YAML	Ain’t Markup Language
YARN	Yet Another Resource Negotiator

1

Introduction

This master thesis addresses the need for a telemetry processing for Delfi Space spacecraft operations, a nano-satellite program of the Delft University of Technology. It lays the groundwork for the telemetry processing system (TPS) implementation and explores the design guidelines for a highly scalable but robust and adaptive architecture.

Brief history

The history of the nano- and pico-satellites, artificial satellites with a mass below 10kg, and in particular the CubeSats, cube-shaped pico-satellites, started with a conceptual introduction in 1999 by California Polytechnic State University [1] as an attempt to reduce cost and development time. Although not designed as such, the proposal became a standard over time [2]. From 2009 up to early 2013, research in the nano-satellite industry was primarily driven by the civil market, which can be mostly attributed to research institutions and universities [3]. In 2013, the nano-satellite technology matured to the level where the commercial stakeholders stepped in, rapidly increasing the market share from 8 to 56 %. At this point, the majority of the launched spacecraft were no longer technology demonstrators, but remote sensing and earth observation missions [3]. This growth affected the availability of commercial off-the-shelve products and services (e.g. launch and operations services). However, due to the historic risk-mitigating nature of the commercial applications [4], the Commercial off-the-shelf (COTS) solutions are less affordable for the non-commercial and educational markets. The cost of the ground segment has lead to collaborative solutions. The Global Educational Network for Satellite Operations (GENSO) [5] and SwissCube EPFL [6] network, expand the satellite coverage by connecting the participating ground stations to their networks. Delfi Space [7] utilizes a drastically different approach by involving the radio amateur (RA) community for telemetry reception.

Satellite telemetry processing software is provided to all participating RA, allowing received telemetry frames to be transferred to the central Delfi server for processing and visualization in form of a website. With participating radio amateurs all around the world, a larger downlink budget can be achieved allowing for more scientific data to be transferred to earth.

Problem statement

The need for a robust, adaptive and low-cost solution for spacecraft operations, and thus telemetry processing originates from research institutions and universities in the civil part of the market [8]. Academia seeks collaboration with radio amateurs to increase the coverage while maintaining low operational costs. This results in a unique ground segment infrastructure, often not supported by the commercially off-the-shelve solutions.

While the need for telemetry processing was solved in the legacy missions, common flaws are observed in both C3 and n3Xt software applications leading to their lack of re-usability. First, both systems are tailored to specific missions, failing to (timely) accommodate for new missions and changing requirements. Secondly, the processing errors present in both systems [9] and telemetry frames loss in Delfi-n3Xt TPS directly affected data quality and reliability.

Focusing on generic CubeSat mission requirements instead of a specific Delfi mission would allow for a system with wider applications, thus facilitating reusability for future missions. While possible in theory, key requirements can be missed leading to a sub-optimal solution. Thus, implementing all foreseeable features, i.e. future-proofing the system, is a futile attempt.

The aforementioned challenges open the door to a different approach to the problem, by which the PTS can be designed to provide intuitive interfaces for functionality extension and modifications instead of future-proofing and locking system development.

The recent popularity of CubeSat and nano-satellite [10] swarm and constellation missions driven by decreasing hardware and launch costs have started a shift in the perspective of the ground segment design. The new infrastructure ought to support multiple spacecraft, technologically superior and capable of generating more data than ever before, leading to the need to collect, transfer and process potentially large amounts of data in quasi-real-time. The trends in Big-Data have accelerated the shift of the processing paradigm from 'cleaning and decoding' towards data mining with the goal of identifying hidden trends (such as, possible failures) at satellite level. [11]

The PTS should therefore not only support multiple satellite missions but consequently be able to process and store large data sets, such as imagery data. With RA participation follows the general rule of "more participants, more data", but suffers from unpredictable participation and therefore large variations of data reception are to be expected. Due to the distributed nature of the ground segment, data duplication due to RAs reception overlap is inevitable, driving the ingestion rate even further.

The Delfi team require a system that is not only capable of providing a platform to facilitate new missions but can also accommodate for changing data ingestion and processing rates while maintaining the flexibility to incorporate new functionalities in limited periods of time (typically linked to the duration of a thesis or internship).

Research Question

How should a telemetry processing system that acts as a bridge between satellite ground stations and the end users operate to ensure robustness and reliability, while meeting the necessary system requirements and without limiting the (future) missions and satellite design?

Due to ambiguity and the complexity of this problem, it is divided into the following subtasks:

1. Aggregate and define system requirements based on the legacy Delfi missions
2. Study and define the suitable server architecture(s)
3. Asses the impact of the future missions on the telemetry server requirements and chosen architecture (scalability and adaptability)
4. How can the robustness and reliability of the system be ensured?
5. How should the telemetry server framework operate to meet the requirements?
6. How can architecture benefit the data science?
7. How to ensure that the implementation is feasible within a thesis/internship period?

Expected outcome

The expected outcome of this thesis project is a framework architecture and design for a telemetry processing system. The proposed data processing architecture should be flexible and pose no requirements on the satellite and Radio Amateur community. The implemented proof of concept of PTS will be tested on the data from the Delfi-n3Xt missions.

The process that led to the solution for the complex problem is presented in chapter 2 and covers the methodology and chronological steps of this MSc thesis. The remainder of the document is divided into three parts: Foundations, Application and Results. The first part focuses on the requirement definition of the system and start with the high-level analysis of the Delfi mission segment, and the legacy system in chapter 3. The analysis showed two major areas of improvement for overall robustness and reusability of the system: data storage and data processing, that are further studied in chapters 4 and 5. Chapter 6 aggregates the applied concepts for the system architecture design and does not follow the chronological order of this document.

The second part of the document, focuses on the application of the study performed in part one and presents the possible architectures in chapter Architectures. The discussion leads towards two feasible architectures, the client leveraged and server based. The former is presented in chapter 8 and operates on the hypothesis that by harvesting the available computational resources of the RA the computational requirements of the central server can be reduced. The proof of concept disproved the hypothesis and lead to requirement refinement that led to the Server-based processing architecture. Due to scalability and reliability requirements, distributed server-based processing will be further studied and discussed in chapter 9.

Part three contains the design of the proposed architecture in chapter 10 and experiments in chapter 11 aiming at the investigation of robustness, reliability and reusability of the system.

The discoveries, requirements acceptance and results of the experiments are finally combined in the section 12 providing the conclusion for the project.

2

Project Methodology

This chapter focuses on the project methodology and aims to provide the background on the long-term decision process which leads to the system presented in chapter 9. The project consisted of numerous steps (which were required to build up the knowledge base), understanding of the available system, and comprehend the stakeholders' needs. The process will be discussed in section 2.1, followed by application in section 2.2 linking the process to the material presented in the document. Section 2.3 provides an overview of the tools utilized during the project and discuss the usability and contributions to the project outcome.

2.1. Theory

The system discussed in chapter 9 has a strong emphasis on modularity required for the redundant deployment. The skills necessary to achieve the level of design and understanding of the available system, as well the scope of data processing, were not available at the start of the project.

To facilitate the learning process and to prevent the premature sub-optimal decisions affecting the complete design, an interactive process was required. With a strong emphasis on a reusable system, a strong emphasis was laid on the demonstration of separate elements. To facilitate the necessity of iterations and requirement for intermediary prototyping, spiral development was selected.

The iterative process is applied both on the high level of the project, leading to the high-level proof of concepts (PoC), such as PouchCouch and Lambda system, as well as on the low level, for example, the iterative experimentation.

Proposed by Boehm [12], spiral model structures an interactive process into four distinct categories: Determine Objectives, Identify & Resolve the risks, Development & Testing and Plan for the next iteration. The portion of the project is run sequentially, allowing further refinement of the system and (re-)discovery of the requirements at any stage of the project. Figure 2.1 provides a simplified overview of the iterations, while the individual components will be discussed in the following chapters.

2.1.1. Determine Objectives

The first activity of each iteration focuses on the discovery of the objective: requirements, alternatives and potential constraints. The objectives aim to broaden the scope of the available technologies and when applied iteratively increases the understanding of the problem scope.

2.1.2. Identify & Resolve the Risks

The second activity focuses on the evaluation of the alternatives and observes the design of the high-level design. Although there is an overlap in iteration with the first phase; the second phase shifts into the technical and hands-on spectrum of the project, and ultimately leading to the prototype phase.

2.1.3. Development & Testing

During this phase, the Proof of Concept is developed and tested.

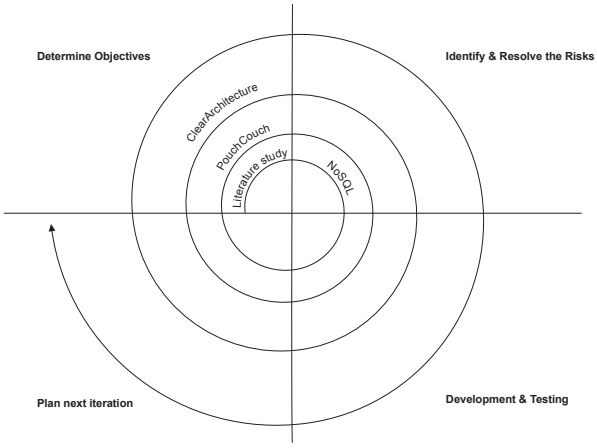

Figure 2.1: Thesis iterative process

2.1.4. Plan the Next Phase

The final phase concludes the previous iteration and provides the guidelines for the next. In practice, the planning overlaps the initial phase in the first two iterations, as a literature study and research was required.

2.2. The Application

With the theory explained in section 2.1, this sections aims to apply it on the project and aims as a guide to this document. The structure of this document follows the iterative process, placing the chapters in the sequence of iterations. Furthermore, each chapter is contains a conclusion summarising the findings, further redefining the system requirements.

2.2.1. First Iteration

The starting point of the project is the literature study aimed to understand the global scope of satellite telemetry processing and identify the available solutions (COTS). In conclusion, a in-house system was developed to comply with the requirements and associated risks.

With direction set towards an in-house developed system, a deeper look at the existing software to better understand the software operations, requirements and the needs of the stakeholders was required. The legacy systems is discussed in chapter 3 and complied with a preliminary list of requirements, this is expanded on each iteration of the project, in appendix A.

One of the significant weakness associated with the in-house developed software was the storage system. To ensure robustness; the storage system was built for single frame definition (as found in case of Delfi-software) or required following strict vendor specifications in case of COTS [13]. To circumvent this limitation, alternative systems were considered, leading to research in data storage systems presented in Chapter 4 in the second phase of the iteration. In retrospect, it can be argued that an error was made in this phase, this shifted the focus towards the data storage solutions, instead of recognizing a more widespread problem of the system inflexibility.

In the third phase, albeit not explicitly included in the report, NoSQL systems were assessed, leading to the further research into deployment mechanisms. This identified the weakness of single node database deployment with regards to hardware failures and increased chance of data loss. Furthermore, a brief study of a typical system availability was performed, leading to requirement updates. The study identified that a single database node operation has a higher potential for data loss than previously expected.

The fourth phase concluded the first iteration. It was determined that the NoSQL was a valid approach for data storage and provided extremely flexible and robust alternative for SQL systems. Secondly, the need for a distributed, multi-node database system was found necessary for the redundancy of operations and prevention of data loss. Historic Delfi-n3Xt and Delfi-C3 data study lead to two conclusions. First, the database system was not properly sized in terms of hardware. Second, the

load on the system was found to be proportional to the number of users ingesting the data. This in combination with the need for multi-node deployment, with nodes preferably spread geographic, lead to the concept of scaling over the client application, rather than scaling of database nodes on the Delft premises.

2.2.2. Second Iteration

The second iteration represents the work done in chapters 4 and 8. The first phase aimed to research the feasibility of client-side deployment, and impact on the existing architecture. Second, alternatives for deployment were assessed. With a wide range of available hardware and OS on the market, development of an application capable of running on the majority of available machines (RA's end) was not trivial. At the time of writing, two systems were available, Java virtual machine (JVM) and browsers. Java applications, such as DuDe client, is a well known and well-tested way of achieving this. Albeit and JVM approach was discarded due to additional requirements on the client, e.g. installation of JVM, memory use.
The research instead focused on the browser deployment vector. As discussed in section 8.2.1, browsers are available on a broad range of equipment and provide high capabilities at a low development cost. No comparable systems were previously attempted in browser deployment. At the time of writing,, no similar academic research was found on the database scaling over the client machines. The system promised a distributed storage and compute cluster over heterogeneous hardware, distributed over client applications.

The second phase, covering the capabilities, opportunities and possibilities is aggregated in chapter 8. A partial PoC was implemented, covering critical components: server-client data replication and processing. Brief research on Delfi-n3Xt dataset was conducted to determine data quality, errors and possible loss. The results are presented in the aggregation chapter 3 and lead to the conclusion that the majority of data quality degradation is due to human errors, primarily, in the processing. The investigation of the Delfi-n3X client and server software lead to the conclusion that these errors can be resolved with the use of unified processing. The research is left out of the report due to time constraints, on focuses on the most applicable Kaitai framework 5.

The third phase provided a broad and critical look at the proposed PouchCouch system. As presented in chapter 8 numerous weakness were identified. For sake of argument, phase three will be presented here. First, client applications could not be trusted; hence data provided by client applications needed to be verified. Second, the browser deployment is an insecure platform. To address the issue, data has to be re-computed on three nodes to be validated. Furthermore, the re-computation is magnitudes faster than the intra-node data replication. Hence, moving data to clients for computations is magnitudes slower than the computations themselves. Finally, analysis of the long-term support revealed high complexity regions with the architecture in the most critical parts; hence, the most vital system components are the easiest to make mistakes in.

The fourth phase consumed the work of the third phase - acting on the conclusion and findings, leading to the abrupt decision to discontinue client application use. The in-depth look at PouchCouch system triggered the need for refinement of the requirements and lay focus on the long-term use of the system, rather than an optimization of a single component: storage.

2.2.3. Third Iteration

The third iteration focused solely on the software design methodology. Albeit, not explicitly contained in this report, research was done on the available architecture design tools. Separation of Concerns was studied early on in the project but was applied incorrectly and insufficiently in the PouchCouch system. With no specific high-level tools applicable to the project scope, classicly low-level tools were studied.
Clean Architecture was discovered early in the project but was initially discarded due to popularity driven semantics and cult-like following, rather than scientific reasoning. Later, with a better understanding of available techniques, the core concept of Clean Architecture was extracted and applied on the high level to the Delfi project, presented in chapter 6.

The second phase of this iteration focused on the application of modified Clean Architecture on the Delfi mission, with emphasis on the ground software. The history of software development in Delfi projects revealed parallels with work presented in 1995 by Berczuk [14], and lead to the conclusion that the most suitable solution of the problem is a strong decomposition of the system components. This

is required since insufficient knowledge is passed between developers of the legacy systems, primarily limited to software and minimalistic documentation. The students required to build system components are likely inexperienced, and have a lack understanding of broader system architecture. This lead to the existing systems to be discarded in favour of newer systems, development of which cannot be completed fully within a timeframe of the masters' thesis. A vicious circle.

By following the Clean Architecture principles, thus by abstracting all system components, the scope of the (sub)-system to be designed can be greatly reduced. Early on in the iteration, there was a strong emphasis on freedom of choice on the programming language to be used for the component. This reduced the start-up time and increases the quality, as developer most likely has a preference and experience with one.
In the third phase, this assessment lead to the conclusion that choice of programming language does not guarantee anything, and that even higher level of abstraction should be applied. This lead to the conclusion that Delfi telemetry processing system should consist of discrete, well-defined components that can be exchanged and replaced with ease.

The concluding phase of the iteration focused on the available tooling and feasibility of the approach. The findings are appended to 3 chapter and lead to the conclusion that Delfi project is unpredictable from telemetry system point of view. This is best illustrated with "last minute" adjustments in Delfi-n3Xt telemetry processing system that lead to virtual data loss on the processing side. To address this problem on architecture level, the system should be adjustable to the extent that is not observed in the general software development world. However, one field of computer science was showed to share the same level of volatility: BigData.

2.2.4. Forth Iteration

The first phase of the project aimed to understand the scope and the needs that lead to the design of BigData systems. The popular definition of BigData, referring to a system containing a volume of data that cannot be stored on a single machine, showed a remarkable disconnect from the actual needs and applications. Similar to the Delfi needs, these systems aim to ingest data, without prior data (schema) definition that is not only unknown at the time of design but will most likely change frequently in time. Data processing is often extremely extendable, and easily adjustable in production allowing old processing code to be executed in parallel with the new algorithms.

In the second phase of the project, viable architectures were considered, lead to the material presented in 9 chapter. The process was driven by the SoC/Clean Architecture principles and lead to the architecture presented in chapter 10.

The third phase focused on the experimentation with the test setup to identify the required skills to maintain the cluster and aimed to explore the system capabilities. The experiments were performed on under-powered hardware to illustrate that a) no expensive equipment was required to run and execute such systems and b) that any hardware devices better than a credit card sized computer would be capable of supporting the designed architecture.

The fourth and the final phase of the project concluded the research in the form of this report. Logically, following the iterative design methodology, the next possible steps are presented in the 12 chapter.

2.3. Additional Aid

In the software industry, software engineering plays a vital role to keep development speed constant. This along with risk mitigation, makes product delivery timeframe predictable. The major risk in software project is change of requirements. There are two major methodologies: ensure perfectly defined requirements or embrace the change. This along with a clear, predefined set of steps as a guidance, makes the product delivery timeframe predictable. As described in section 2.1, globally, a spiral model was selected. While being very useful on the high level of the project, model did not provide any support for the day-to-day activities. Later the project, an additional tool was selected to assist with keeping track of progress, actively acting as a motivator. Due to incremental design philosophy, the selection was limited to the agile techniques. The incremental development is deeply embedded in techniques such as Kanban, Scrum, Prince2 and many other. Having worked with Kanban and Scrum in internship settling, Scrum was directly discarded due its overhead with regards to team participation that is not applicable to an one-man team. The second issue with scrum are the fixed sprints: an

abstract entity of the work performed in a fixed period of time. Due to the limiting experience with system architecture and utilised tools, reducing the certainty in the estimation of work load and run-time of the sprints. In professional and team setting, sprint provide a clear deadline for a given task, a clear separation of work with clean interfaces to the rest of the system, allocated by the experienced project manager. Additionally, daily stand-ups serve as a motivator and assist in knowledge sharing between team members. One-man-team, lacking both brilliant parts of the Scrum, makes it unusable, leading to extended duration of the sprints and eventually to abandoning the system. The visual aid, provided by the framework such as Jira, provide set of charts showing the completeness, but often based on the number of sprints and not the total work package. To accommodate for the former, and reduce the effect of insufficient planning, Kanban was considered, as in its core, the framework is designed to deal with uncertainty by not placing emphasis on planning. Within the framework, the work is divided by requirements in smaller, manageable chunks. Each chunk is prioritised, and divided in groups: backlog[”far”, ”soon”, ”next”], ”in progress” and ”done”. This ensures a clear overview of all tasks. Similar to Scrum provides a visual representation of work-done. The additional Cumulative Flow Chart can be utilised to visualise the progress in groups, with the idea that any sudden changes in visual composition is an indicator of issues with the project. Therefore Kanban is the most suited for projects with unknown and unstable nature.

In profession setting, Kanban contains a set of rules, often designed for team interaction in mind. This integration was unnecessary in the one-man-team, and where reduce to:

1. Keep track of ’chunks’
2. Two ’chunks’ to do at the same time

In practice, for the final prototypes Kanban ’pull’-nature was maintained. The connected ’chunks’ very prioritised in sequence, focusing the completion of interdependable components first. Consider the data ingestion system of the lambda architecture as an example. The implementation of the processing system: Spark, was not commenced until data ingestion system: Kafka was functional. This sequential approach was required to adhere to the global spiral model to evaluate the prototypes.

2.4. Project and System Requirements Structure

The iterative design lead to a growing set of requirements. The initial project requirements are derived in chapter 3 and are expanded overtime. To provide a better overview, the requirements are structured hierarchically starting with major system, followed by system or component, a modifier and a counter: GN-<MAJOR>-<MINOR/Modifier>. All requirements presented in this document start with GN, denoting Ground Network, encapsulating both the server and client systems.

The list of the major requirement grouping:
UI - User Interface
NET - Network and communications
ING - Data ingestion
PR - Data processing
DS - Data storage
DD - Data delivery
DQ - Data Query
PERF - Perforance related
SAFE - Safety related
QA - Quality (Attributes) related
BR - Business rules
AUX - Auxiliary requirements

The list of the minour grouping and optional modifiers:
GEN - Generic
RA - Radio Amateur usergroup related
OPS - Spacecraft Operator usergroup related
ADMIN - System administrator usergroup related
RAW - Related to unprocessed, raw data

INT - Related to integration with external system or component
ADMIN - System administrator usergroup related
STAT - Data Statistics
CL - Cluster, related the queries exectured on cluster, based on the complete dataset
SEC - Security related
REL - Reliability related
AVAIL - Availability related
MAIN - Maintainability related
DATA - Data related
DOC - Documentation related
LIC - Licensing related
LEG - Related to legal aspects of the project

I

Foundations

3

Project Context

The Delfi Space program is the development line of CubeSat and Pico-satellites at Delft University of Technology, with the goal to provide hands-on education and training for students, while serving a technology demonstrator for the Space Industry. [15]

Delfi missions utilized innovative space technology applications emerging from within TU Delft and external partners from the space sectors [15], often as a secundary system component or the spacecraft payload. To demonstrate the technical capabilities, various system parameters are gathered over the mission lifetime and analyzed on earth by the team of experts. To deliver the gathered data, or telemetry, to the stakeholders a reliable connection between satellite and ground segment has to be established. Historically, the Delfispace ground segment is set up as a collaborative effort from various stakeholders: TU Delft, TU Eindhoven, ISIS BV and others. However, to achieve the worldwide coverage, Delfi team relies on the assistance from a global radio amateur community to cover the blind spots [15].

This research thesis focuses on the permanent telemetry storage, processing and delivery of the processed data to the various mission stakeholders. This chapter serves as introduction to the past missions, architectures and implementations, and is used to kickstart the discussions contained in the following chapters by presenting the intial system requirements.

The discussion starts with establishing the high-level system components in section 3.1, followed by the discussion on the need for, and the results of Radio Amateur participation in section 3.2. The subsequent sections follow the data flow through the system, focusing on the aspects of telemetry generation onboard the spacecraft in section 3.3.1 as well the encoding of the data to the data frames in section 3.3.2. Section 3.4 provides an functional overview of data ingestion on the client side from telemetry perspective, defining the AX.25 protocol in section. 3.4.1. Section 3.5 focuses on the server side processing system, with section 3.6 defining the data budgets of the previous missions. Finally, the section 3.7 aggregates the gathered information and provides a preliminary analysis, a springboard for the discussion in the consecutive chapters.

3.1. The Space Mission Segments

The segment of Delfi-C3 and Delfi-n3Xt missions, applied to the telemetry processing system shown in the figure 3.1. Within the provided system breakdown, the space segment consists of the spacecraft generating the telemetry data and the communications interfaces. The ground segment is built up from three elements: managed Delfi Ground station, the semi-managed Backup Ground station at Eindhoven and ISIS B.V and an unmanaged network of Radio Amateurs. Delfi Ground station (DGS) element contains the TU Delft ground station hardware as well the systems responsible for Data storage and processing. The User Segment consists of the end-users of the mission: Payload stakeholders, Data scientists, Radio Amateurs and General Public.

For the scope of the project, the space segment is considered a black-box system, generating telemetry data and accepting telecommands. However, an exception will be made in this chapter, in order to understand and quantify the possible changes that may affect telemetry processing system.

The interface between Space and Ground segment are the radio transmissions, the downlink via very

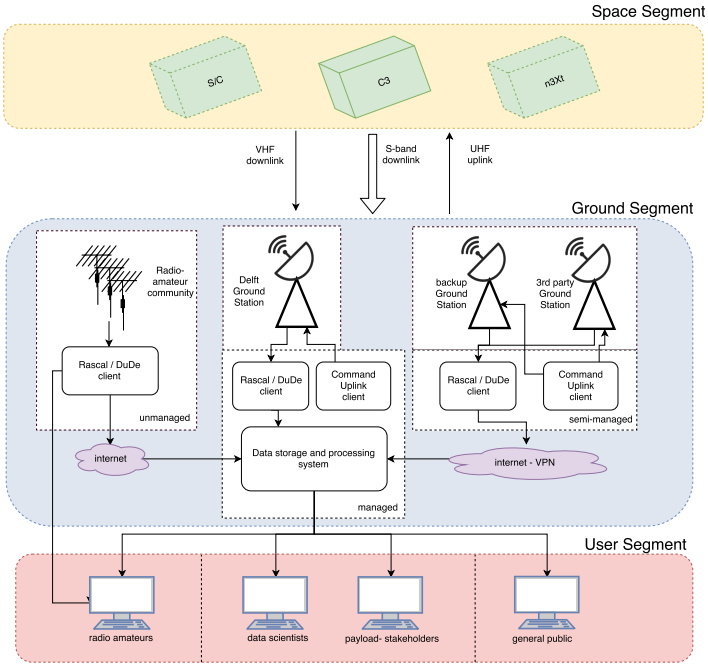

Figure 3.1: Delfi mission segment

high frequency (VHF) (and S-band), and uplink over ultra high frequency (UHF) bands [15]. It should be noted that the Radio Amateur community offer higher support for UHF/VHF bands, and a limited S-band coverage, therefore restricting the high data rate transmissions to the more sophisticated, and often professional ground stations [15]. Initially, the Space-Ground interface was segregated into low and high bandwidth channels, with high bandwidth available at the managed portion of the ground segment. With the recent efforts to lower the hardware costs, enabling a broader S-band acceptance within the community in form of the software defined radio's, BladeRF [16] and HackRF [17] in particular, retailing for a fraction of the profession all-in-one unit price, providing an acceptable alternative at the cost of limited bandwidth capabilities. The adoption rate and S-band capabilities require further research, the interface is, therefore assumed feasible [GN-AS-1].

In both Delfi-n3Xt and Delfi-D3 missions the interface between Space and Ground Segment is established by use of a client application DuDe/Rascal, discussed further in section 3.4. Collected data is stored centrally on Delfi server, providing interface tailored for mission operators and data scientists. It should be noted that the client application serves a dual purpose by performing decoded telemetry visualisations, on-premises of the ground stations (DGS and RA) acting as Ground-User Segment interface. Historically, client application used by Radio Amateurs and spacecraft operations, for direct telemetry processing, due to the server lag. It should be noted that server is the primarily data source for s/c operations when satellite is outside the direct communications links. For the future missions, the situation is unlikely to change due to numerous considerations, e.g. processing and visualisation redundancy, performance, local data access and many others.

3.2. Ground Segment: a geographical view

The Delfi telemetry reception system leverages the radio amateur community for data collection. Shown in the figure 3.2, the coverage is biased with some regions more densely covered than the others. One would, therefore, expect to see extensive data duplications for the densely covered areas, as multiple radio amateurs would receive the same telemetry frames. This is partly true, with roughly 33% of received data being duplicate. The presence of RA's in the area, however, does not guarantee the reception. This is illustrated in figure 3.3, where two consecutive Delfi-n3Xt passes on 28-11-2013 were recorded by three stations only. The bias is dynamic, as RA's participation rate changed over time, as can be seen in figure 3.4.

To visualize the impact of the duplicates, consider figure 3.5 showing the number of duplicate

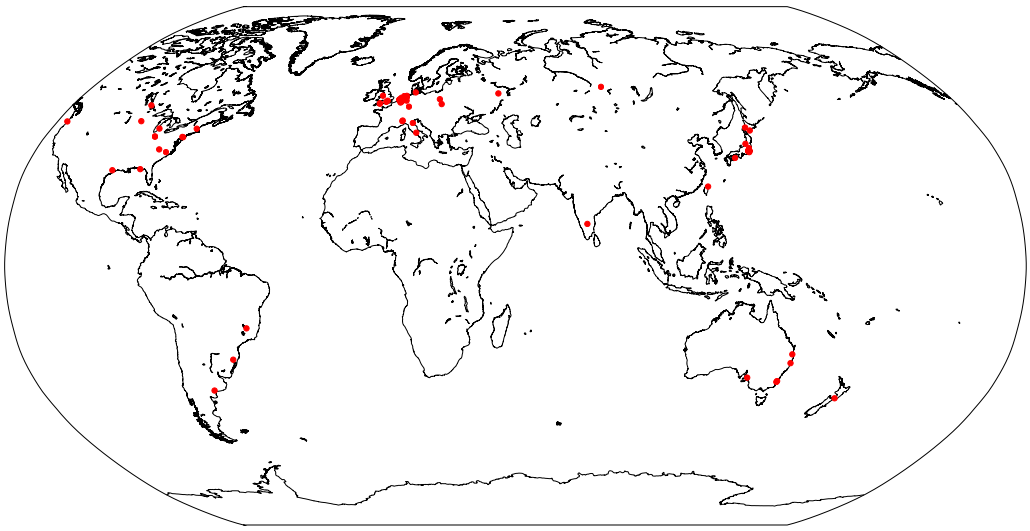

Figure 3.2: Radio amateur locations

frames received per day. Figure 3.6 provides a better metric, showing the normalized count of the data duplication. The high data inflow does not necessarily correlate with the high percentage of data duplication. To bring the regional duplication in perspective, figure 3.7 shows the number of non-unique frames received for each geographical location. The high number of duplicates in the European region is mainly driven by one radio amateur contribution, located in ` close proximity to the Delft GSN.

3.3. Telemetry

The telemetry frame generated by the spacecraft consists of the payload data, such as onboard measurements, along with the status parameters, i.e. housekeeping data of the satellite. Former is required for the science aspect of the mission, while the latter is instrumental for correct spacecraft operations. Additionally, missions, such as Delfi-n3Xt, can contain payload, requiring data to be accessible via client defined interfaces, leading to the requirement GN-DS-14.

3.3.1. Telemetry Generation

Deviating from black-box view of Space Segment, this section focuses on telemetry generation onboard the spacecraft.

The focus lays on the Delfi-n3Xt implementation [18], for being the most recent, most researched and understood by the author. Delfi-n3Xt supports multiple operational modes, as depicted in the figure 3.8 [18], with use cases for each of the primary Transceiver (PTRX), secondary ISIS Transceiver (ITRX) and S-band (STX) radios. The experimental operations are out of scope of the project, as all modes, except ”transmit” PRTX: HK + PL (housekeeping + payload) and ITRX: HK+PL, are ignored by the client application.

Early in the project, this raised the question whether the future ground segment could be required to support other communications modes. For example, considering the linear transponder experiment, with the spacecraft relaying the RA communications, monitoring could be required to perform spacecraft debugging or to quantify the feature use. This is, of course, best achieved with a worldwide coverage, requiring support by the client application.

Speculating on the future missions, one will quickly realize that the operational modes are only one

Figure 3.3: Groundtracks and reception points

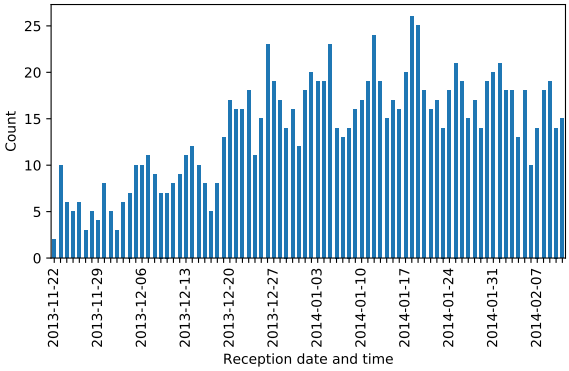

Figure 3.4: Radio Amateur contribution

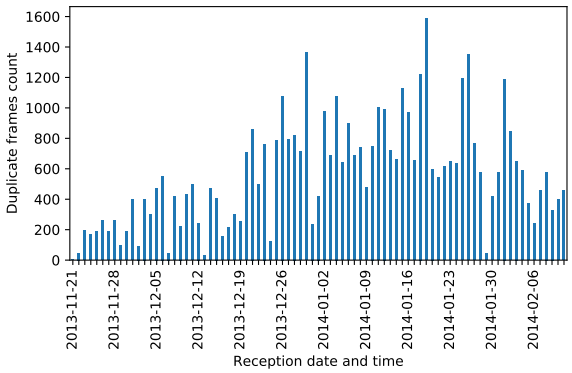

Figure 3.5: Duplicate telemetry frames counts per day

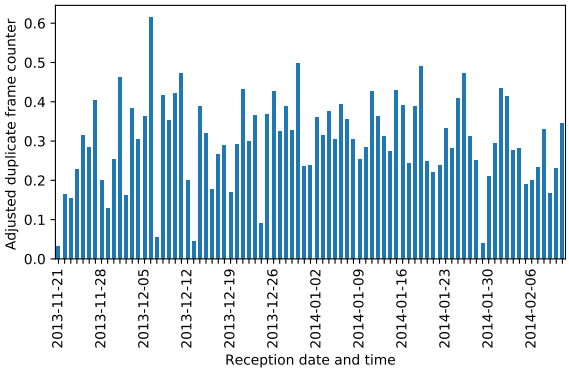

Figure 3.6: Normalized duplicate telemetry frames counter to total number of receptions [%]

Figure 3.7: Unique telemetry frames received per region

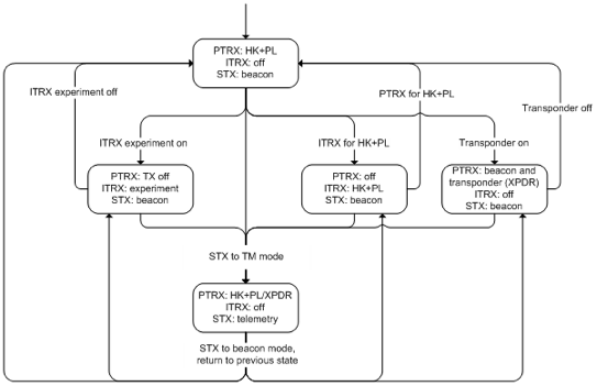

Figure 3.8: Delfi-n3Xt radio operational modes [19]

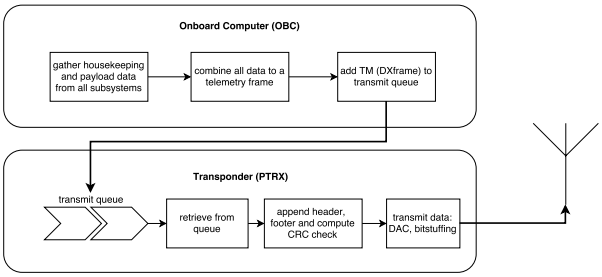

Figure 3.9: Simplified onboard dataflow model [19]

a small part of the picture since the space-system requires a wider downlink strategy. For the sake of completeness and as a measure to foresee the possible system changes, the summary data flow onboard a Delfi-n3Xt spacecraft has been complied in figure 3.9

Clearly, in this approach, the radios are operating in a continuous mode, broadcasting live satellite data, without 'long-term' onboard telemetry storage. This relative simplicity comes at cost of data sparsity, as only the spacecraft parameters measured during pass are captured. A number of different strategies can be defined. For example, the QB50 mission required whole-orbit data: an aggregation of payload data between the ground station passes. The choice of the downlink strategy will influence the telemetry reception rates and volumes, possibly placing processing requirements due to the duration of the pass limitations. System may, furthermore, be required to support alternative modes of operations, requiring a flexible storage and processing solution (requirement GN-PR-2, GN-DS-12).

In case of Delfi-n3Xt, 34% of the received frames were determined duplicate. The amount of rejected frames is not explicitly stored, but can be estimated. The frequency of frames sent by the radio is determined by the on-board computer (OBC), therefore the reception packets frequency can be estimated as:

$$f_{rec} = f_{sent} + \Delta f_{error}$$

With f_{rec} the perceived telemetry frame reception frequency, f_{sent} the send-frequency of the OBC and Δf_{error} term responsible for the frequency deviation due to medium interaction. The deviation can be caused by drift secondary effects such as onboard clock drift or radio signal scatter perceived by the receiving radio, and it is assumed that $f_{sent} > \Delta f_{error}$, therefore Δf_{error} can be ignored. By observing the telemetry packets received by a single RA on a single pass, can be estimated as:

$$N_{theory_{received}} = \frac{T_{begin} - T_{end}}{f_{sent}}$$

With $N_{theory_{received}}$ defined as the number of theoretically receivable telemetry frames between start T_{begin} and the end T_{end} of the satellite pass. Therefore the number of frames lost N_{lost} due to errors can be estimated as:

$$N_{lost} = N_{theory_{received}} - N_{received}$$

With $N_{received}$ defined as number of received frames per pass for each receiver.

Due to all-or-nothing Cyclic Redundancy Check (CRC) validity check algorithm, the packet loss is directly related to the environment noise. Previous work by Milano [19] and Hartano [18] suggest that error correction is not required for Delfi-n3Xt operations due to limited frame size. Figure 3.10 illustrates the ratio of lost-to-recieved telemetry frame per satellite pass aggregated per day for the top 15 Delfi-n3Xt receivers (RAs and GSs). It should be noted that the graph indicates the lost telemetry frames per radio amateur, and does not necessarily mean that the frame are not received from another source. On average during a satellite pass, 50 telemetry frames are received per receiver, then, by selecting the passes with above average telemetry reception rate, the calculated telemetry frame loss

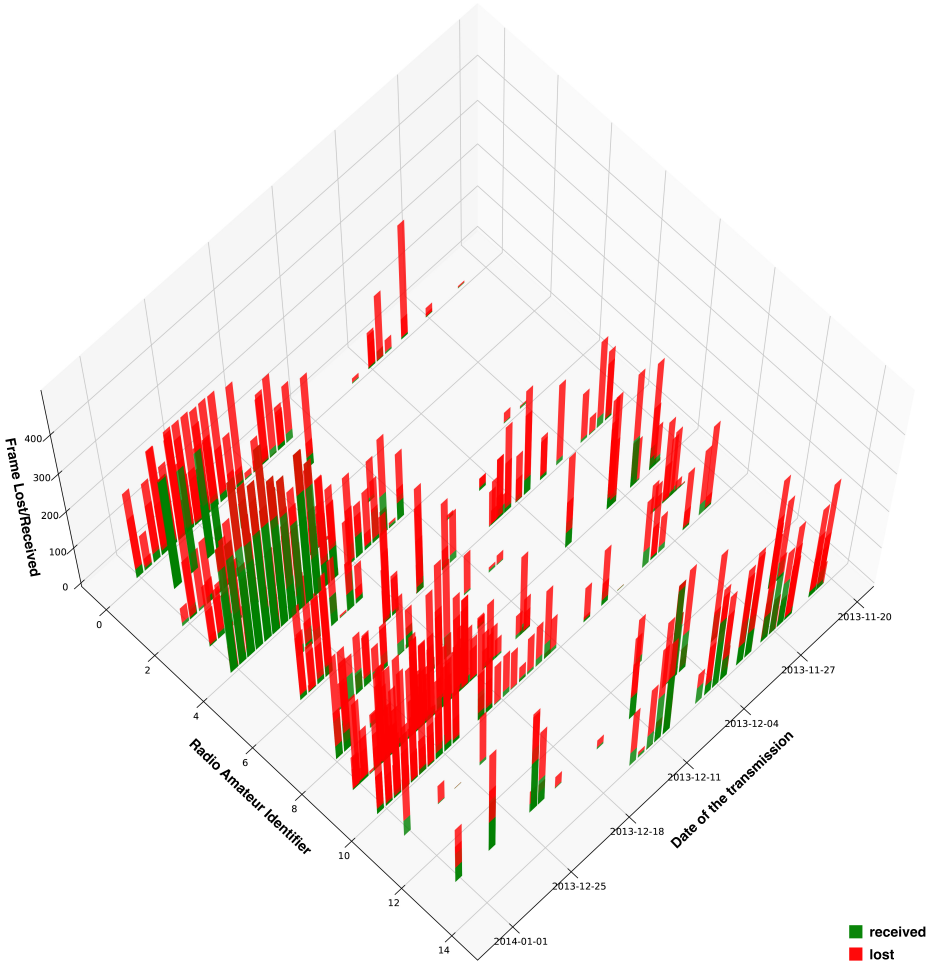

Figure 3.10: Change of the ratio of received to missed telemetry frames in time during intial 3 months of Delfi-n3Xt operation

aggregated over all clients is determined to be 49 %. Which is inherently biased, since the passes meeting the 50 frames threshold accounts for 22.6 % of all the passes (450 of the total 1539 recorded passes). Normalising the threshold to accommodate for the half of all passes, results in threshold of 22 frames, resulting in a packet loss of 88 %. Which is comparable to 78% loss rate found in the experimental results published by McGuire [20]. The high frame loss can be explained by DuDe software malfunctions and bugs.

3.3.2. Telemetry Protocol: AX.25

To provide the background and the need of the telemetry data transformations, this section will focus on data encapsulation and transfer protocols. As shown in figure 3.9, the telemetry data is encapsulated twice. First as a data frame (DF): an aggregate of the payload and system parameters. Second as a telemetry frame with AX.25 encapsulation.

The schema of DF frame, albeit constrained in length, follows a custom protocol, determined during the design phase of the spacecraft. In case of Delfi-C3, the amount of data encodable in the DF frame fit the available data (link) budget, allowing a single DF type to be utilised. For Delfi-n3Xt mission, two DF frames were required.

In case of Delfi-n3Xt, the data serialization follows a custom protocol, which is merely a predefined sequence of spacecraft and payload parameters. The software switches, as well a variety of hardware

Table 3.1: AX.25 model for single link [22]

Data Link	Segmenter	Management Data Link
	Data Link	
	Link Multiplexer	
Physical	Physical	
	Silicon/Radio	

Table 3.2: U and S frames schema [22]

Flag	Address	Control	Info	FCS	Flag
0111 1110	112/224 bits	8/16 bits	N*8 bits	16 bits	0111 1110

metrics, like deployment status, translate natively into binary ”1” - ”0”, while all floating point values ought to be serialised.

Each floating point parameter is defined based on the position within the sequence and bit-length dictated by the measurement precision. The serialization is merely a mapping of the floating point value to an integer using a set of predefined formulas. The resulting (rounded-off) integer, is then natively converted to a binary value. As an example, consider main bus voltage metric with 12 bits allocation and serialization formula:

$$U_{subsystem} = Value_{serialized} \cdot Constant$$

With a constant set to 0.0034 V the subsystem voltage is the range of 0 to 14 volts. The position of the individual values within DF frame sequence is theoretically free to choose, however, in practice the parameters are grouped per subsystem to ease the data aggregation onboard the spacecraft.

The telemetry frames are structured in accordance with AX.25 protocol, to ensure the data-link layer compatibility with available ground stations and radio amateur community. The data-link layer, as defined by the Open Systems Interconnection (OSI) model [21], is the second layer of the communications stack [22]. The layer provides functional as procedural means for data transport between network entities: establishing connection parameters and connecting the clients. And in theory, also being responsible for error detection and correction. With the radio amateurs as the end users, the protocol is designed to deliver the data between two computer terminals regardless the communication link, the physical layer, making the protocol extremely versatile.

As defined by the OSI specification [21], every layer serves the layer above and is served by the layer below. In practice, the communication between OSI layers is achieved by Data-Link Service Access Point (DLSAP). Management Data Link is responsible for the connection and negotiation of operational parameters between the stations. Furthermore, the Data Link logic is responsible for establishing and releasing the connections between the stations. The link Multiplexer is the layer between data link logic and transmitter/receiver: the physical layer. Data from layer three is provided to layer two via DLSAP. When data volume of the frame (UI/I) definition is exceeded, the payload is split into multiple frames by the segmenter.

The payload, or the transferred data, is encapsulated in frames. The AX.25 protocol defines three general frame types, Information (I), Supervisory (S) and Unnumbered frame (U). Information and Unnumbered frames are further refined into Exchange Information Frame (XID) and Unnumbered information frames (UI) respectively. Each AX.25 frame has a specific function, for example, to comply with OSI model or required for establishing the connections. However, in its current setup, the CubeSat cannot establish two-way communication. This leads to a modified, reduced AX.25 specification used for the majority of CubeSat missions today. The broadcast telemetry is based on an unnumbered information frame (UI) with the schema presented in table 3.4 .

Table 3.3: I (information) frame schema [22]

Flag	Address	Control	PID	Info	FCS	Flag
0111 1110	112/224 bits	8/16 bits	8 bits	N*8 bits	16 bits	0111 1110

Table 3.4: AX.25 UI frame definition used in Delfi missions [22]

Flag	The start and the end flag of the frame. Used to locate the radio transmission.
Address	Originally intended for the source and the destination address of the frame, the RA call signs. In case of the CubeSat operations used to identify the spacecraft. Since the one-to-many type of the operation, the address field is invariant throughout the mission.
Control	Contains the information of the frame, such as frame type and the sequence set by segmenter. For CubeSat operations field is set statically to 0b00000011.
PID	Defines the third layer protocol. In case of CubeSat operations, the protocol is undefined and therefore set to 0xF0.
Info	Contains the spacecraft telemetry data.
FCS	Frame-check-sequence, a sixteen-bit integer recomputed by the receiver to assess the data validity. The Frame-Check Sequence is computed following ISO 3309 [22]

The AX.25 specification is not fully utilized. The telemetry parameters transmitted within the info field of each frame, follow a predefined sequence. Without encoding embedded in the Info field, a lookup table is required to extract the information. The AX.25 frames are transmitted in one-to-many fashion (one S/C to many receivers), without frame reception acknowledgement (QoS 0 strategy) no guarantees can be provided for the consecutive frames to be received. Thus, the segmenter cannot be implemented (assumption GN-AS-4). Therefore, the data cannot be sent arbitrarily relying on the AX.25 protocol functionality. Furthermore, due to one-way communication between the satellite and the receiver, any errors in the transmission are irrecoverable. Presence of bit-flips due to a noisy environment, cause CRC checks to fail, rendering the received frame invalid. [23]

3.4. Client Application: telemetry parsing

The client application, i.e. Dude or Rascal, executed on both GCS's and RA's premises, is responsible for data extraction from the radio signal, decoding and transfer to the database. The frame sampling is the initial part of the client based data processing [24]. The Analag to Digital Conversion (ADC) is handled by the hardware, while the application is scanning the incoming sequence continuously, performing the following actions:

1. Scan for the frame flag (01111110)
2. Extracting the data between flags, and dropping if allowed frame length is exceeded

Once AX.25 frame has been extracted, the CRC-check is performed to check for data corruption. The AX.25 protocol provides no means to correct for errors in data transmission. Furthermore, the CRC does not provide a degree of "incorrectness" of the data: the frame is either valid or invalid.

3.4.1. Telemetry Encoding Schema

As stated in the Telemetry protocol 3.3.2, the AX.25 opposes a requirement on the maximum allowed length of the frame, therefore limiting the permitted bit-length of the Info field containing the data. However, in the legacy missions, the frame length was limited by the utilised hardware, rather than the software specificaitons. In case of Delfi-n3Xt, the data link budget could not be met with a single DF frame definition, requiring data to be split into two frames, broadcast consecutively. The consequence of this rigid design is that information, e.g. satellite and payload parameters, sent over the lifetime of the mission are immutable. For example, deployment status of the solar panel is sent throughout the entire mission. While being vital on the initial spacecraft contact, the value need diminishes in time and serves only as a very basic check of the OBC and radio operations.

To resolve the exceeding the data budget, one may consider to:

1. Break the AX.25 length requirement
2. Compress the telemetry data within the frame
3. Split data into multiple frames
4. Apply the re-configurable frame (FlexTLM) definition [25]

In the design phase of the Delfi-n3Xt mission, a custom DelfiX [19] protocol was considered, allowing 10-20% saving but dropping support for Radio-Amateur community [19], rendering the option

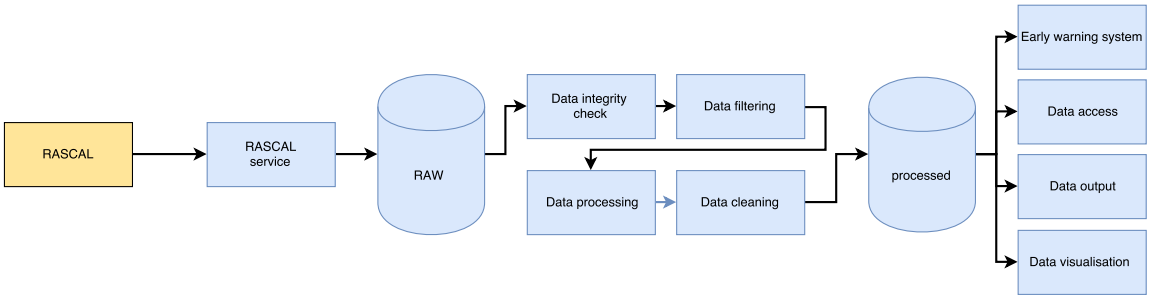

Figure 3.11: Delfi-C3 server application overview

unsatisfactory. The main consideration against the flexibility of the reconfigurable frames as well the telemetry data compression is the required fault-tolerance onboard of the spacecraft. Any errors in telemetry encoding corrupt the data and the downlink capability. Schoemaker [25] argued that reconfigurable telemetry risks can be mitigated by not allowing fully custom on-demand encoding schemes, but by predefining metrics and metrics groups and selecting them based on the scenarios:

Time defined frame identifiers
The frame definition sequence depends on the time of the onboard real-time clock. To decode the frame, encoding is selected based on the time value embedded in the frame. This is required, as a onboard clock will drift. (Bias + linear)

Unique frame identifiers
Each frame definition has an unique id. Each frame can be decoded, by mapping the frame id to the accompanying decoding scheme. (Similar to Delfi-n3xt)

Packet group identifiers
The concept is based on the grouping of similar parameters or subsystems and identifying data bundles by an id. Processing identifies the id, and the predefined map of parameters within each group and position of the groups within the payload.

The final solution proposed by Schoemaker is based on a sequence control, defining the next transmitted (or received) frames. The specifics of the approaches are less relevant within bounds of the discussion and serve merely as a proof that each satellite mission may require drastically different approach with respect to the frame definitions to balance the overhead, robustness and the flexibility. This leads to the GN-PR-15 and GN-PR-14 requiring processing software to identify the DF encoding in a flexible manner.

Analysis of the Delfi-n3Xt data showed a consistent pattern of data reception: a significant number of consecutive frames received in succession.

3.5. Server Side Application: data processing

The need for the server-side application stems from three elements: persistent data storage, telemetry data handling, e.g. processing and data distribution to the project stakeholders. System exposes an interface, a central data ingestion point, that serves as an entry point for all DuDe/Rascal clients. The telemetry handling consists of data verification, binary operations, e.g. bitstuffing and least-significant-bit (LSB) re-shuffling, validity checks and is ulimately responsible for duplicates mitigation, i.e. filtering and discarding. The data distribution portion of the application is responsible for data visualisation,a web based dashboard, as data export options such as Excel spreadsheets.

Albeit serving the same purpose, the actual implementation of Delfi-n3Xt and Delfi-C3 servers differ, as shown by the functional data flow diagrams in figures 3.12 and 3.11. Please keep in mind that the figures are simplified, with all of the secondary tasks such as users management and client-application authentication removed.

Based on the prior analysis of the documentation and source code [7] [9], a list of points of interests

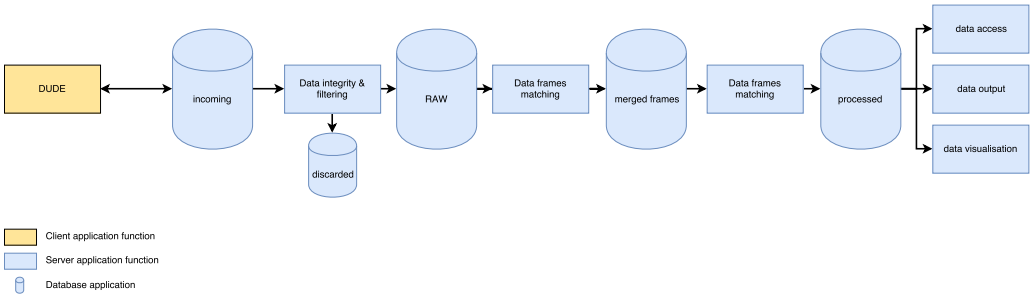

Figure 3.12: Delfi-n3Xt server application overview

has been compiled, shown in a reduced from below.

- Multiple databases
- Database exposed to the internet
- Batch processing via Cron-daemon
- Quasi-hardcoded frame processing specification
- Telemetry is decoded only if both frames are available
- Binary data is stored in base-2 format as String object
- Data parsing is based on base-2 String representation
- The system is built on assumption of immutable DF schema definition
- DF schema definition is not exchangeable with the client application
- No standardized Application Programming Interface (API). Modifications to the database (DB) require changes of the client and server applications.

The prior effort in form of a literature study added in appendix C both Delfi-n3Xt and Delif-C3 implementations were marked as undesired for the future missions under their current form. The decision is to a high degree the consequence of the subsequent assumption:

Satellite broadcast telemetry data with a single schema that is fixed and immutable

During the design phase of the Delfi-n3Xt mission, the telemetry budget was exceeded. To comply with the RA community by not breaking the AX.25 length requirement, the telemetry data was split into two data-frames, transmitted consecutively. The designed telemetry processing software was, however, not intended to support multiple definitions.
In theory, the most logical processing approach is based on the decoding telemetry frames independently per frame type, as shown in figure 3.13 (option *A* in the figure). However, in the Delfi-n3Xt implementation option *B* in the figure was utilized, where the consecutive telemetry data frames were merged based on frame counter and type, and processed as a single entity.
Consider a use case where one of the matching telemetry frames was not received. This caused the other matched part of the data frame, that was received correctly, to be discarded.

The choice for solution *B*, is a classic manifestation of a monolithic architecture. Frequently observed in the industry [26], the system flexibility is achieved by suboptimal software adjustments, since the proper adjustment would require a complete system overhaul.
Globally, the shortcomings of a software application can be accounted by the insufficiently defined requirements at the initial stage of the project, or change of the requirements throughout the mission or the project lifespan. An example of former, are the missing user interface (UI) elements on the dashboard of spacecraft operators, such as operational limits of the satellite parameters and lack of the logging functionality for the telecommands sent to the spacecraft [9].
The monolith design manifests in other parts of the systems too, for example the telemetry frame processing component accepts one single frame definition. As the direct consequence, the application is unable to facilitate the changes in the processing and accommodating for 44.5% of dataloss.

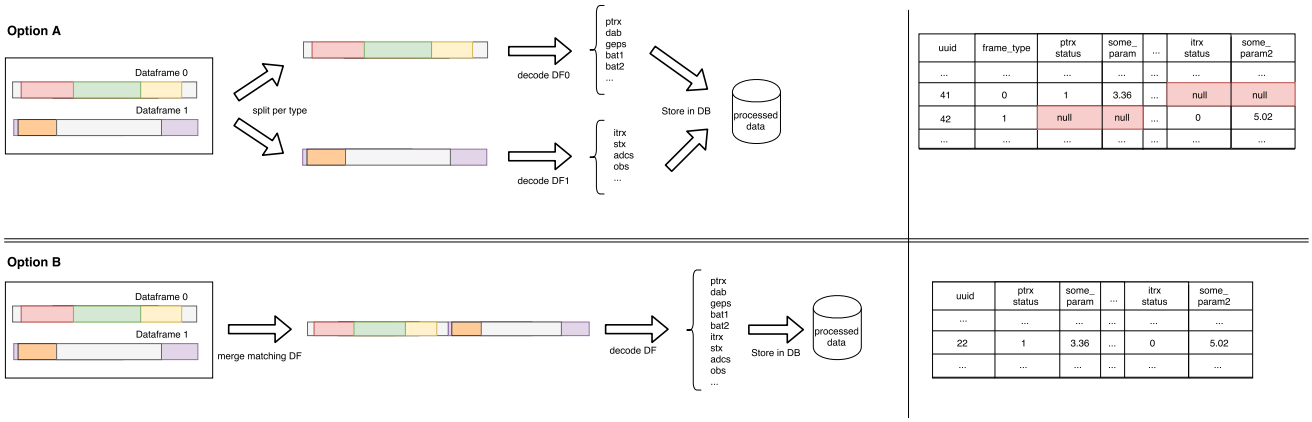

Figure 3.13: Delfi-n3Xt telemetry frame processing alternatives

Analysing the decision process behind options *A* and *B* show that the amount of work required to implement the option *A* outweighs the added benefit. This decision ultimately lead to a purpose-designed labour-intensive to adjust Delfi-n3Xt system.

In theory, the purpose designed systems shine in the trade-off by the high mark for the applicability (appendix C). However, in practice, as illustrated above, this is not the case, as implementation of new features and functionality is cost-ineffective. The designed system ought to allow addition of custom and potentially advanced processing scripts [GN-PR-21].

3.6. Data Volumes

This section focuses on the usage statistics of Delfi-n3Xt to serve as a baseline for data throughput and storage requirements.

The computation of the system throughput is based on the timestamp given to each received frame by the server storage system upon ingestion. Illustrated by the figure 3.14, the throughput or the data ingestion rate, varies significantly in time. Comparison of the storage system and client application timestamps provides a metric for latency, illustrated in the figure 3.15 and described by equation 3.1.

$$\delta T = T_{received\_server} - T_{received\_client} \tag{3.1}$$

The average value of the latency is 1.0*s* with a standard deviation of 1.5 *s*. Practically, 75% of all data is ingested by server within 1*s* of reception by the client application. The previously established latency metric is proven to be unreliable and potentially biased, as a difference between server and client clocks has been determined. Figure 3.16 provides an overview of faulty timestampted frames per client, where based on the timestamps values, the frame was received by server prior to the client application. This analysis leads to the requirement GN-PERF-1 and GN-PERF-4.

With client application performing writes directly to server hosted MySQL server, the queueing and write locks, e.g. caused by the multiple concurrent writes, would results in a correlation between observed latency and data ingestion rates. The analysis did not reveal any significant correlation.

Figure 3.17 illustrates the data ingestion rates, expressed in frames per hour. Looking at a higher precision metric of frames per minute, identified two global peak load occurrences on December 25th and January 19th, plotted in figures 3.18 and 3.19. Finally, the figure 3.20 illustrates the highest peak load measured: 7 frames/s on January 19th at 10AM.

Notably, the measured peak load of December 25th, does not correlate with a peak in radio amateur participation rate as shown in figure 3.4, nor a high duplication count as shown in figure 3.5, while the January 19th does. This lead to the requirement GN-PERF-2, pushing for minimum ingestion rate of 10 frames per second.

To express the data storage system requirements, one should convert the frame count to memory based unit system. This is achieved by multiplication of the frame count metrics by the object representing the telemetry frame with the encapsulated metadata size. The metadata, such as reception timestamp and RA's information, is magnitude smaller in size that the actual telemetry data and will be ignored for the further analysis.

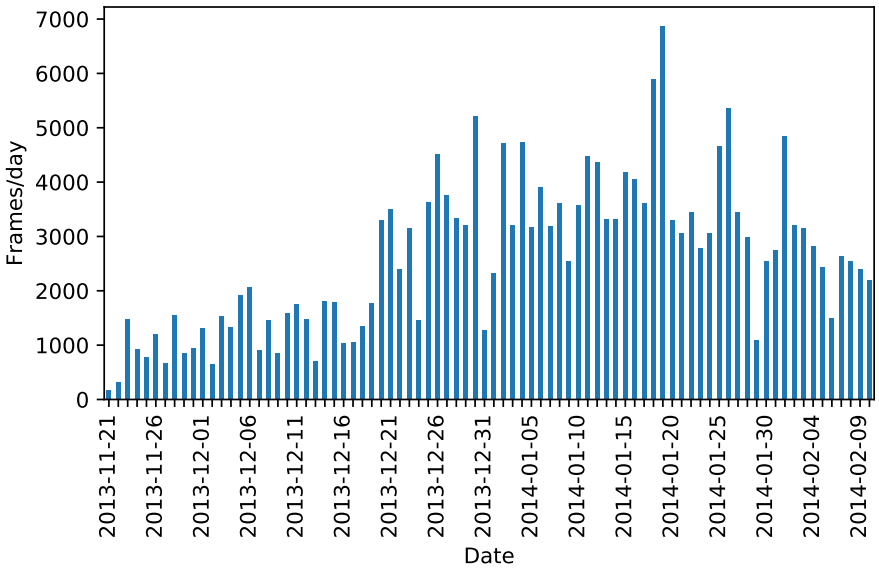

Figure 3.14: Delfi-n3Xt reception rate

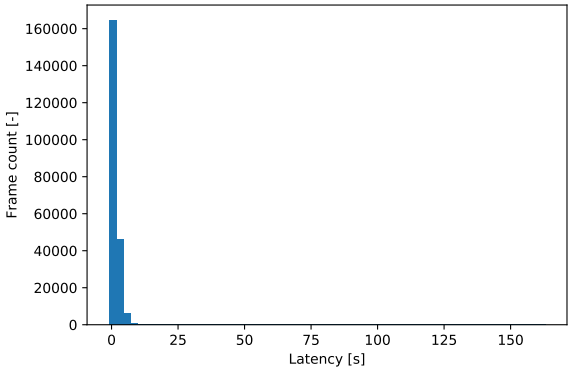

Figure 3.15: Latency histogram

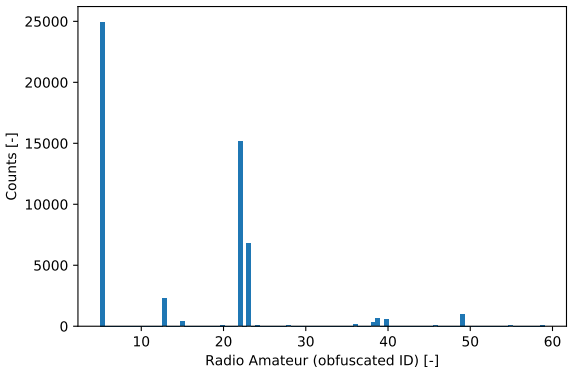

Figure 3.16: Count of frames with Server reception time prior to the client reception timestamp (error) per radio amateur

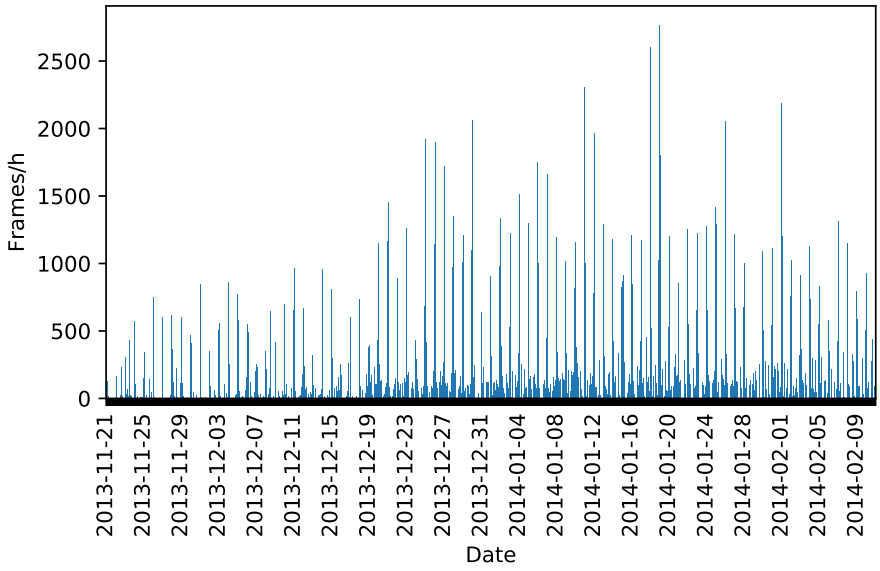

Figure 3.17: System throughput throughout the mission

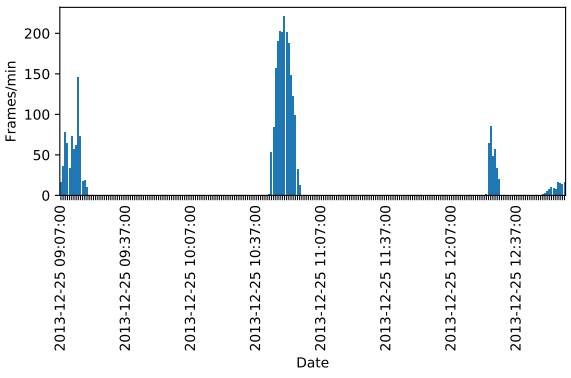

Figure 3.18: Peak throughput: december

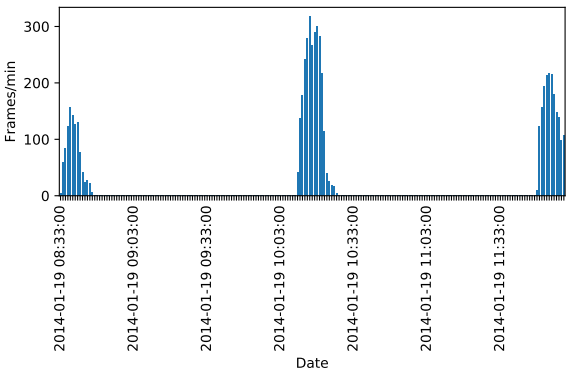

Figure 3.19: Peak throughput: january

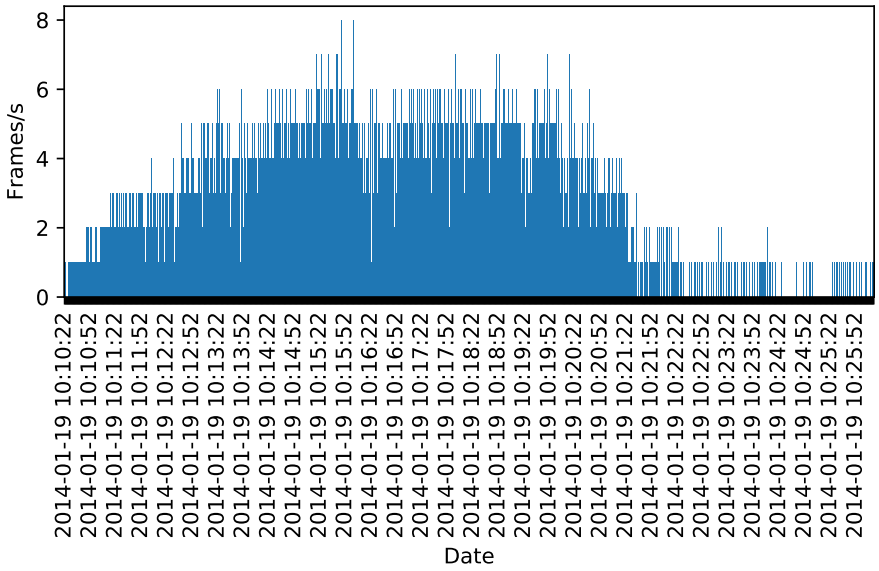

Figure 3.20: Peak load january

A single AX.25 frame as utilized in the past missions [25] is 2400 *bits* long, or 300 *bytes* or 0.0003 *MB* in size. With the previously determined peak load of 8 *frames/s*, resulting in 0.0024 *MB/s* or with peak load of 300 *frames/min* 0.09 *MB/min* for Delfi-n3Xt mission with generation rate of one frame per second.

The analysis of the Delfi-n3Xt and DuDe applications showed use of String objects for data storage and processing. String, an object or a data type, depending on the programming language, is represented as a sequence of characters used to store text with a known, predefined encoding, e.g. UTF-8. Depending on the programming language, String can be overloaded as an array of characters, allowing the array-like methods. A deeper look at the client and server source code shows the reason for this odd decision. Data encoded in the frame is defined in terms of its position and length. For example, extract first 12 bits of a DF stored in string format c,an be performed via `frame[0:12]`. While, the binary format would require binary operations, e.g. bit-shifts. String, being an object, requires higher memory allocation, adding unnecessary overhead and dependency on the character encoding.

The actual memory allocation is difficult to predict, as it depends on the operating system, programming language and more. In python, the shallow command `sys.getsizeof` will show direct memory taken by an object, excluding pointers, i.e. contents of the array, fortunately, using the deep memory libraries such as `pympler`, the memory utilization can be assessed more reliably. In python, the String serialization of a DF allocated 1841±8 *bytes*, while the digit notation, independent on the base, only requires 264 *bytes*. The analysis showed that in Python version 3.6.4, the memory allocation of a string object is 49 *byte* with an additional byte per stored character. The deep memory approach, showed utilization of additional eight bytes, source of which could not be determined. In short, the change of the data format from String to binary will decrease memory consumption seven fold. Even with String inefficiency, the adjusted data ingestion rates are negligible low circa 0.02 *MB/s* and *0.2 MB/min* under peak load.

3.7. Discussion and Requirements Refinement

The major lesson learn from the inital Project study, is that, albeit a product of a single institution, two consecutive spacecraft designs, being similar in operation and offering comparable functionality with regards to the telemetry, may require drastically different telemetry processing requirements.

Driven by tight datalink budgets, the frame definition or schema is assumed unlikely to be embedded into the telemetry frame body. Furthermore, due to the uplink complexity considerations, it is higly unlikely that two way communication and reception acknowledgment can be implemented reliably for all of the clients (including RA). Combined with the missing schema of the telemetry sequence, renders

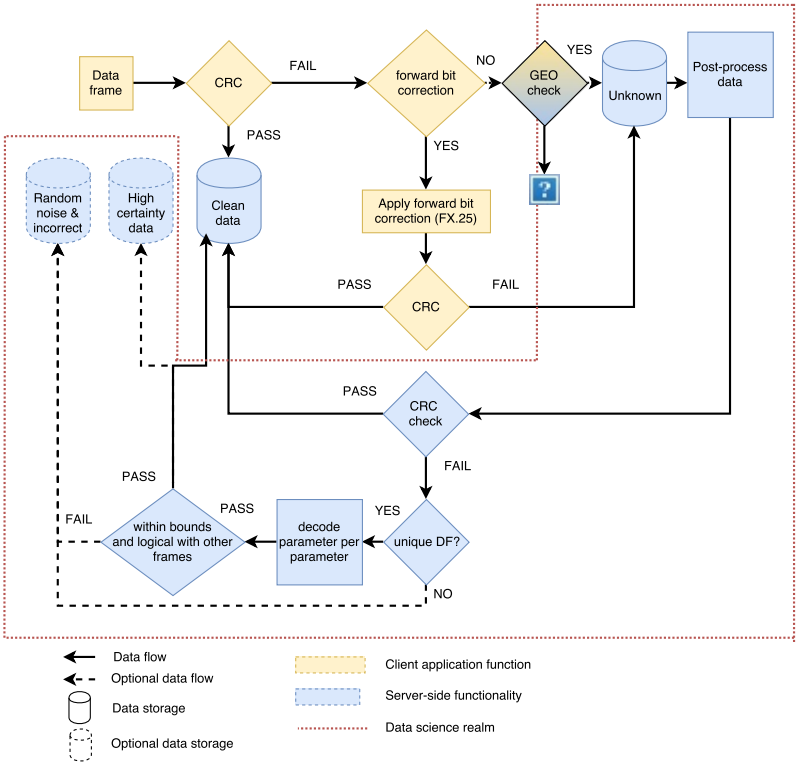

Figure 3.21: Example of data science: error correction

AX.25 segmenter prone to data loss. Aforementioned leads to the assumption that the ”arbitrary data” approach is unlikely to be used, hence, system will be required to have a priori knowledge of the telemetry data encoding (Assumption GN-AS-4).

Considering the future missions, with the compatibility requirements towards radio amateur community, AX.25 will likely be supported in its current form, with the info-field contains payload with a predefined encoding. Section 3.4.1 showed that the schema within the info-field is not immutably defined and will likely change during the design (see section 3.5) or possibly change in the operational phase. The proposal of the flexible telemetry schemas [25], shows a variety of drastically different approaches, requiring drastically different processing techniques, forcing the system to be easily adjustable, flexible in operation during the mission lifetime.

The operational testing of the forward correcting FX.25 protocol, performed by Stensat Group [20], showed that merely 15 % of the telemetry frames received were decodable with AX.25, while with the forward correction up to 46 % . The use of the forward error correction, due to its additional wrapper, is often negated by the lower throughput [27], making protocol useful primarily for environments with the noise values greater than BER 10^{-4} [28][27]. This leads to the requirements GN-DS-15 and GN-PR-20.

Post-processing discussion

It can be argued that the lack of error correction in AX.25 can be solved with data post-processing. As discussed in section 3.4, to distinguish frames from the background noise, the application continuously scans for flags (01111110), followed by the CRC check. In case of CRC check failure, but an address field match, it can be assumed that the telemetry frame is partially correct. With a known frame length, the CRC check failure can be reduced to either arbitrary noise or bitlfips. Number of metrics such as satellite presence with the reception range of a RA or ground station, influence the probability of frame correctness and can be used to improve the recovery model. Although requiring further investigation (e.g. bitflip influence on bitstuffing) the concept, in author opinion, is feasible. Looking forward, attempt to post-process frames requires a versatile system able to cope with uncertainty in portion of stored data, allowing classically data science tasks from the ground up (GN-PR-25, GN-PR-26).

The aforementioned lead to the concept outlined in the figure 3.21. Aimed as an attempt to investigate the needs of a post-processing system, serves as an example of a basic data science task that

any modern data driven system ought to be capable of, ignoring the bitstuffing and SDR use effects. The concept builds on top of the frame filtering based on the radio amateur parameters: location, time and the validation that the satellite is within reception range. The metris are easily retrieved and are readily available in the Delfi-n3Xt system.

The research leads to the question on the function of the client application. It can be argued that, by utilizing the post-processing error correction technique, hence by disabling the CRC-check, the processing can be performed solely on the client-side, due to possible high data volumes. This claim requires further research; however, any modern distributed data processing system, is capable of ingestion range of MB to GB per second, a far cry from 0.2MB/min peak load of Delfi-n3Xt.

3.8. Conclusion and Preparation of the Next Project Iteration

In conclusion, the legacy Delfi-C3 and Defli-n3Xt telemetry processing systems tend to show a high dependency to the satellite mission. It can be argued that the telemetry server is purposely designed with a single mission in sight. Theoretically, it could maximize the mission applicability C; however, after analyzing the source code, it is clear that the applicability comes at the cost of reusability. The ability to build on top of an existing framework is generally accepted as the most effective way to expand system capability as well as to understand the (high-level) system operation. The effort of writing systems from scratch and optimizing the low-level functionality will unlikely benefit, as it will come at a higher cost. This and many other factors, stimulate the use and reuse of the software, which comes at the cost of transparency, as the developers may be unaware of the assumptions, limitations and possible compatibility issues. This leads to the idea of software frameworks, which by bundling libraries along with a particular methodology stimulates developers to follow the best practices for a design particular methodology.

Therefore a drastically new approach is required to ensure the reusability, pushing the system capabilities, allowing the data science tasks while increasing the quality of the final product over time.

The analysis presented in this chapter, combined with more in-depth literature study in Appendix C lead to the conclusion that the legacy system showed two major areas of interests:

1. Data Storage
2. Data Processing System

Both the Data Storage and Processing systems used in the legacy applications are rigid towards the data schema, requiring both databases schema as code base changes to facilitate for changes in the telemetry definition. The Processing System disallowed flexible telemetry processing and was found error-prone in both legacy systems.

The goal of the second project iteration is to investigate possible solutions for both areas of interest and refined the requirements for both subsystems. The Data Storage system and Processing are presented in chapters 4 and 5 respectively.

4

Data Storage

The assessment for re-use of the legacy telemetry system for the future mission, reveals system limitations towards the processing and the storage of telemetry data. The change in storage requirements throughout the mission, has three distinct effects. The data can be seen as arbitrary or missing by the storage system, or contain not previously defined data types, all of which inevitable leads to performance implications [29] [30] [31].
The storage systems are developed to solve specific needs, e.g. write/insert performance, complex data aggregations, application specific data formats, objects relationships and many others, what lead to a large variety of systems and techniques available for use today. The section focuses on three conventional storage options: the file system 4.2, database 4.4,and object data store 4.3 systems.

4.1. Categorisation

Prior to the study of the available storage system, the scope of the storage system has to be defined. As illustrated in figure 4.1, globally storage systems can be separated into two broad categories: Centralized and Distributed. Centralized storage is defined as a single node system, while a distributed system denotes multi-node system. The distributed category is further divided into homogeneous and heterogeneous depending on the node environment similarity. The homogenous category implies the use of identical storage software on all of the nodes. In contrast, the heterogeneous systems may be created with the use of different software systems and rely on an additional system component responsible for integration and control of the storage functions. Both categories are further categorized based on the mode of operation, systems requiring a coordinator to control or an aggregate/router controller are non-autonomous and multi-system categories respectively.

The centralized storage systems are out of the scope of the project, due to lower availability and hence lower reliability, hence implicitly failing the performance and availability requirements GN-PERF-1, GN-PERF-2, GN-PERF-4 and GN-QA-AVAIL-3. The use of the heterogeneous system, i.e. a mix of various storage technologies, is an opportunity to integrate with legacy and currently available on-premise systems. The approach increases system complexity both on software as operational levels, which is known to affect quality, i.e. maintainability, and reliability [32].

The scope is therefore limited to Homogenous Distributed Systems.

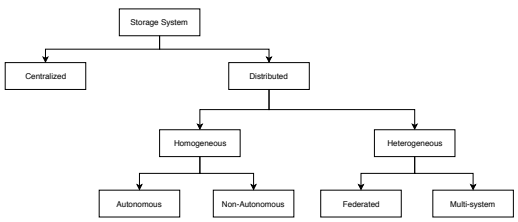

Figure 4.1: Distributed Storage systems

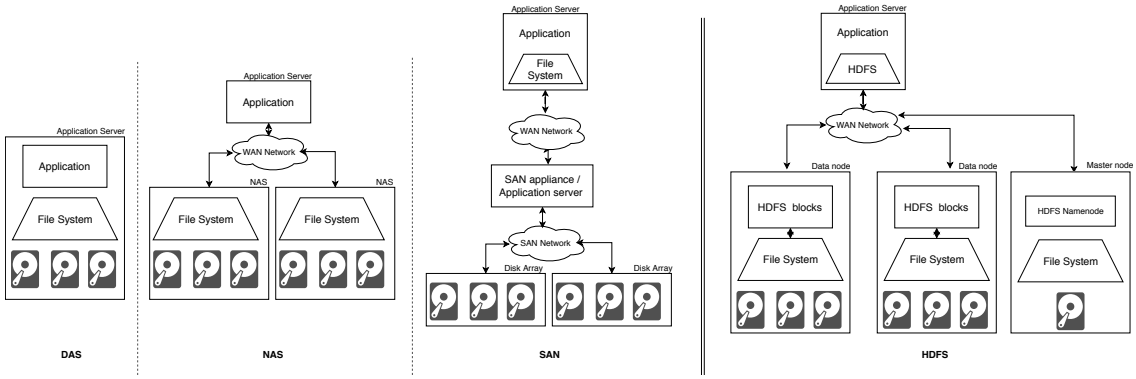

Figure 4.2: File System Storage

4.2. File System

The file system storage provides a hierarchical data store in form of files placed inside a folder structure. The file system may be local, handled by the operating system, or distributed across multiple physical machines in form of shared drives and folders. Within an enterprise infrastructure, one can distinguish between Directed Attached Storage (DAS), Network Attached Storage (NAS) and Storage Area Network (SAN) systems.

In the DAS systems the storage, i.e. the disks, are attached to the application server directly via hardware-level connection (SATA, SCSI, SAS ...). Therefore, any machine with a hard-drive and operating system is classified as DAS. Generally, DAS systems provide a cost-effective solution for small applications and can be tuned for high data throughput. It should be noted, that in many cases DAS stored data is accessible only to the application server. Therefore, the scalability of DAS systems depends on the application and is primarily performed vertically by adding disks.

For the shared storage systems, one can distinguish between File Level Storage and Block Level Storage solutions. The File Level storage, as seen in the Network Attached Storage systems, provides a DAS-like interface over the WAN, by mounting the NAS devices as a local drive abstraction on the application server/client. The Block Level storage, is a low level storage system, comparable in mechanism to the cylinders of a hard drive controlled by the (OS) file system. In Block Level storage, the hardware devices form a SAN network, allowing SAN devices to interact with the drives directly.

While the NAS systems perform Create, Read, Update, Delete (CRUD) operations directly on the File System spanning over the attached drives exposing the FS directly to the client applications, the SAN systems perform low-level, block manipulations on the connected drives, exposing an unified protocol to the client applications providing an abstraction of a file system. In practice, the NAS systems are connected directly to the WAN networks, while the SAN's create a storage network that is exposed to WAN via controller node(s).

In terms of scalability, the NAS systems scale both vertically by adding storage disks, or horizontally by adding new machines providing additional interface to the WAN networks.
Generally, the NAS systems are deployed as a cluster, with a distributed file system exposed to all client applications, providing access to data, regardless the node client is connected to. The storage hardware is often managed by OS that is accessible to the host node only. Any node-to-node operations require additional logic, a backbone or WAN networking which in some cases is handled by the operating system.

As shown in the figure 4.2, the SAN systems, form a network of storage drives, allowing a single disk to be accessed from multiple nodes. Generally, the SAN systems are scaled horizontally, allowing additional drives and appliences to be added to the SAN networks.

The trade-off between SAN and NAS is ultimately handled by the service providers. For sake of the argument, from a high level, both systems provide the same functionality, i.e. the file system access for the application, albeit with a different level of performance.

The popularity of data science, requiring significant amount of data for the prediction model accuracy, gave rise to a variety of new technologies. First, due to the data volumes and redundancy constraints, the datasets are split over multiple machines. Secondly, aggregating and moving large

datasets from the storage to the processing machine(s) is not always feasible, as large datasets cannot be loaded into RAM and may require more disk space than locally available [33] . Additionally, moving large datasets require fast networking, affecting any other network dependent systems, for example, data ingestion.

The challenge is solved by reversing the process, instead of moving the data, performing the processing closer to the storage: near-data computing (NDC) [34]. One of the popular frameworks utilizing the NDC approach is Hadoop, with underlying Hadoop Distributed File System (HDFS) providing a redundant distributed storage on a commodity hardware. HDFS operates in Master-Slave architecture, with one Master, i.e. the namenode, controlling the CRUD operations on all Slave machines, utilized for storage. Similar to the DAS systems, each node is a single machine with directly attached drives running the HDFS application and, potentially, the processing jobs. Similar to the SAN systems, the stored data is saved in form of files, split into lower-level blocks of fixed length distributed across the data nodes. As shown in the figure 4.2, the blocks are stored directly on the OS file system, similarly to the NAS systems. Furthermore, with every node connected to the WAN network, the stored data can be accessed by connecting to any of the nodes via client application that exposes a programmatic interface in form of file system abstraction.

4.3. Object Storage

The storage solutions described in the previous sections expose a file system to the applications, storing data hierarchically as nested folder structure. In many cases, the hierarchy is not required, leading to the data being stored in a single folder. The OS journaling performance, linking the file locations (blocks/cylinder) to the file system, is influenced by the file counts influencing the OS performance and all dependent applications.To avoid the situation, data has to be scattered across multiple folders, increasing complexity without added value.
In many cases, data is machine generated, modified and used and can therefore be stored natively in the application specific object. As hinted by its name, the object based system allows data to be stored in the raw object form, directly on the storage medium, i.e. disks. Utilizating the low-level storage, the storage organisation is flat, allowing objects to be stored and retrieved via an unique identifier and/or additional metadata.

4.4. Database Storage

Database Management System is an umbrella term for software applications that handle and store collections of data, built from the need to accommodate the varying data types, structures and relations, as well data retrieval and querying.

The database models aim to logically structure the data, based on the application operations and data handling. For sake of simplicity, this document will focus on the two major models: the Relational, loosely known as RDMBS or SQL systems, and Model-less or NoSQL, databases.

4.4.1. Relational Database Management Systems (RDBMS)

The Relational DataBase Management Systems (RDBMS) are database system that store information in accordance to the relation model based on the normalization principle, while exposing a SQL-language based querying abstraction. The data stored in the database follows a schema with write operations performed in accordance with ACID or BASE principles, based on the deployment model and application logic.

Normalization

The Structured Query Language (SQL) is developed to solve data querying needs. Due to various historic reasons, the querying of the RDBMS systems is optimised towards on-demand querying, or random-reads. This is achieved by storing information in form of relations, that can be retrieved by mapping of the normalised data. In practice, the normalisation is achieved by separation of the repeating data into a set of inter-dependable data collections, or tables, allowing contents of each to be linked using join operations on the (unique) identifiers or keys.

When applied correctly, normalisation prevents information duplication, facilitating to the overall database consistency. For example, a client's email address may be stored in multiple locations on the

database: e.g. the billing and personal information tables. When developing functionality for email address update, it's inherently easy to overlook one of the locations, e.g. billing, and create an anomaly. The problem is solved with normalization, dictating the user information to be stored only in the user information table and referenced for billing purposes via a query, i.e. JOIN operation.

Schema

The majority of the RDBMS systems require schema's, the definitions for the tables of the database. The schema defines the table columns, data types to be stored, as well any additional parameters such as data uniqueness, counters, (foreign) keys and more. The schema directly influences the efficiency of the storage medium and (random) read-write performance that is to a high degree driven by the efficient RAM memory allocation and low level optimisations. The data type definitions are required internally for querying, e.g. table joins as well computations and logic comparisons.

The schema enforcement is strict, allowing only the matching data to be written into the database tables. Dictated by ACID and BASE principles, presence of errors automatically discard the entire transaction.

ACID

Due to historically high storage and computational costs, many of the database systems were designed to be deployed as a single database instance. To ensure reliability and robustness of the transactions towards possible hardware malfunctions, a number of safeguards has been designed and implemented over the years.

Originally proposed by Haerder [35] in 1983 as a set of principles to facilitate database recovery, ACID became an instrumental methodology to guarantee data validity under various circumstances, ranging from software to hardware errors and malfunctions. The transaction, as defined by ACID, consists of a set of actions, required to be performed indivisibly. According to the principle all actions ought to be executed correctly, or the complete transaction is rolled back, bringing the database to the pre-transaction state. The concept is based on four properties:

(A) Atomicity The transaction is either all-or-nothing: either all actions succeed or the transaction fails. An aborted transaction does not have effect on the state of the database.

(C) Consistency By design, the transaction cannot bring database into an invalid state. The state transition, can only occur if transaction is valid in accordance to defined rules and database constraints, therefore preventing illegal results.

(I) Isolation ACID dictates that the events of a transaction ought to be hidden from other active transactions. Specifically, this means that concurrent transactions are guaranteed to return the same result as if they would have been performed consecutively.

(D) Durability Once a transaction has been committed, the system will guarantee permanent storage and robustness towards any subsequent failures.

BASE

Keeping data in sync across multiple machines or database instances, is a well studied problem [36] predating the ACID principles [35]. Until recently, working with a distributed RDBMS systems due to ACID concerns required a two-phase commit [36]. As suggested by its name, the commit is performed in two steps. First, transaction is pre-committed to all of the database servers, if action is determined valid by all of the nodes, the commit is agreed and finalised in the second step. If transaction is invalidated by one of the nodes, the transaction is rolled back on all of the nodes. The invalidation can occur due to various reasons, for example, due to an unexpected downtime of one of the database servers. The risk of the downtime mid-commit increases with the number of servers. For example a two server database cluster, with 99% uptime or 8.7h/year downtime per individual machine brings the total system availability to 98.01 % or 7.2 days/year, reducing the availability in twentyfold!

In many cases, the availability is preferred over the consistency. Opposed to ACID, BASE (Basically Available, Soft state, Eventually consistent) is optimistic towards CRUD operations and accepts temporary database inconsistencies. The availability is therefore achieved by tolerating failure.

(BA) Basically Available Basically available does not guarantee availability, but imply that portion of the data is available with servers downtime, i.e. that losing a node will not result in complete data loss.

(S) Soft state Soft state indicate that the state of the database may change over time, even without external inputs. The eventual consistency implies that parts of the database may be out of sync, and can be updating at any given point of time.

(E) Eventually consistent Changes of the database state, due to clients input, are not executed directly on all of the nodes and take time to propagate.
The applicability of the BASE approach depends primarily on the application, for example, a bank application should not allow consecutive withdrawals exceeding an account's funds; however, a processing time between withdrawal from one and addition to another account is perfectly accessible.

CAP

Maintaining consistency, hence providing identical read results across multiple nodes is challenging. ACID ensures consistency, but comes at cost of the availability. BASE tolerates networking inconsistencies while maximising the availability, but loses in the consistency. Ensuring both availability and consistency is feasible, but requires an ideal network availability [37].
Published by Brewer in 2000 [38] the trade-off between aforementioned is called CAP theorem: Consistency, Availability and Partition tolerance/resilience. It should be noted that the consistency (C), deviating from the earlier ACID definition, defined as consistency of results from reads across all nodes rather than a consistent of the database as defined in ACID. The availability (A) implies the minimum downtime, but also quick access to the data. The Partition Tolerance (P), is system tolerance towards the network related malfunction between database nodes. Gilbert [39] states that '...a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost ...'. Brewer and Gilbert [38] [39] argue that in a distributed system only two out of three properties can be satisfied. This leads to three types of distributed systems: Availability-Partition Tolerance (AP), Consistency-Partition Tolerance (CP) and Availability-Consistency (AC).

The two-phase commit (2PC) follows the CP methodology, forfeiting the availability for consistency and partition tolerance. In theory, a successful transaction will always result in a consistent state across the cluster, regardless of the network. Specifically, any network induced issues cause partitioning, invalidating the transactions and the partition(ed) (node(s)).
In practice, the 2PC requires a single coordinator, which for example, due split-brain can lead to inconsistencies [40]. Split-brain is the situation where multiple database nodes are assigned primary (Master) roles, effectively acting as two separate clusters. Additional issue of the 2PC design is the blocking. Failure of the controller during the first phase, i.e. the pre-commit, locks the nodes until the controller is recovered. The recovery will depend on the RDBMS system, the configuration and the failure mode, which in some cases require manual intervention. The problem can be mitigated in various ways, for example via three-phase commit, setting a time-out of the operations.

Deployment

Following the Moore's law, the computational and storage costs decrease in time and reaching point to be negligible in comparison to the value of the stored data [41] and the costs associated with the system downtime. The mission critical systems are rarely deployed on a stand-alone machine, with the onsensus leaning towards redundant, High-Availability (HA), Cloud hosted FaaS and other distributed schemes.

The ability to scale a deployed database can be achieved in two ways, vertically, e.g. by adding RAM, updating CPU and storage; or horizontally, by adding discrete machines to the cluster. The vertical scaling does not ensure HA and serve as a band-aid solution for performance issues. The horizontal scaling is considered a long-term maintainable strategy to facilitate both the data volume growth as the Read-Write scalability.

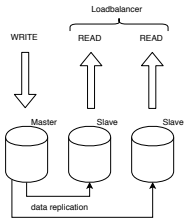

Figure 4.3: BASE clustering?

In the RDBMS systems the horizontal scaling is by design optimised either for Read (R), Write (W) or Read-Write (RW) operations, with applicability depending on the RDBMS system at hand. The following discussion focuses primarily on the MySQL systems, due to author's better understanding and experience,the MySQL active development, systems market share as well the MySQL's use in the legacy application.

READ-optimized scalability
This section focuses on Read-queries scaling which is applicable for systems with Read-Write ratio of at least 60 % [42]. Regardless the ACID or BASE approach, the entire dataset has to be stored on each nodes of the database, allowing the application(s) to connect to nodes arbitrarily. In practice, the connection between nodes and clients is proxied via a load balancer, preventing a single node overutilization.

With ACID, the transactions are performed synchronously ensuring consistency across the cluster at any given time. As indicted by CAP theorem, the ACID approach, e.g. Galera Cluster (GC), trades the availability in for consistency and partition tolerance [43]. The consistency is achieved by simultaneous transactions to all of the database nodes. It should be noted that any delays or queuing in the communication layer, actively lock the entire cluster for both Read and Write operations. The phenomena is mitigated by use of timeouts, forcefully removing slow nodes from the cluster, hence creating a partitions, and by invalidating the pending transactions. The outcome depend on the selected engine and whether the quorum (the minimum number of healthy nodes permitted) is maintained. In practice, the servers are scaled to operate at 60 to 80 % of total capability under the peak load. Depending on the application, the nominal off-peak utilisation is close to 10-30 % mark. The failures rarely occur in off-peak periods; and for a three node cluster equally utilised at 60%, the re-allocation of a single failed node theoretically brings the load of remaining two to close to 90 %. The reconfiguration of the load balancer, recovery of the failed transactions, re-connection of failed client applications will create a queue, increasing the load beyond the original node.

The BASE principle utilises the data replication visually shown in figure 4.3. The cluster contains a single Master node, containing binary log, i.e. a list of pending and past transactions. In accordance to two-(or three-)phase commit, transactions are commit to the binary log, evaluating the validity and followed by the execution or rejection. The binary log is then synced with the slave nodes, performing the transaction independently. Due to networking and log syncing delays, the transactions executed on the master are not applied instantaneously, causing data inconsistencies across the nodes. According to CAP, the system provides Availability-Partition Tolerance by forfeiting consistency. In this approach, write operations ought to be performed on the Master node, and reads from any of the Slaves in a load-balanced architecture. To facilitate HA write operations, or write-optimisation, a multi-master architecture can utilised, for example Master-Master, Master-ring and many others [44]. Albeit providing great flexibility towards multi-site deployment, the approach is vulnerable for data collisions due to auto-increments and primary-key duplication.

Read & Write-optimized scalability
While the Read-optimisation places the complete datasets on multiple nodes, (read-)write optimisation requires more sophisticated techniques. One can distinguish between application layer sharding, i.e. functional scaling, proxy layer sharding, i.e. MySQL cluster and low-level database sharding, i.e. Spider storage engine.

A measure to solve the problem is to diversify the stored data across the database cluster. With data stored in tables in accordance to the normalization principle, tables that are unlikely to require

join operations can be placed on different databases on different nodes. Due to functional breakdown of data, the approach is called functional scaling. Functional scaling pushes the scalability dependencies from the database layer, to the application layer, requiring application to be aware of the cluster configuration. The Functional Scaling is limited by the database schema, for example in Delfi-scenario, only the users, payload and housekeeping data can be split.

Another disadvantage of the functional scaling is the server utilization balancing. The databases residing on the machines should not only be operated as independent as possible, but utilized equally for the efficiency considerations. To resolve this problem, one can opt for data sharding. Sharding splits the dataset into parts, or shards, based on intrinsic property, e.g. reception data, and places the shards on different nodes accordingly. With the complete dataset split in shards across multiple nodes, aggregation queries require node-to-node operations, creating severe bottlenecks in operation. Tasks such as shard recovery, shard incosistencies or even backups, increase the operational complexity.

An industry response to the challenges of the manual sharding (for MySQL/MariaDB) came in two solutions: MySQL Cluster (NDB) and Spider Storage Engine. MySQL Cluster was developed as a response to NoSQL popularity, providing auto-replication key-value (KV) storage across database nodes. To interface with RDBMS storage, KV contains hashes of the table index(es) stored on the node. With inherent dependency on the network latency, NDV is tuned to keep the KV in RAM, pushing the hardware requirements beyond the commodity hardware. In some cases requiring the equal size of RAM as the table size [42]. Due to key-values based indexing the query operations require per-row lookup when boundary of single node is crossed, drastically reducing the performance per network hop (and queuing).

Spider storage engine relies on MySQL native partitioning interface for table splitting and open-source library for managing splits/shards across the nodes. The architecture consists of one single Master, responsible for partitioning and Slaves nodes, containing the partitions. By utilizing the native MySQL functionality, the ACID principle of RDBMS is partially enforced, allowing transactions roll backs. While providing excellent INSERT performance [42], the JOIN operations across the cluster are slow and require out-of-box thinking for optimisation.

RDBMS Conclusion

The strength of the database systems following the relational model approach is often accounted to the ACID, the insurance of transaction and database states. The insurance comes at cost of availability, and BASE concepts are often applied. To ease the operation and maintainability of the system, all nodes of the cluster ought to have the complete datasets, stepping away from the sharding models. This limits the available storage, as well the system WRITE/INSERT performance. It should be stressed that the table alterations lock the database tables, both for read and write.

Observing telemetry processing software use, it became apparent that system ought to be as low maintenance as possible. The processing or visualisation errors are tolerable and can be reversed; the data storage is most definitely not. Loss of data, either historic or at data ingestion is unacceptable. Running a single node database is a high risk, not only in terms of availability, shown to be excellent in practice, but primarily in terms of data corruption. Regular data exports and backups are only a part of story; resolving data corruption or data collisions, recovery, e.g. via binary log, takes hours if not days depending on the data size and administrator experience. It can be argued that problem can be resolved in different way, for example with a secondary node, acting as an active backup system and not used for the processing or data delivery subsystems. Which in authors opinion is computational and storage waste.
Analysis of Delfi-n3Xt system presented in chapter 3 required database export, utilizing the MySQL native backup functionality pushed the server load far beyond the tolerable limits. This lead to the conclusion that in Delfi-n3Xt model, aggregating bulk data for data science tasks via the database cannot be performed on-demand and has to be scheduled, unless server is scalable.
Horizontal scaling of SQL servers is a complicated process, requiring fine tuning of the queries and various data type storage considerations. Any changes to the table definitions, e.g. new columns or data types, are costly. Prototyping with MySQL cluster in Docker showed that table alterations across multiple nodes, while serving clients, i.e. storing and retrieving records, proved to be impossible withing skillset of the author.

4.4.2. NoSQL Data Stores

Due to the inherent uncertainty of the system specifications during the design phase of the future Delfi missions, that expose dependencies towards the frame schema, the contents and the processing techniques, a more flexible data storage system is preferred. As discussed in section 4.4.1, the relational databases require a high degree of rigidity with respect to the frame definitions. Regardless the format or the database schema, the uncertainty manifests is the complications with regards to the database alteration. Albeit a minor inconvenience in the development, the alteration of a live system may become in a time consuming process, possibly requiring redundant layover databases, master-slave architectures, backups or a system downtime.

With the rise of social media platforms in the past decade, a shift in the data structures can be observed. The once static webpages, made way for dynamic webpages, serving varying content asynchronously. The rise of the mixed web-content, and facebook-like social platforms required flexible storage solutions for the mixed content with a high throughput capabilities.

Another significant consequence is the change in the data interaction. The classic relational models, designed in the early '80 are transactional in nature. The use cases, often compared to the banking systems, require frequent and extremely reliable data modifications. One example of the reliability, in form of reading consistency, are the write/read 'locks', disabling database table or row alterations or even read's while data is been updated.

The majority of data stored at the time of writing is immutable. This is partially accounted by the social media and IoT devices, resulting in large amounts of user and machine generated data that is never modified, but overwritten.

The third and the final concern to be discussed is the data throughput. The previously mentioned mixed content contains a growing percentage of the streaming data. For example, video streaming accounted for 74% of data traffic during peak hours in North America in 2016 [45].

The shift in the data patterns unsurprisingly lead to the variety of new data storage and management technologies. Granted, the major share of data traffic on the web originates from a limited set of providers,e.g. Google, Netflix and Amazon, and it can be argued that the tools aim to solve the large storage needs solely apply to the select set of companies. Furthermore, considering the online rankings [46] the aggregate of database system mentioning on the internet, the relational database spans the top 4 of currently utilised storage systems. This leads to the question whether the NoSQL-trend, originated from the social media and IoT specific needs, applies to a university grade telemetry processing system.

The What

The concept of non-relational database storage likely predates the relational data storage models [47]. The "innovative" aspect of NoSQL technologies is not the storage strategy, but in the performance of the tools facilitating this behaviour. The term NoSQL dates back to 1998, introduced by Strozzi [46] for an application capable of storing relational data without support for the SQL querying language. In today's terminology, the Strozzi's applications acts as a wrapper, or an abstraction, for the underlying file system storage, providing an API for CRUD operations. While Strozzi utilised NoSQL terminology to emphasise the lack of SQL querying language, a computationally expensive technology at the time; the term NoSQL used today applies to class of all non-relational databases.

The majority of the popular NoSQL systems like Cassandra, and predecessors like Dynamo and Google Big Table date back to late 2000's. Designed by Google (Big table), Facebook (Cassandra), VMware (Redis), Amazing (DynamoDB), these projects have been released under an open source licence, gaining popularity in early 2010's.

The initial need for these systems can be distilled back to:

- Amount of aggregated data grows exponentially
- Reliable horizontal scaling is required
- Data does not fit into the tables and exponentially increase the database complexity
- SQL querying slow for sparse or loosely defined data
- flexiblity in scaling, e.g. the opposite of RDBMS sharding
- low operational costs, less features, less overhead result in a lower cost of operations

- less requirements on the hardware, i.e. 'commodity'. (no 128GB RAM for <100 GB of data)

The popularity of IoT devices created a new paradigm pushing the previously established needs even further. The generated data is by design unstructured, containing anything from geo-location, images, video streams to a variety of senor data. For example, authors simple home automation system consisting primarily of light switches and heating device controller, generates 2MB/day worth of data. A comparable commercial solution likely generates tenfold of that, aggregating various user data for the commercial needs.

Datamodel Classification

With a large variety of NoSQL systems, one can identify four major data models sharing a distinct feature: lack of any significant requirements on the stored data, i.e. schema-less. In the agile development cycle, NoSQL allows data to be stored in a structured, semi-structure and unstructured manner, and modified at any points, even in the production environment. Furthermore, working with OOP languages, NoSQL allows objects to be stored natively into the database. For Delfi missions, the need for a flexible data model originates from the requirement to support flexible telemetry frame encoding, allowing parameters to be stored without manual adjustment to the databases.

Within the realm of NoSQL systems one can identify four distinct data models:

- Key-Value stores
- Document stores
- Column stores
- Graph-databases

Key-Value store

Key-Value store is a simple storage mechanism that operates on similar principles as the object storage, persisting the serialised data (value) under an unique identifier, i.e. key. The storage mechanism supports Create, Read, Update and Delete (CRUD) operations, but a limited querying functionality. Deviating from RDBMS schema-on-write methodology, Key-Value store assumes schema to be encoded in the data, e.g. XML, JSON, Avro [48] or available the target application logic. Applying the right-tool-for-the-job principle, the KV stores are applicable for situations requiring fast lookups, for example, session information management, e.g.browser cookies. Limited to Key-lookups, store is not suited for querying as application logic would be required to perform the (un)necessary query operations, increasing the complexity while potentially decreasing the efficiently of the system. In case of Delfi projects, the Key-Value data stores are not suited due to lack of query language.

Document store

Document data store is based on Key-Value principles, allowing Key lookups, but with "Value" a queryable document structure. The document, i.e. the data stored, follows predefined format, for example XML, JSON or BSON, containing a series of fields and attributes allowing a hierarchical nesting. Albeit dictating document format and syntax, documents schema is not required, allowing storage of arbitrary fields and parameters. Similar to KV stores, documents are retrievable by key, or via an API queries, with implementation varying capabilities.

Column store

The Column or Wide-column data model leans into direction of RDBMS systems, by using a relational table. While RDBMS systems by design read data per row, the Column store systems retrieve data per column drastically increases the read performance. For example, consider a query to retrieve anomalies on Spacecraft bus voltage:

```
SELECT received_time, bus_voltage
FROM processed.OBC
WHERE bus_voltage < 11 OR bus_voltage > 13
```

Since none of the parameters of the query can be used a secondary index, all records of the table has to be read. Due to MySQL design the complete rows are retrieved, followed by filtering of the required columns, i.e. received_time and bus_voltage, and evaluation of the query statement:

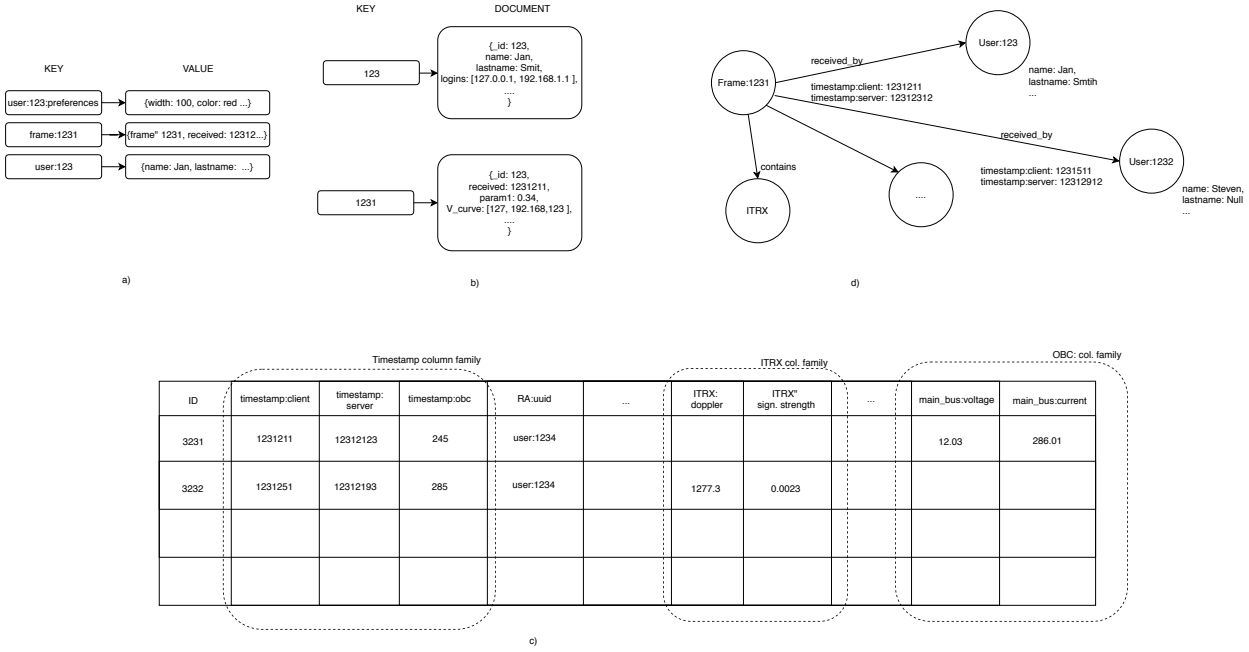

Figure 4.4: NoSQL data store types:
a) Key-Value, b) Document, c) Column-store d) Graph-data store

bus_voltage < 11 OR bus_voltage > 13. With an excess of 60 columns read per table, a substantial performance improvement can be achieved by reading only the required columns. Unlike the RDBMS, KV or document-stores storing the data as a complete record, Column orientated systems store the column data separately. In contrast to RDBMS, splitting based on data relations, the Column based system split columns based on the application use. Furthermore, by column storage, support arbitrary number of columns, partitioned into column families. On low level, instead of partitioning data per row, as done in RDBMS, data is additionally grouped per column family. Furthermore, the Column Data store facilitates flexibility by allowing sparse columns.

Graph-data stores
Graph data store (GDS) provides a schemaless infromation persistence, following graph theory by structuring data in accordance to implicit relations. While the RDBMS relations are explicit in from of foreign keys, and serve as a constraint to to increase the dataset consistency, the (high-order) relations in GDS serve as a source of additional information and can be used to detect hidden relationships between entities. In its core GDS is optimised for queries that are typically complex and slow in RDBMS systems, e.g. multi-level joins, by attempting to describe the real-world relationships instead of information normalisation relationships leading to schema's optimized for the application use-case querying.

The Graph-data store is proven useful for data with high variance in implicit relations, e.g. social networks. In case of Delfi, Graph theory can be applied to the dataset to assist with anomaly detection, e.g. by trying to investigate illogical relations, but is not suited for the entire dataset storage.

Scalability, ACID, BASE and CAP
As discussed in section 4.4.1, the CAP theorem implies that a distributed persistent data storage by default ought to cope with partition tolerance, focusing either on the availability (AP) or consistency (CP) [49]. It is argued that in practice Google Spanner is only one of few systes that achieves Availability and Consistency (AC) [37], however, a closer investigation reveals that Spanner is optimised towards C and forfeits A and the claim of AC [37] originates from user's believe in the availability, implying a low down-time, implicitly ignoring the partitioning. The consistency-partition (CP) and availability-partition (AP) resilience relate back to ACID and BASE principles, requiring system to be either consistent, hence locking [50], or available but eventually consistent. It should be stressed that CAP states the capabilities of the system under normal operations, rather the preference between consistency and

Table 4.1: Replication Policies [52] [51]

	Eager	Lazy
Master-Slave	RDBMS	[CP]: HBase, MongoDB
Multi-Master	Megastore (google)	[AP]: Dynamo, Cassandra

availability under network partitioning [51].

Replication and Sharding Classification

The CAP theorem directly relates to replication, i.e. the data synchronisation across the cluster. Gray et al. [52] classified the replication as Synchronous or Asynchronous, nicknamed Eager and Lazy respectively. The Eager approach commits the changes to all database nodes, prior to returning transaction status to the client, effectively guarantying the consistency across the nodes. The Lazy approach, commits data to the receiving node, passing the changes asynchronously to the remaining replicas. Gessert and Gray [51] [52] furthermore classify replication based on update location, i.e master-slave or multi-master (update anywhere).

Where SQL based systems may rely on shared-disk storage (SAN, NAS) to operate in a distributed manner, the NoSQL systems are generally built upon a shared-nothing [51] architecture with the database nodes connected via network. The scalability and throughput is therefore achieved via partitioning of the data across the nodes.

The data partitioning or sharding depends on data type and use by the application and can be divided into range-sharding, hash-sharding and consistent sharding. The range-sharding is achieved by splitting of key-space in chunks, co-allocated with data nodes, based on range of values. Any incoming data is redirected to appropriate shard or the node via the master node operation [53]. The hash-sharding is applicable for the look-up operations, by coordinating the location of shard based on the primary key and is primarily used in Key-Value stores. The Consistent hashing [54] instead of hashing to database node, hashes key to the partition that is contained within a shard, that can exist on multiple nodes.

Next to the inter-node partitioning, i.e. hashing, and replication, the intra-node partitioning cannot be ignored. Data store systems may operate in-memory, utilising the volatile memory, i.e. RAM, for storage. To ensure the persistency, the data is either distributed across multiple nodes, i.e. VoltDB or persisted to durable memory, i.e. Redis. The use of flash memory storage, i.e. SSD, allows a high performance Random Reads, historically unavailable with use of HDDs. This lead to SSD optimised data stores such as Aerospike [55].

Gessert [51] classifies NoSQL databases further by data access patterns, i.e. update-in-place and append-only. Due to low performance of random writes, the disk-dependent database systems, utilize caching and logging in form of buffer pools. The append-only systems maximize throughput by perform write sequentially, primarily by appending data to a log. To facilitate the performance of random and sequential reads, the log is indexed using various strategies [51]. Gessert argues that the garbage collection, e.g. compaction or removal of deprecated data is computationally expensive.

Query Based Classification

The majority of the RDBMS systems are optimised for normalisation, often scattering the data over multiple tables within a single node or duplicates across a cluster. The process of dissection ought to be performed by the application prior to writing, making it computationally expensive, especially for high-throughput scenarios. The NoSQL systems contain, by design, tend to aggregate the related information in a single collection, increasing the performance of the write operations.

In terms of reading operations, one can distinguish two types of querying: retrieve by index and retrieve by scan. Retrieve by index applies for RDBMS, KV and Document-based data stores, allowing retrieval of entire object based on the object key. These systems tend to encode metadata in the key, as shown in figure 4.4, to ease the lookups. The secondary indexes are available in some of the systems, but maybe inaccurate and slow for cluster-wide operations.

The column based store does not support per-row retrieval as done by RDBMS systems, and requires join operations between column families. In this case, multiple objects are addressed or scanned by the query. With exception of CDS systems, the RDBMS due to normalization, often outperforms NoSQL in terms of random data access. In terms of recurrent application queries, NoSQL systems are often

optimized for the expected application queries and are generally slow for the random querying (with exception of Graph data stores). For example, correlating the satellite data from two collections, user data and satellite data will grossly underperform in majority of NoSQL systems. The available optimisations depend on NoSQL system and can in some cases be achieved via 'view's predefined and constantly updated queries.

Next to the scan and key lookup queries, database can perform analytical queries, either natively, i.e. MapReduce or via external platforms, such as Hadoop, Spark, Flink ...

4.5. Conclusion

As discussed in the previous sections, the RDBMS is rigid towards data ingestion and storage. For purposes of the telemetry data storage, each telemetry field ought to be defined in the database schema prior to be stored. It can be argued that infrequently used fields can be stored as an unstructured JSON object; the approach that is supported by MySQL and Postgres, however, requiring a NoSQL-like schema-on-read behaviour of the application. Furthermore, it should be stressed that unstructured storage is not supported for the querying and does not contribute to robustness of the application.

The vertical scalability of RDBMS is excellent, however, as discussed in the section 4.4.1, scaling horizontally introduces numerous challenges. The database utilisation has to be determined prior to the scaling attempts, which may require partial application redesign to comply with the changes. When stored data exceeds the single machine storage limits (either storage or RAM), sharding has to be applied. Sharding increases the complexity exponentially, introducing secondary challenges such as reduced the JOIN performance due network I/O limitations requiring application data query tuning, complex back-up and schema alteration processes.

The storage should dictate any requirements on the business logic. Therefore, the business logic of the mission, the spacecraft, should not depend or be aware of the ground segment storage. To support this requirement, the storage subsystem should support various types of telemetry definitions, implicitly supporting flexible telemetry frames. The mission can be performed by multiple spacecraft, requiring storage, processing and querying of satellite specific data.

To assist with the data science tasks, the data should be stored in the same database, facilitating the efficient querying. The little big-data approach, exporting the raw data and storing the processed results locally on researchers machines and external drives should be avoided. Next to data loss risks, providing long-term storage and sharing is inefficient, while te re-introduction of the results back to the data storage can be a complex task due to schema rigidity and access rights.

The popularity of NoSQL systems in BigData and Data Science based industries is consequence of flexibility of the storage and ease of scaling. Popularised in late 2010's, Data Lake [56] approach facilitates unstructured storage of all aggregated data as an attempt to monetise data in use of analytics in the future. The concept focuses on the storage, and aims to process only when data is required. In contrast, Data Warehouse, often a SQL-based system, focuses on storage and processing of all available data, increasing the setup efforts as well requiring maintenance. It should be noted that the Data Lake approach often attempts to bridge the classically separated entities, merging the data from various sources, e.g. departments.

Albeit limited with an expiration date set by the data retention policies, the stored data growth is potentially unbound. To decrease the storage costs the commodity hardware is preferred, deviating from RAM memory hungry SQL systems. This tendency naturally lead to horizontal scaling and AP principles observed in the majority of NoSQL systems. Driven by unstructured and high heterogeneous datasets, the schema's are avoided leading towards simpler storage structures. Combined with data-throughput optimisations, the querying is often limited and pushed to the application layer. The tendency is clearly expressed by near-data-processing, execution of data processing scripts close to the storage.

The data storage requirements of the legacy Delfi missions both in data volume, as the ingestion rate are magnitudes lower than any typical data processing system. The study of the aforementioned techniques originates from the flexibility, long term support, unbound storage, ease of maintenance, reliability aspects, classically not achievable for small system due to inherently high development or licensing costs. Considering the legacy application use-cases, the querying is limited to application specific tasks and ad-hoc analytics queries. Both types of queries as well the stored data types are

incompatible with the Key-Value and Graph data stores, due to lack of querying of the former, and lack of data relations of the latter. As a side note, Graph theory may have an application in study of RA use and satellite reception patterns, which is out of scope of the telemetry processing system.
The choice between document-based or wide-column stores, depends on many aspects of the system, but for sake of simplicity reduced to the data types and querying needs. Document based systems are designed for denormalisation of data: the documents should not link to each other. The PouchCouch use case, strongly focused on Document-based store. Primarily driven by the availability of tools supporting the data store in browser and ability to pre-compute the application queries, the Document data store provided sufficient results, as discussed in chapter 8.

4.5.1. Query

In the legacy Delfi systems, the querying is limited to the UI and website tables and graphs. The telemetry data analysis presented in chapter 3 is based on Python data analytics libraries: Pandas, NumPy and SciKit. The locally hosted MySQL database was primarily used for data aggregations from different databases and tables. It should be noted that the use of MySQL did not contribute to the ease of the analytics. For example, to determine the earliest received version of a telemetry-frame requires multiple dataset scans and an additional view:

```
CREATE VIEW temp AS
SELECT incoming.frame, min(incoming.frame_time) AS minTime
FROM incoming.incoming
GROUP BY incoming.frame;

SELECT raw_data.rtc, temp.minTime, incoming.radio_amateur, incoming.receive_time
FROM raw_data.raw_data INNER JOIN incoming.incoming ON raw_data.counter = incoming.counter
INNER JOIN temp ON temp.frame = incoming.frame

DROP VIEW temp;
```

It should be noted that the JOIN is performed on the binary, but String saved, 1793 characters long telemetry frame. This leads to the conclusion that analytic querying will most likely be performed in Python, Excel, Matlab or any other tool, rather than in the database native query language, as users are likely to be more skilled in the latter.

4.5.2. CAP and BASE

The satellite telemetry data is immutable and represent the instantaneous spacecraft state in the legacy missions. The data consistency, expressed as the time required for new data to be propagated through the system, depends on the data delivery rate, network and system latencies. The eventual consistent approach is acceptable for the Delfi-n3Xt and Delfi-C3 operations since sequentially received frames have a high degree of bias. For the DelFFi type of missions, requiring the whole orbit data, a higher degree of Consistency is required, as information is contained within the sequence rather than individual telemetry frame.

Comparing Delfi missions to a typical high-throughput application, the Consistency can be forfeit for the Availability, as <10s delays are found historically to be acceptable. Therefore ACID is not required and BASE is acceptable.

4.5.3. Storage Volume

The use of per-satellite storage system is expensive in term of operations, requiring the legacy systems to be updated to the modern security standards. Maintaining a single system allows the reuse of the existing components as well as the further development and improvement. This, however, requires the system to facilitate the upgrades of the components while in operation. Inherently, the support for an unknown number of spacecraft for an unknown period of time requires a system capable of storing an unbound volume of data.

Gessert et al. [51] argue that Databases can be selected based on Data Volume, Access type and CAP/Consistency, and based on the proposed binary tree lead to the CouchDB or MongoDB for HDD-

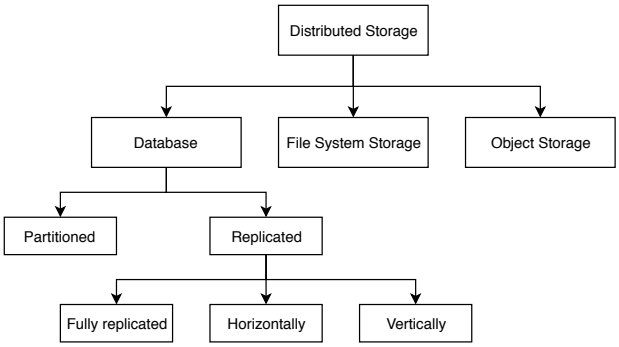

Figure 4.5: Distributed Storage systems

Size datasets and Hadoop/Spark type of systems for the unbound data growth. The applicability is debatable, and depending on the point of view, the decision tree can be seen either as a selection for the data model or as a selection process that ignores the data model altogether. Furthermore, Gessert argues that the data stores with comparable data models have similar properties, which is highly controversial. The work of Siddiqa et al. [50] is an attempt to provide an overview of NoSQL systems, but limited to the proposed use cases and ignores the effects of, e.g. sharding or indexing on the application.

A different approach is proposed by Martin [57] and discussed in section 6.2, stating that database selection should not influence the (software) design. Specifically in this case, this entails the database application should be exchangeable, therefore, selecting a less suitable data store is acceptable.

4.6. Preparation of the next iteration phase

The study of the database systems showed a wide variety of available systems. To comply with flexible telemetry definitions, flexible data stores are preferred. Furthermore, due to unbound data storage requirement, single node systems cannot be considered as the performance of the seek operations degrades with larger data volumes. This decision is additionally supported by the operational redundancy and storage redundancy (backup) requirements.

Figure 4.5 provides an overview of the distributed data storage systems presented in this chapter. The partitioned, but not replicated database system, as discussed in section 4.4, place the tables and databases on nodes separating based on the function and target application. This approach fails requirement GN-DS-1, requiring a replicated data store. The replicated stores can be further classified based on the replication into three major categories. The fully replicated data stores place complete replicas on each node, increasing the storage requirements. The vertical replication is applicable primarily to table based RDBMS and Column-based stores. As introduced in section 4.4.2, the tables are split vertically per column groups that are replicated across the cluster. The horizontal replication splits the data collections into row-groups or shards in RDBMS and NoSQL systems. The performance of the whole data queries is the highest for fully replicated nodes and column-based stores when no column family aggregations are required. The sharding decreases the query performance and requires an alternative for the frequent whole dataset queries.

The need for a distributed storage system leads to two architectural options. First, the data store nodes can be deployed as a server cluster, a de facto approach found in the industry. Another possibility is to deploy the database nodes on the client applications and actively creating a highly distributed cluster spanning over both server and client systems.

The use of client leveraged system provides a unique opportunity to scale elastically with user demand, as each user could be responsible for the data operations it requires. The client leveraged approach is studied further in chapter 8.

5

Telemetry processing

Data processing is the primary goal of the telemetry processing system. The stored data is therefore expected to be error-free, with a marginal tolerable error. Previous work based on the satellite data [9] identified inconsistencies, that can be traced back to human-errors. This chapter focuses on the techniques to reduce the chance of human-errors and mitigate the impact of these occurrences.

5.1. Rationale

As outlined in the chapter 3, the legacy Delfi systems followed distinct architectures and designs. The processing of Delfi-n3Xt telemetry was performed redundantly on both client and the server side, with only the raw, unprocessed data forwarded to the server. In Delfi-C3, only the processed data was delivered to the server application [24]. Although not documented, errors were present in the Racal, the Delfi-C3 client application, leading to the errors in the database. Furthermore, Rascal was purpose designed, with hardcoded telemetry frame definitions in the java source code [24]. Update of the application requires source code recompilation, and re-installation on all of the client machines. Failing to do update all of the client applications, would results in inconsistencies in the ingested data, reducing the degree of certainty of the changed parameters. Logically, *a known error is preferred over the uncertainty of the complete set of data*.

Looking at Delfi-n3Xt, the telemetry data is sent unprocessed to the server. Therefore any client-side processing error will not affect the server stored data. This is important, since a small portion of users, utilized an outdated client application. Unfortunately, a number of errors have been identified in the processing definitions of the server-side application, requiring bulk data reprocessing.

The errors in the legacy systems can be distilled to the following:

Each software component uses own proprietary telemetry processing routines, lacking an unified telemetry schema/encoding definition

The Delfi-n3Xt frame definition contains up to 260 entries, describing the parameters present in the telemetry frame, along with position and decoding rules. While some of the legacy applications did not contain definition files, others require a plain text file or a GUI application. Regardless the implementation, the process is highly prone to human-errors, which inevitably lead to the inconsistencies in the processed data across the entire ecosystem. It should be noted that the problem was recognized by the developers and frame-simulators have been designed to perform tests to validate the decoding part of the system (weak type of unit tests). Unfortunately, this didn't ensure the correctness as errors are still present in both systems.

5.2. Unified Processing

Early in the project, it was determined that to ensure the quality of the designed system, it was vital to be able to provide consistent and reproducible results across the entire platform, both in the client to the server applications. To truly ensure the consistency, all applications should perform the state

change, in this case, data transformation, identically. Ideally, this would require the use of the same state machine: identical processing software in all of the applications.

To comply with the currently utilized systems, without limiting the future applications, the processing software is ought to be language agnostic, and inherently cross-platform compatible. Since no true language independent computation system exists, one can opt for two viable alternatives. The first alternative is a stand-alone processing software, deployed in conjunction with the main application, acting as a processing plugin. The second alternative is an external definition file that compiles into a processing object or class in the native application language. The first alternative, in theory, provides the highest rate of success, by truly offering an unified processing. However, this ultimately leads to the recursion by reintroducing the original problem, as the decoding software will likely be purpose-only designed, re-introducing the original human-errors for telemetry definitions and programming. Opting for the second alternative still leaves the option to deploy the source code in the stand-alone fashion, effectively mimicking the first alternative.

As introduced in the chapter 3, the telemetry processing performs four steps:

1. Binary AX.25 frame transformation
2. Payload extraction from AX.25 frame
3. Separation of the individual parameters based on the telemetry frame definition file
4. Parameter decoding in accordance to the definition file

The first step of the process, the frame transformation, is required since the telemetry frames are least significant bit (LSB) ordered for the purpose of the cyclic redundancy checks [24]. Practically, this means that the binary frame is reordered 8 bit pairwise.

The second step is the extraction of the AX.25 fields, such as satellite identifier, telemetry payload and CRC values. Both CRC and satellite identifier determine whether the telemetry frame can and should be parsed. In the third step, the telemetry payload is parsed based on the definition file that describes the position of parameters within the telemetry frame. As described in section 3 , the parameters are expressed as integers in the binary form, requiring further decoding to the appropriate datatypes.

Globally, three types of processing operations are being performed. First, the complete telemetry frame is modified. Followed by extraction operation, for example, spacecraft identifier or parameters within the payload of the frame. And finally, the parameters are decoded based on conversion rules, for example, the spacecraft parameters.

The definition of the initial frame transformation is immutable and is part of the protocol description and radio operations. It was therefore decided to separate the transformation from the processing and decoding logic as it is not determined by an external definition.

5.3. Processing Frameworks

As discussed in the previous section, the binary telemetry frame is first dissected into parts that are then transformed into metrics via mathematical transformations. To ensure the 'unified processing', both parts should be done in the processing framework.

At the time of writing, two categories of binary dissector were considered. The first type operates in a dissect-only manner with tools focusing on the parsing of the network packets (Pcap) and predefined binary structures. The second category operates in encode-decode fashion, providing an abstraction for data compaction with the goal to reduce the data size for network transfer or storage.

The research showed that the dissect-only category is developed primarily as plug-ins for a single programming language. This is logical since the tools aim to ingest data structures, for example, to analyze network packets or extract data from document files. In case of Python language, Scapy [58], Construct [59] and Hachoir [60] are the most actively developed.

Kaitai Struct [61] is an exception on the rule an provides dissection and decoding capabilities for a wide range of the programming languages, thus supporting 'unified processing.'

The second category, encode-decode, is utilized to encode and decode data structures to optimize the transport or storage, hence linking multiple systems and tools. These tools are, therefore, frequently multi-language compatible, for example, ProtocolBuffer [62] and MsgPack [63]. The encode-decode tools lead to lock-in to their ecosystem, requiring the same libraries for encoding and decoding

of the data as the schema is embedded in the binary structure. While some tools such as Apache Avro [48] circumvent the problem by defining the schema externally, the schema cannot be modified to recognize and decode AX.25 packages. This limits the applicability as telemetry frame parsing.

Therefore, only the Kaitai framework provides the flexibility and language independence required to decode satellite generated binary telemetry data, regardless of the selected format and encoding. It should be noted that Kaitai is explicitly developed for software reverse engineering and decompiling and therefore cannot encode the data to binary structures.

5.4. Kaitai Framework

The Kaitai frameworks consist of three separate parts or projects: domain-specific language, compiler and runtime object. Section 5.4.1 introduces the Kaitai programming language, followed by section 5.4.2 discussing the compilation of the KS definition file into Runtime object discussed in section 5.4.3. Finally, the Kaitai processing is applied to the Delfi-n3Xt example in section 5.6.

5.4.1. Kaitai DSL or Kaitai Struct (KS)

Domain-specific-languages are programming languages designed for one specific domain and in contrast to general purpose languages, such as Python that focus solely on the domain of the application, and are therefore inherently more expressive.
KS Language is a declarative language, designed with sole purpose of parsing the binary structures, such as files, memory, stream and any other generic objects. The Language is, in fact YAML, a common format for configuration files. YAML, or Yet Antoher Markup Language, [64] is often compared to XML, where XML is, in fact, a structured documentation and YAML a human-readable serialization. YAML is therefore closely related to data structures, supporting collections and scalar contents, enabling developers to use language native structure manipulations, without additional object models (DOM).

The data structures, expressed in KS language are stored as KSY files. Any KSY file is built from user-defined types specification, consisting of meta, doc, seq, instances, enums and types tags.

Meta-tag provides information, for example, an id, title, intended file extension, encoding (UTF-8) en endianness and any dependencies of the intended binary structure.

Doc-tag allows human-readable documentation for each of the entities and parameters both within KSY files, as in docstring for the target compiled language, for example in Javadoc or Doxygen for Java.

Seq-tag contains the sequence of attribute specification of the given binary object. Each attribute has a unique id and a type that automatically determines attributes the bit-length within the binary object. Types can be either of a built-in type, for example, u8: an unsigned 8-octets long integer or a custom user-defined type. Each user-defined type, recursively consists of a meta-tag and a sequence definition, allowing reuse for different scenario's. It should be noted that the attributes will be available as attributes of the Kaitai decoded object in the target langue, and are parsed by default.

Instances-tag defines instance objects, that can be determined ad-hoc based on the built-in specifications. For example, contents of an instance may depend on the telemetry frame id or any other set of parameters. Instances are available as instances of the decoded Kaitai object and parsed by request.

Enums-tag provides a mapping between an integer to a symbolic name by key.

5.4.2. Kaitai Compiler

The goal of Kaitai is to describe a particular format in KS language only once and compile it into source files of any of the supported programming languages. The source files include a parser that consumes the described data structures and provides an API to parsed attributes and instances.

Being compiled in the native target languages, the source files and structures utilize language native features and allow debugging through-out entire transformation. Latter proved itself useful as Kaitai 0.8 contained a bug in exponentiation, compiling $1 * 10 \wedge (-7)$ to $1 * 10 \wedge (-7)$ instead of $1 * 10 * *(-7)$ for Python language. This lead to unexpected behaviour as $10 \wedge (-7)$ is evaluated as XOR operation between 10 and -7, resulting in -13, rather than expected 1e-07.

In addition to parsing, the compiler performs additional checks based on defined types and attribute length in the ksy, catching errors in the definition files.

5.4.3. Kaitai Runtime Object

The generated source files, as the name suggests, are C-like structs. The structs are translated into language-native objects, allowing access to the attributes and parameters as defined sequences in KSY file. The Struct object, resulting from Kaitai parsing, is not iterable in Python, requiring manual parsing into the target structure. With a NoSQL as the permanent storage medium, JSON notation was preferred due to its compatibility with document and column store systems. It should be noted that created JSON ultimately determines the schema of the database, and potentially affecting the visualization logic. It was, therefore, determined to keep JSON flat, without any embedded structures.

The objects, containing decoded telemetry data, can be stored as language-native objects, for example, as Pickles in Python, without the need for JSON passing. Furthermore, the objects can be stored in the native format in the database, allowing more fine grade access to the embedded structure, resulting in the performance increase of both parsing and data retrieval.

5.5. Processing Application: Delfi-n3Xt Example

With unparsed and unprocessed raw data readily available, Delfi-n3Xt was the most logical testing ground for Kaitai parsing. With frame protocols explained in chapter 3, this section solely focuses on an overview of the KS implementation and benchmarks in Python version 3.

The Kaitai based decoder for Delfi-n3xt is build up from four files, the main Delfi-n3Xt.ksy file, and three dependencies: obc_shared.ksy, frame1.ksy and frame2.ksy as can be seen in Figure 5.1. For presentation purposes, the magic field as well frame1 and frame2 files have been shortened, with the unmodified files in the appendix of this document.

The main Delfi-n3xt file is responsible for three functions: frame filtering, loading of the dependencies and selecting correct frame type. Although not implemented, cyclic check functionally can be added as required. Conversion of binary and integer values to metrics, is performed in the frame definitions.

Frame filtering is performed by comparing the frame magic, the signature of the binary structure, against AX.25 address field. As stated in the technical documentation [24] allDelfi-n3Xt telemetry frames share the common and identical address field, making it ideal for the frame filtering. It should be stressed that a mismatch in magic is a run-time error, and requires exception handling not provided by the library.

The obc_shared type is defined in the obc_shared file, containing metrics used in both frame types. Note the obc_shared frametype parameter use in the main file to determine the correct frame definition file. The use of True and False boolean objects stems from one bit allocation of the frametype metric, which is converted to boolean implicitly. Allocation of two or more bits, prevents the cast to boolean, allowing use integer values for frame selection.

Some of the parameters, available as struct attributes, require further processing in accordance to the conversion tables. The conversions are built into frame1 and frame2 files as instances. As mentioned in the Struct Language section, the instances and enums, being functions, are not called and therefore computed by default. Calling the instances require exception catching as some of the Delfi-n3Xt frames proved to contain switch positions, for example ADCS and OBC modes not defined in the telemetry definition documents.

To illustrate the use of instances, consider two subsystem metrics embedded in frame2. Since same measurements are taken, the same KS type can be reused:

```
    - id: obc1_dssb
      type: obc_dssb_type
      size: 3
    - id: obc2_dssb
      type: obc_dssb_type
      size: 3
...
 obc_dssb_type:
    seq:
      - id: subsystem_status
        type: b1
        enum: subs_status_enum
      - id: overcurrent_status
```

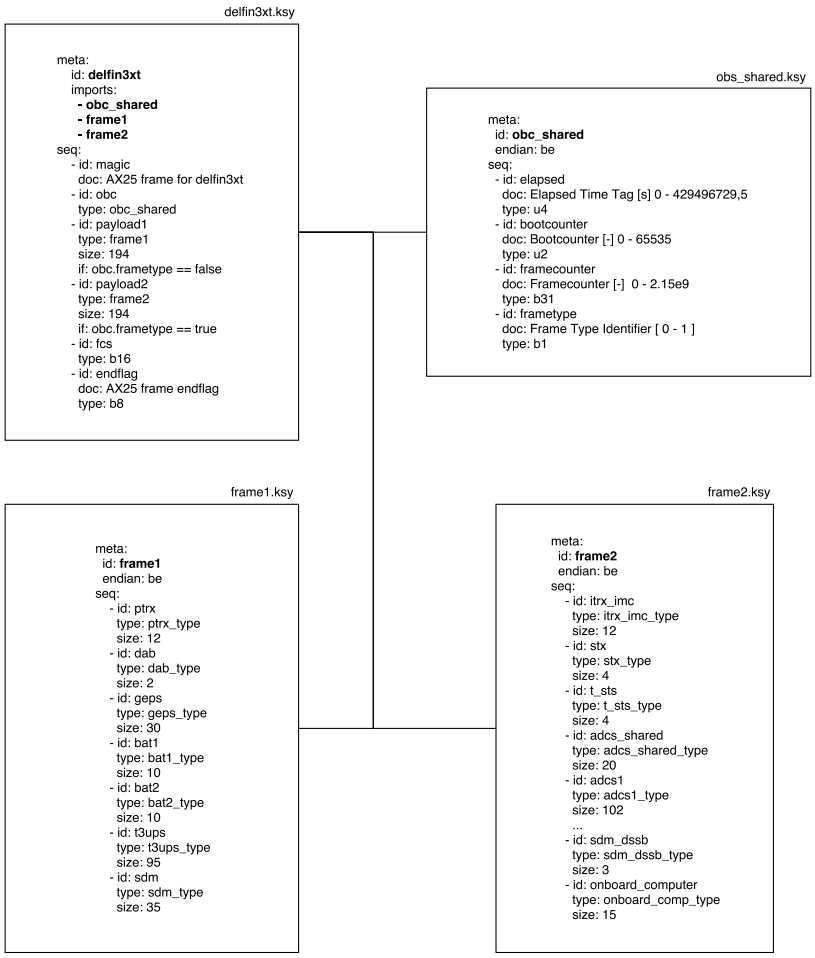

Figure 5.1: Kaitai processing Objects for Delfi-n3Xt

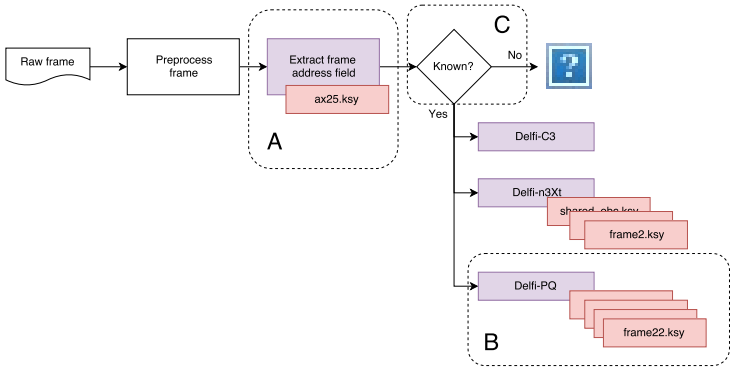

Figure 5.2: Kaitai processing challenge

```
    type: b1
    enum: oc_status
  - id: sys_c
    type: b10
  - id: sys_v
    type: b8
instances:
  subsystem_current:
    doc: Subsystem Current [mA] 0- 130
    value: sys_c * 0.12708
  subsystem_voltage:
    doc: Subsystem Voltage [V] 0 - 14.3
    value: sys_v *  0.05608
```

In the runtime the current of obc1 and obc2 can be accessed as object.payload2.obc1_dssb.subsystem_current and object.payload2.obc2_dssb.subsystem_current respectively.

The processing of the complete Delfi-n3Xt dataset from String objects to python based Kaitai decoded objects on average consumer-grade hardware requires 120 seconds. The parsing of the entire dataset to JSON objects requires 300 seconds.

5.6. Processing Application: generic deployment

Kaitai framework provides a flexible processing and parsing functionality in application native language. The proof-of-concept outlined in the previous section provides satisfactory results both on flexibility of processing as well the performance measurements. However, before deployed in the production, one must establish operational procedures with regards to new telemetry frames definitions as well bug testing.

When designing the application, regardless whether the client or server side, one has to establish the frame identification steps as well the frame update policies. To illustrate the challenge, consider figure 5.2 with three points of interest has been identified as A, B and C.

The figure 5.2 illustrates the data transformation from a raw frame to the processed Kaitai object. The first point of interest denoted by A, is the frame address extraction. With popularity of AX.25 protocol, it is assumed to be utilised in the future telemetry systems. Failure to do so, or deviating in the address field length or starting position will require additional procedures, albeit possible within KS language, preferable performed outside Kaitai programming with multiple address extractor definitions.

The second point of interest, denoted by B, is the selection of KSY decoding object. The two choices are trivial, either performed using KSY or externally via the application logic. It should be noted that the selection pivots on the procedure A, as expected satellite identifiers are required. Opting for the Kaitai approach, actively merges A and B to a single unified KSY file, with imports per satellite. Adding a new satellite, would require recompilation of the unified file, replacing the original and addition of new processing objects for the satellite. The unified KSY file, would contain:

```
meta:
```

```
    id: Delfisats
    imports:
      - delfic3
      - delfin3xt
      - delfipq
seq:
    - id: axstart
      doc: AX25 frame startflag
      type: b8
    - id: identifier
      doc: satellite identifier
      type: u14
      ...
   - id: c3
      type: delfic3
      if: identifier ==  ...
    - id: n3xt
      type: delfin3xt
      size: 194
      if: identifier ==  341953523248182583797378259724902 4
    - id: pq
      type: delfipq
      size: 194
      if: identifier == ...
...
```

Opting for the application approach requires satellite identifier extraction at point A and comparison against known definitions and application of the matching KS object. This approach would require a procedures for the satellite definition updates next to the updates of KS files.

The final and the most important consideration is point C. Adding new frame definitions, not only require recompilation of KSY files to target language libraries, but reloading of the libraries in the runtime. The implication depend on the applications language, for example, python contains imp.reload(module), that can be triggered runtime, without the need to restart the application.

The final consideration is the delivery of the source files. Whether compiled from a remote definition repository, or simply loaded the externally pre-compiled source file, ultimately depends on the degree of trust in the local compilation, e.g. availability of up-to-date compiler and other considerations. It is highly advised to design and implement unit tests to the correctness of definition files as safe guard for Kaitai compilation bugs. Unfortunately, since Kaitai is one way stream, focusing solely on decoding of the data additional encoder or frame generator is required. The frame generator is considered out of the scope of the project.

5.7. Conclusion

The data processing systems used in the legacy systems are purpose designed, lacking any conformity with regards to the telemetry format and processing definition. With telemetry frame counting over 260 fields, requiring a per field transformation definition, the process is prone to human errors as observed in both Delfi-C3 and Delfi-n3Xt missions.

The use of unified processing definition, such as Kaitai or a standard such as XML Telemetric and Command Exchange (XTCE), facilitates the reuse of the frame definition and allows for the definition validation.

Within the project, Kaitai framework was found most suitable due to its native support in the target programming languages, allowing identical telemetry decoding and parsing across any application. The processing performance is excellent, allowing the complete Delfi-n3Xt dataset parsing and decoding within 120 seconds.

6

Software Engineering Tools

The review of the legacy telemetry processing software [13] [11][18], as well the open-source alternatives [65], show a slow change in methodology towards a modular design, both on the software as the hardware levels. Driven by the need to improve the existing system, this chapter attempts to provides an overview of technologies and techniques addressing the limitations of the legacy telemetry systems.

6.1. Separation of Concerns

A classic example of approach focusing on the modularity is the separation of concerns (SOC) introduced by Dijkstra in 1974 [66]. The theory behind the separation of concerns is straight-forward: a software is divided in sections, each addressing one single concern. The ”information” of the concern, i.e. its function, is contained within the section of code and exposed to the rest of the system via an interface. Applying SOC helps to refine the ”what” and ”how” of the system, but does not necessarily enforce the modularity. A more practical application of SOC is defined by Reade [67] in 1989 as a tool to:

1. Describe ’what’ has to be computed
2. Organize the computation sequence in steps
3. Organize memory management for the computation

Today, many of the tasks listed above are handled by the operating system (OS). The core of the SOC concept is however still very much applicable, arguably on a much higher level.

To illustrate the use of SOC, consider an example of a company producing goods with the shelf life depending on the storage facility conditions. Several metrics such as air temperature, humidity, and other external sources like vibrations are monitored by a set of off-the-shelf IOT devices. The system is expected to determine the remaining shelf life of the goods and serves as a pricing tool for the sales department. Additionally, it has been decided that the system should keep track of users and use of the system, to monitor the performance of the sales department.

Using the SOC method, the following simplified concerns can be derived:

1. receive the sensor data
2. store the sensor data
3. retrieve the historical data
4. perform the data processing and execute prediction model
5. retrieve the prediction for given product
6. serve the results (web)
7. retrieve user data (client)
8. store the (clients) data

The concerns 3 and 5, as well 2 and 8, preform same functions and are therefore called ”cross-concerns” since the functionality has been used by different part of the software. With no obligations opposed on the system deployment or programming language, the programmer decided to implement the system as a single Python application. Due to missing long-term data retention requirements, with the sensor data is stored temporarily, allowing the programmer to use local filesystem and in-memory objects and a simple flask library for publishing data to a webpage.

Assume that at a later point data retention was enforced. The local filesystem storage is becoming the major bottleneck, with growing log file slowing down the journaling and ultimatly OS. Migration to a database system involves changes in the concerns managing the data storage. In this case, affecting only the two nearest components, and not the business logic: prediction and pricing models.

In the near future, the system is required to offer services to the external clients: for example, affiliate sales partners. A complete overhaul of the website related concerns is requried to accommodate for security, authentication and user access rights requirements. The developer could, for example, select popular Django framework instead of Flask, to provide the authentication and to comply with MVC pattern. From the high-level overview, the consequents of the re-utilization of the existing business logic in ”model”, the existing website design in the ”view” and new authentication logic in the ”controller” of the MVC pattern.

The example may appear as a very straightforward solution, which it is, however, from the developer point of view, the boundaries often get blurry, and SOC principles can be easily ignored in favour of faster results. A data access object (DAO) is generally used to provide an abstraction for data retrieval and storage logic. The developer may, for example, ”optimize,” skipping the DAO altogether and executing SQL queries directly in-line, for example in the view of the MVC pattern. This would save in the development time of DAO object, the amount of generated code, and may even seem an appropriate solution.

By taking a step back and looking at the larger picture, it is apparent that the behaviour actively mixes the business logic: computations, with storage logic: the data retrieval, and data delivery logic: the web services. The mutual dependencies cause changes that affect the others components, for example, a change in the storage logic affects both the business and data delivery logic directly. Furthermore, any database-level runtime errors will affect the web-services directly.

A failure to separate the concerns inevitably leads to the subsystems opposing unnecessary requirements towards each other. As shown in the DAO-less example, a failure to define the abstractions scatters the data access logic throughout the application, reducing the code readability. It can be aruged that a hard to comprehend feature leads to the programming errors and bugs. It should be noted that testing of the interdependent elements require more complex testing procedures since the unit tests cannot be performed reliably on interconnected elements.

Theoretically, if applied correctly, the SOC promotes the modularity and low dependency between the components, enabling reuse and smooth exchange of the system components. In reality, the situation is complicated, with dependency creep present even in a perfectly defined SOC. Let’s assume that in the previous example, the business logic was developed in Python version 2.7, the default OS X version at the time of writing. With the bugs in business logic have the highest impact on the system, the returned value is the highest if the mission-critical, highly tested component is reused. This leads to the decision between migration to Python 3, required for latest version of Django, or use of Python 2 with older LTS version of Django that will be maintained until 2020. Assuming that system still fullfill the needs in 2020, may lead to use of oudated webserver version, increases the risk of vulnerabilities. For example, a 2016 study [68] identified 7% of all web publicly accessed servers vulnerable due to legacy version, and 29% due to lack of system updates. In 2018 the figures changed to 15% and 18% respectively [69], with 20% of all discovered vulnerabilities to be of the High and Critical categories.

The example is meant to show that solely relying on SOC is not enough to ensure reusability and long-term robustness. Following the SOC, guidelines assists with the (functional) separation of the business (supporting) logic, without the tools for mitigating the interdependencies. Without a long-term development strategy, the initial single monolith is getting replaced by mini monoliths.

6.2. Clean Architecture

The separation of concerns is a useful technique to separate the application logic, however, deceptive for inter-dependencies. In the recent years, a number of new techniques have been proposed, e.g.

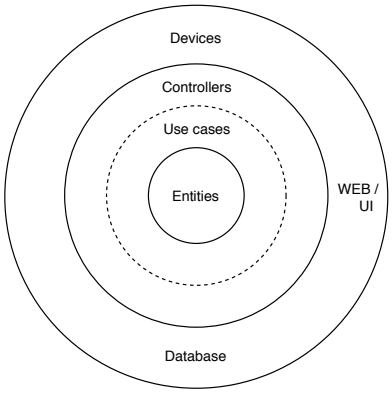

Figure 6.1: Clean Architecture

Hexagonal Architecture [70] , Onion Architecture [71] and many others. In its core, the solutions tend to apply SOC by structuring application into layers, e.g. data logic, business logic and presentation layers. The differences between approaches is beyond the scope of the discussion, but is bundled by Martin in 2012 [72] as Clean Architecture and is based on the established approaches as a continuation of layered software scheme, SOC.

The choice to investigate clean architecture is not accidental and coincides with need for low coupling, something that approach is well accredited for [73][74][75].

When applied to software development, the Clean Architecture focuses on the following aspects [57]:

- Independence of frameworks
- Independence of User Interface
- Independence of Database
- Independence of Suppliers
- Testability

The clean architecture attempts to structure the application by providing a visual representation of separation of concerns. As shown in the figure 6.1, the application is split into layers, i.e. concentric circles, in accordance to the function, from high level functions internally to low-level functions externally, e.g. Entities, (Use Cases), Controllers and External Interfaces. Clean architecture dictates that the dependencies may only point inwards, therefore, a software object declared in the outside the layer, may not be referenced by any of the inner layers. For example, a database may not be directly used by the Entities, the business logic of the application. To solve the connectivity problem, the business logic is addressed by the Controllers, who convert the data formats from external agencies, e.g. HTTP requests, Databases, to Entities format. It should be noted that although modifying the database data, the layer itself is not aware of any specifics of the database. The most outer layer contains frameworks, data storage and provides glue-code to the Controller layer [76].

The principle explores and defined intricate control flow of the application, which is irrelevant for the high level architecture design.

6.3. Conclusion

The take away of the Clean Architecture 6.2 and SOC 6.1 study is that regardless the system view, the architecture ought to be divided in modules in accordance to application logic (SOC function6.1). The high level modules should not depend on the low level modules (Clean Architecture 6.2). By implicitly following SOLID principle, Clean architecture require abstractions to be independent of details, e.g. the Controller layer is independent of the selected database or web framework.

The review of Delfi-C3 and Delfi-n3Xt datasets revealed inconsistencies, related to the processing errors and discrepancies between client and server generated results. In accordance to SOC, the processing out to be abstracted from application logic, which can be labelled as Business Logic. To ensure

the consistency across the system the suggestion to utilise language agnostic processing subsystem (see section 5.4), aligns with immutable Entity, the inner circle the Clean Architecture.

The inherent ambiguity of usage and mission duration, leads to the realm of the future proofing. Gorman argues that future proofing of the system, requiring software changes without software change, is impossible [77]. The notion of unpredictability of the future dictates that the system requirements may change at any time, which lead to two alternative options. Either the code is untouchable and changes are re-configurable externally, or the changed functionality is removed and replaced by the new code. Regardless the selected solution, this means:

- Logic that changes frequently, has to be placed outside the code.
- Any module that frequently changes, has to be decoupled from the system

Reviewing the usability experience of Delfi-C3 and Delfi-n3Xt system, a conscious decision has been made to separate the system into decoupled blocks. Inspired by Actor Model architecture and Akka framework [78], the decomposition towards task specific tools rather than purpose coded functionality was preferred. The rationale of the decision is inherent lack of documentation of the in-house developed application, leading to notion of redesigning rather than re-use. Utilising purpose-designed tools, ensure component understanding, and with Clean Architecture decomposition, easier change, satisfying the future proof requirements.

II

Application

7

Architectures

Following the iterative methodology presented in chapter 2 and the background and, the problem study summarized in chapters 3 to 4 lead to the decision moment of the system architecture.

First, the role of the client application has to be determined.

Figure 7.1 illustrates the possible split in Client to Server functionality with the use of four elements: Data Management, Data Logic, Business Logic and Presentation. The Data Management element contains the data storage system and as discussed in chapter 4 may require additional logic to operate. Data Logic encapsulates the rules that apply to data but are not part of the business logic of the application. For example the normalization and partitioning of data, load balancing, READ-WRITE redirections. The business logic in the scope of the project contains telemetry processing logic and data analysis. The Presentation element is responsible for data visualization to the end users, for example as a graph or in a tabular form. The Presentation layer is highly abstracted and could consist of multiple parts, for example, presentation logic (e.g. website) and presentation support (e.g web browser).

As shown in figure 7.1 ten distinct cases are defined, denoted by letter A to J, each further described in table 7.1. Areas of interest are indicated in yellow, based on the applicability characteristic. Options A and B are not feasible due to limited functionality, lacking networking. Options F and G were discarded due to full control of the storage on the client application. Option J requires the use of a peer-to-peer network, completely cutting the control over the system and stored data.

Option C closely relates to the legacy Delfi-n3Xt server system (without DuDe client) relying on static PHP web pages. Option D provides functionality on the client side allowing for data analysis (filtering, S/C parameter selection, queries) on the client side. Option E relates to the Delfi-C3 system that relied on client application for data processing with the exception of Data Logic component. This is required to authenticate users and manage the access for RAs. Furthermore, the component is ultimately responsible for data schema's and data storage structure (e.g. tables). Options F and G are both inapplicable due to loss of control on Data Logic component to the client. This substantially increases the security risks as data access is managed by the client application. Furthermore, the use of out-of-date software has a higher potential for data corruption and data loss as database schema could be altered between software releases.

Requirements GN-UI-GEN-01 indicate that each user type has to have a distinct user interface. This is required since each user type have different permissions for the stored historical data. For example, RA ought to have access to own received historic data but not the complete dataset. To increase the security of the system, the control of the UI and data access should reside on the server side of the application.

Options H and I are based on the distributed data storage. The client applications share the entire data set with the server application, either as a replica database or as a shard. In theory, by allocation processing tasks to the client application, the load on the server hardware is greatly reduced, allowing much leaner operations.

The difference between options H and I is the presence of Data Logic component. This means that client application in option I is capable of redirecting writes, perform load balancing and operate in the

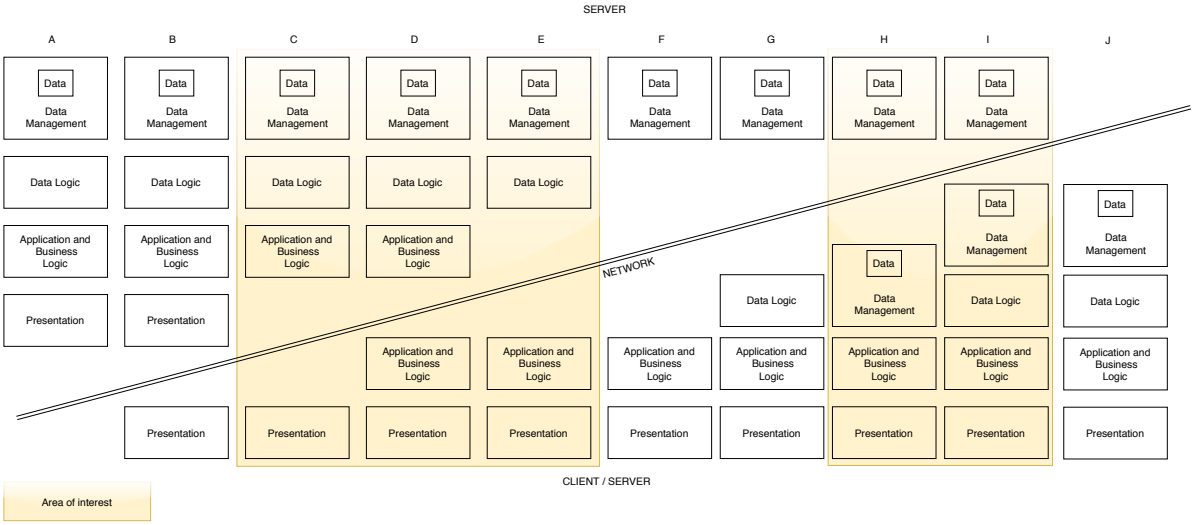

Figure 7.1: Split Client to Server functionality

Peer-to-Peer configuration and more. In this configuration, the boundary between Server and Client applications become obsolete, as identical client applications may be executed on server hardware to perform data processing and analytics, as well on distributed clients such RAs and S/C operators.

Option J pushes the limits even further and operates distributed and decentralized fashion. This is inadvisable for Delfi missions due to lack of central storage system, thus actively losing control of the mission data.

Two approaches were selected for further investigation, I and D. Approach I showed the highest novelty and potential for scientific research. The approach is further studied and discussed in chapter 8.

Approach D provides most features to the client while limiting the access to data. The approach was initially selected as a backup alternative, but due to security concerns of I, was chosen in the end. Approach D is further studied and discussed in chapter 9.

Different deployment methods of both architectures is shown in figure 7.2.

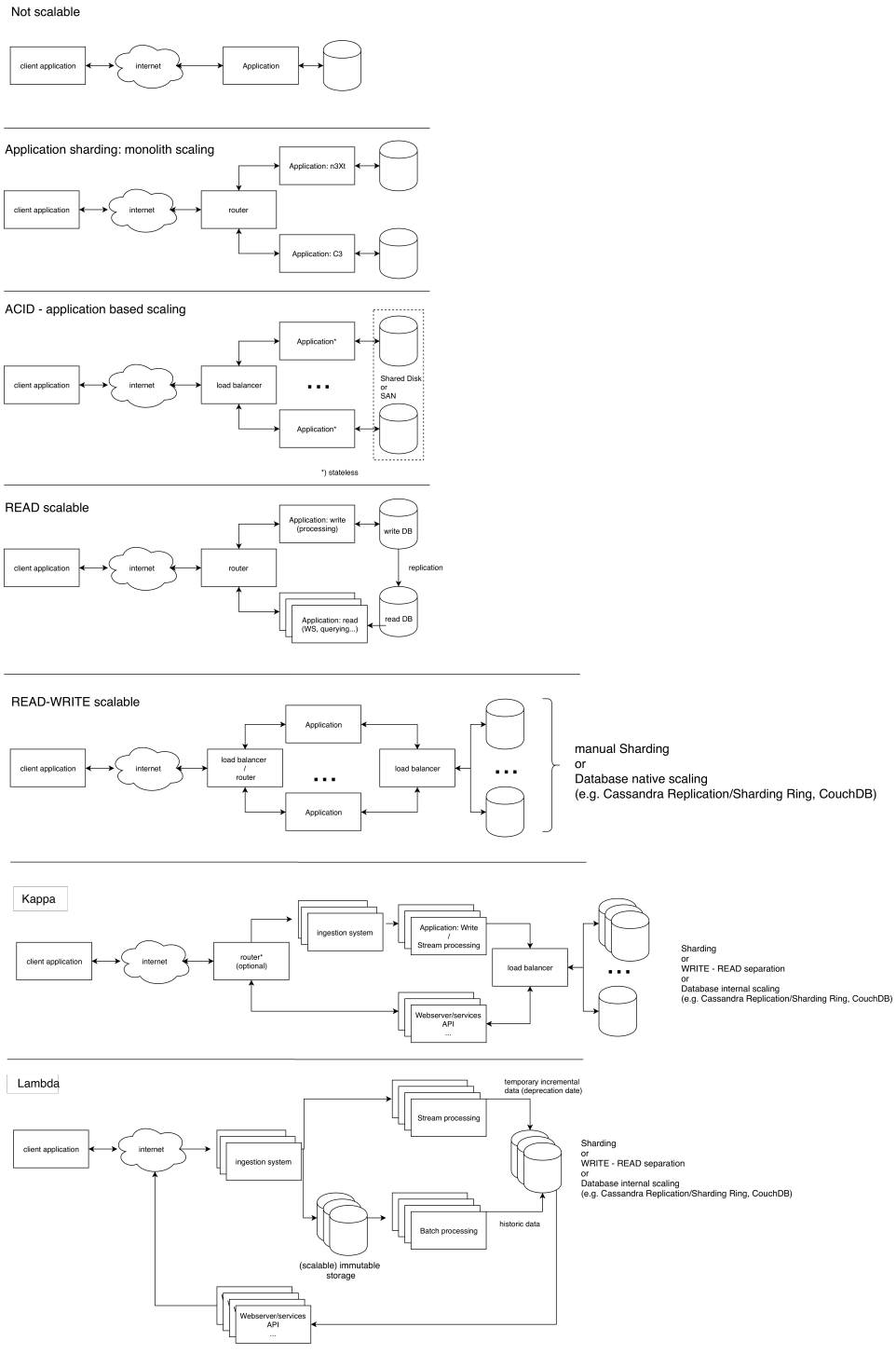

Figure 7.2: Reference Architectures

Table 7.1: Client Server separation

Letter	Description	Example	Applicability
A	No client application, the entire application is executed on the server. (Data Logic is optional)	DuDe client in offline mode	not applicable: RA's are distributed, networking is required
B	Presentation logic is generated on Server and ported to Client. (Data Logic is optional)	Remote SSH session with the server	not applicable: Requirement GN-UI-GEN-03, data filtering and ad-hoc data quering is required
C	Presentation is done entirely on the Client. (Data Logic is optional)	Client: web browser and website. Server: web server	applicable, similar to Delfi-n3Xt server
D	The application logic consists on both Client and Server side. (Data Logic is optional)	Dynamic websites, AJAX, WebSockets, WebHooks. Facebook Graph, Marktplaats, Funda APIs..	applicable
E	The application logic consists solely on the Client side. The server is responsible for data distribution, authentication, permission control etc.	Analytics application running on a Cloud deployed database (Database as a service DaaS).	Applicable: depends on the network performance
F	Similar to E, but without any additional Data Logic.	DuDe client and Delfi-n3Xt system without the web interface	not applicable: control over DB in hands of potentiall untrusted client application
G	Similar to E and F, but with Data Logic on Client Application.	Data aggregation system. Data analytics systems.	not applicable: control over DB in hands of potentiall untrusted client application
H	The Data Management System is present on both Client and Server, but storage system is bound by CAP. The Data Logic on Server side is optional.	Off-site backup systems. Multi-site datacenters.	not applicable: Without data control logic all client applications have the same level of access. This is against Delfi user group policies with regards to RA.
I	Similar to G, but with Data Logic on Client side, possible Peer to Peer (P2P), multi database systems (client)	-	Possibly applicable: PouuchCouch
J	Pure P2P applications	BitTorrent, BlockChain	not applicable: Loss of control over data.

8

Client Leveraged System

The analysis of the collected telemetry data from the Delfi-n3Xt mission reveals a substantial contribution of Radio Amateurs in the telemetry reception. With the Dude and Rascal client applications processing the telemetry frames on the RA's premises, a concept has been investigated to allocate the entire processing to the client, reducing the computational requirements of the central processing server.

As stated by the requirement GN-DS-6, the data growth is potentially unbound, when multiple spacecraft mission are supported. With large datasets, more processing power is required, which is readily available on the client application of the system. Furthermore, by integrating the remote client applications into the storage system, a higher degree of the redundancy and robustness can be achieved. The concept researches the ability to form a cluster with (globally) distributed client applications, rather than centralized servers. The expected observation is that an change in the data consumption or generation, caused by a change in the number of client applicaton, will organically change cluster size, hence facilitating the scalability.

8.1. Architecture

The Delfi-n3Xt architecture, illustrated in the figure 8.1, consists of three parts: client hardware, client application and server application. The client hardware, i.e. the computer's sound card, is responsible for the sampling of the analog signal to the digital waveform. The waveform processing consists of signal demodulation and AX-25 flags detection, and is performed within the client application.
The figure 8.1, albeit a simplified overview of the system logic, shows duplication of the frame parsing, decoding, filtering as well the data visualization logic. This is required, since the data, i.e. the telemetry frames, are delivered and stored in the raw binary-like form on the application server.
As an attempt to streamline the processing, one can determine that the redundant data processing should be eliminated. Globally, this leads to two contrasting possibilities outlined by the figure 8.2.

The first option relies on the client application for telemetry processing, therefore only the 'clean'

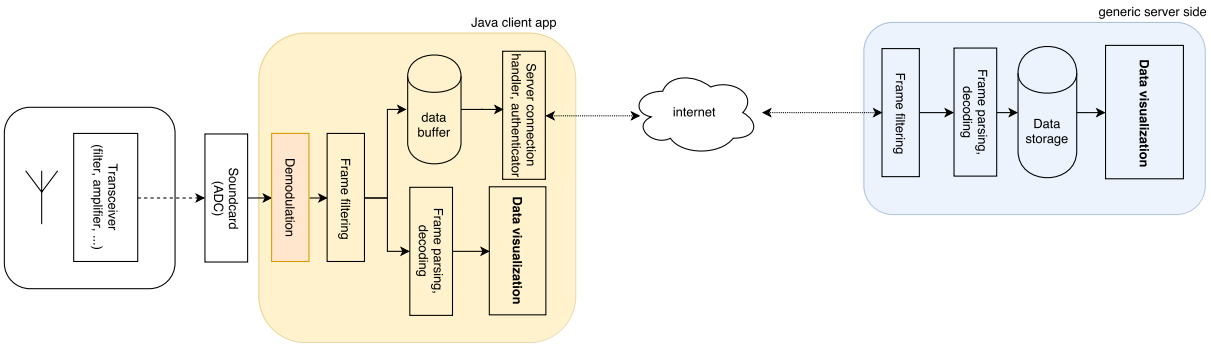

Figure 8.1: Simplified Delfi-n3Xt data processing logic

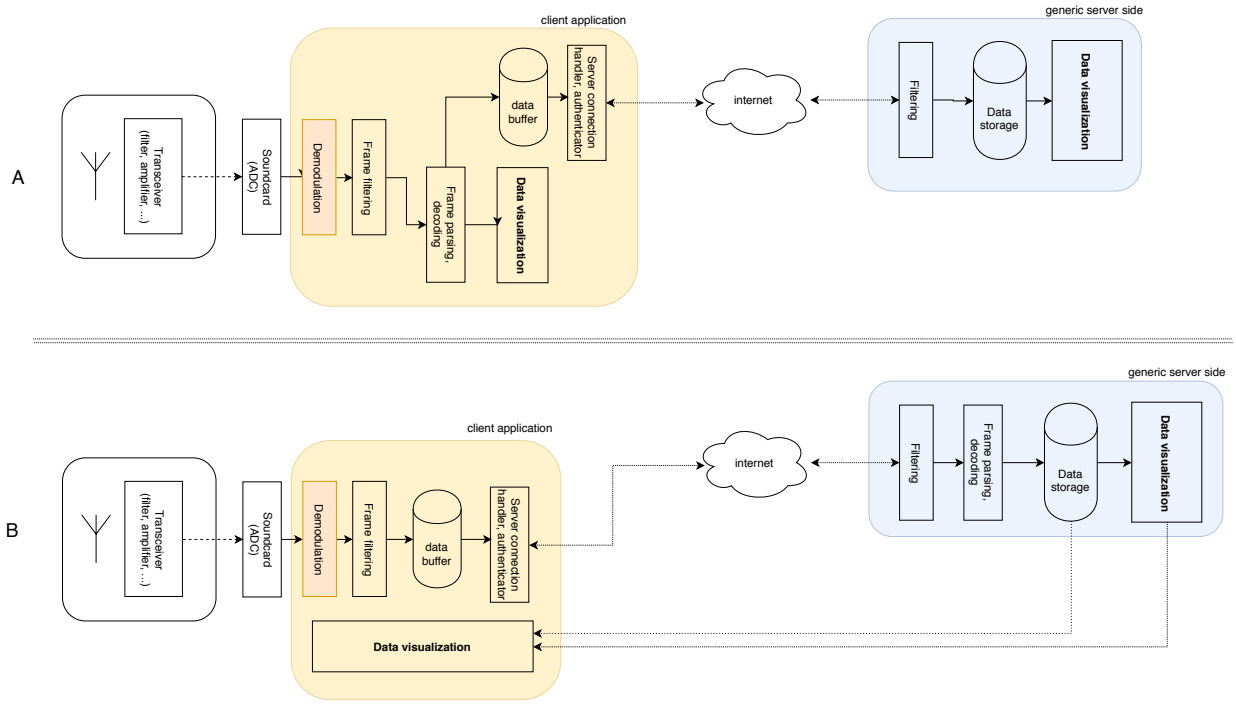

Figure 8.2: Alternative architecture options

and processed data is collected on the server. The concept is great in theory, but has a practical limitation: human-errors [79]. Any bugs in the data processing, e.g. errors in the frame definition, invalidate all of the server stored data. To resolve the processing errors reliably, the original data frames has to be reconstructed and decoded again. Furthermore, with the application being installed on the clients machines increases the risk of being out of date, and therefore using incorrect processing code. Therefore, the identical telemetry frame processed by the two client applications will not necessarily yield the same results. The uncertainty of the results and limited data reconstructions possibilities makes the concept unacceptable.

The second option allocates the processing entirely to the server application, theoretically, ensuring the up-to-date and error-free processing code. For the client-side data visualisation, one can opt to utilize the client application or use other means to deliver the processed data, e.g. the browser.

According to a Delfi-n3Xt operator testimony, the DuDe client application was utilised by the satellite operators during a satellite pass. Independent of the server's status, client application provides instantaneous access to the locally demodulated telemetry data.
In the proposed concept, any downtime or connectivity issues with the server are highly disruptive actively cutting the information feed. The server-side data processing, theoretically, ensures update-to-date telemetry definitions and bug-free processing, therefore, yielding results with a higher degree of certainty.

The proposed optimisation, as seen in the figure 8.3, is an aggregate of both options of the figure 8.1. As an extension of the original Delfi-n3Xt and Delfi-C3 architectures, the architecture require both the raw as the processed data to be transferred to the server.

Outsourcing the processing to the client applications reduces the server capabilities, allowing a leaner operation. With a persistent raw data storage, any client application processing errors are recovered, ensuring the 'eventual' data integrity.

As illustrated by the figure 8.4, the radio amateurs do not update application timely, therefore, to mitigate any computing inconsistencies, the client software requires auto-update functionality, without user intervention.

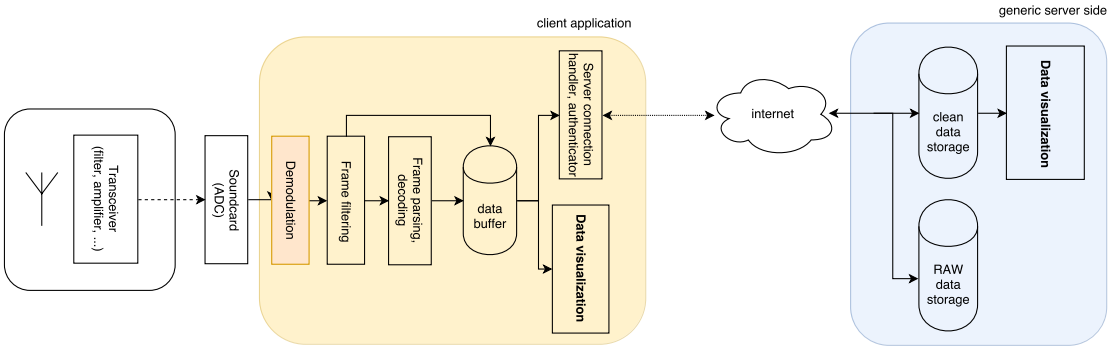

Figure 8.3: Proposed new architecture

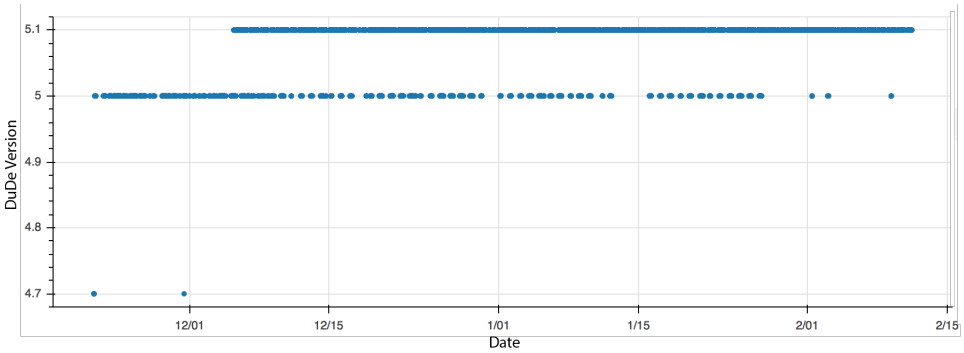

Figure 8.4: Client application version change in time

8.2. Architecture: Tools

In the recent years, many of the classically desktop deployed applications made its way into the browser world. Modern browsers provide a rich programming and visualisation environment consistent over wide spectrum of the client hardware, without the need for applications to be installed on the users' machine. Similar to Java virtual machine (JVM) running the code, browsers provide sandbox model, promising consistent user experience with a strong emphasis on the visuals and interaction. Furthermore, with increased performance and interest in JavaScript programming language, many desktop applications, as well server systems, have been (re-)designed in JavaScript, leading to increased reliability and tools availability.

8.2.1. Browser Application: feasibility

This section studies the feasibility of a browser-based telemetry processing system, following the architecture illustrated by the figure 8.3. The section address the major functionality elements found in the figure, following the data flow, from left to right in the figure.

The first, and arguably the primary function of the system is the data ingestion. Following the DuDe and Rascal methodology, the application should interface with the on-premise radio equipment via sound card ingesting continuous stream of data into the application.

The access to the sound card is provided under HTML5 WebAudio API [80] implemented in the majority of the browsers [81] [82]. The technology is proven in continuous operation by de Boer with WebSDR project [83], delivering Software Defined Radio (SDR) to the web.

The second step is the radio signal demodulation. Porting java based demodulation to JavaScript was out of scope of the project. Therefore, the audio processing capabilities have been investigated based on the available open-source projects. Three tools, perfoming comparamble tasks have been identified: AX.25 demodulator [84], Audio encode/decode suite [85] and a HTML5 based modem [86]. The available projects provide show a sound proof-of-concept (PoC) for browser-based demodulation, however a further research is required to establish the performance and long term feasibility.

The third step is the raw telemetry frame processing: parsing, decoding and filtering of the telemetry frames. The tasks are proven feasible, and explained in more detail in section 5.4.

The fourth step is the temporary storage of the raw and processed data. Historically, browsers offered limited storage support, leaving developers with two options: JavaScript objects, and session storage. Former offered greater flexibility at cost of persistence, while latter allowed limited persistence at cost of the flexibility. The initially defined storage limitations are still present today, limiting the session storage to 4096 bytes. To address the persistent storage limitations, the 2009 HTML5 proposal [87] [88] included a standardisation for local data storage API.
The browser storage API is provided to the client application in form an abstraction, a JavaScript object exposing identical functions for both storage mechanisms. The sole difference is the expiration date of the data, either session bound, deleting data after closing the tab or unlimited, in case of the persistent storage [89].
At time of writing the persistent storage quota's depend on the browser and operating system, requiring user approval upon exceeding OS and browser set thresholds. A quota API [90] has been drafted to ensure browser and OS-wide consistencies, but not universally adopted at the time of writing. In addition to storage volumes quota's, the browser implementations follow own internal guidelines. Mozilla Firefox [91], for example, limits the persistent storage to 10 MB after the browser crash, even if triggered by another application or website tab. Furthermore, the persistence is not guaranteed by design, as users are allowed to manually delete the web storage [91] to reclaim hard disk space.
The local storage provides a simple key-value functionality, wiht support for String objects. Storing data in this serialised form is not ideat, what lead to the implementation of alternative storage mechanisms allowing structured data storage and querying. For example, WebSQL offered a RDBMS database and SQL-querying capabilities for web applications [92]. Implemented only in WebKit and Chrome browser families, the WebSQL was silently discontinued and removed from HTML5 specification in favour of IndexedDB [93] [94]. IndexedDB is a NoSQL database implementation, operating in key-value (object) matter, with support for object schemas and schema-based data querying [95].

Author's previous development work with browser-based applications [96] showes HTML5 inconsistencies across browsers and browser versions, which lead to a strong preference towards an additional abstraction layer for the data storage. The decision is based on the following observations:

- Browsers are updated frequently and automatically
- Browser updates may change or deprecate API functionality and affect the application in unexpected ways
- The performance of the storage mechanisms varies between the browsers
- Schema update are required
- The data stored by the browser is unencrypted, accessible and modifiable by the client

The mission critical software ought to be operational thought the duration of the mission. As demonstrated by Delfi-C3 mission, running for over ten years, the supporting software is expected to be functional and maintainable for the same time span. By relying on an actively maintained 3rd-party (storage) abstraction, the majority of the listed issues can be addressed with limited development effort on the long run.

The last part of the client side system is the communication between the server and the client application. The functionality can be split into two parts: related to data delivery and user-specific tasks. The user-specific functionality, such as authentication, the account creation, passwords reset and many other aspects of user management is a standard functionality for today's websites. The data delivery of the raw and processed telemetry frames from client to the server application can be done trivially via HTTP methods with API or WebSockets.

Based on the aforementioned findings, the browser-based client application is determined to be feasible.

8.2.2. Client Applications: considerations

With the browser-based application proved feasible, further research was conducted to identify tool to satisfy the architecture proposed in the figure 8.3.

The list of requirements can be found in Appendix 2.4, with the minimal list system ought to perform:

- Processing of the locally received radio transmission

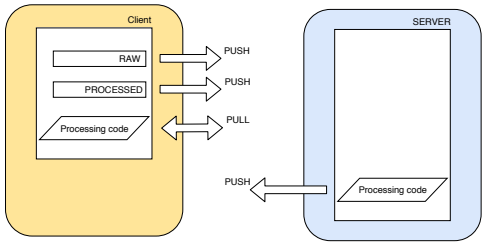

Figure 8.5: Client-server connectivity: original

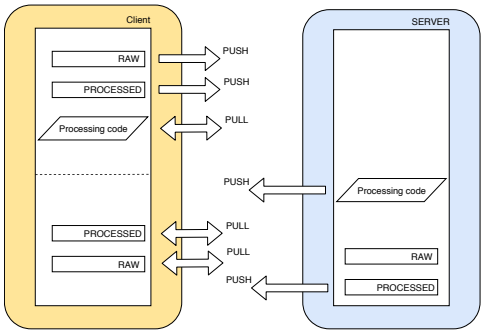

Figure 8.6: Client-server connectivity: with OPS

- Visualizion of the received telemetry data
- Auto-update of the processing code base
- Ability to operate temporarily stand-alone, without server connection

The requirement to (temporarily) operate independently of the application server, requires a stand-alone application which, classically, required a plug-in or chrome app, hence a browser-dependent application. The rise of mobile device use, whose internet connectivity greatly affected by the spatial effects, e.g. tunnels, gave rise to a new methodology: 'offline first' [97]. Driven to minimise impact of the network on UX, the offline-first aims to provide a set of features available even when connection with the server cannot be established.

The functional connectivity between client and server application is abstracted to PUSH and PULL requests in the figure 8.5. The raw and processed frames are PUSHed to the server, while new telemetry processing code is PULLed by the application. Additionally, the new processing code can be pushed by the server to the clients.
The main consideration of the Offline-First system is the assurance of the processing software-state. For example, while offline, the server may have issued (PUSHed) new processing code, rendering the offline decoded data incorrect. Solved simply by providing processing code version, the problem can be mitigated.

Security
The client browser storage as well the processing script are open and accessible to the users, which is considered to be a security threat that should be addressed.

The encryption of the local and IndexedDB storage is supported by numerous JavaScript based libraries. The encryption algorithms, with exception of Base64, require a key to encode and decode the data. The key-less Base64 cipher cannot be considered as the encryption model, but rather a data serialisation method for String based storage models.
It should be noted that not only the storage, but also the javascript runtime is accessbile and can be debugged and modified trivially. To ensure the integrity, the decoding key should cannot be stored in the application and should be retrieved from the server, breaking the offline-first methodology.
As an attempt to resolve the problem, the encryption key can be obfuscated, made unreadable for humans, but accessible for the machine code. The approach is well known in the software industry [98] as a (weak) measure against reverse-engineering of the source code. Most notably, JavaScript library JSFuck [34] [99] provides means of obfuscation by converting Javascript code into set of strings of 6 characters: [,], (,), !, +. The security through obscurity approach is a weak band-aid solution that, in the author's opinion, cannot be applied 'by design' to the production environment.

The statistics can provide an alternative for the encryption model. With only authenticated users having access to the data, the user's pool is known at any given moment in time. Based on the assumption that only a small percentage of users are interested and tempted to modify the code or the processed frames, the data integrity can be validated by comparing the results processed by different sources. Sending all of the raw data to all of the clients for reprocessing is unacceptable, due to network and computational waste on both server as the client application,so only a few active clients

should be selected. It should be stressed that the concept works if the received raw telemetry frame is not modified. Which is problematic with offline-first methodology. Additionally, the approach requires at least three active users at any point in time.

Operations Role
With the client web-based application capable to query and visualise the data lead to the question whether the system can, and should, be applied to the Spacecraft operations. To support the spacecraft operations, the designed client application ought to distinguish between Radio Amateurs, having access to the locally received data, and Spacecraft Operators, potentially having access to all of the historic data. The Spacecraft operators dataset ought to be updated in real-time, requiring real-time sync with the processed data.
The proposed system requires a change to the previously established client-server connectivity as shown in the figure 8.6. The proposed approach to filter out the processing inconsistencies by re-processing the raw frames in parallel on the otherwise un-utilized client applications, goes in hand with the required interface for spacecraft operations: data push capabilities from the server.

Next to the continuous synchronisation of data stores, Spacecraft operations require data querying. Opposed to the simple data retrieval by key, querying allows selection of data based on time domain and filtering based on a set of logic operations. The application executes similar, if not identical, queries therefore a cache mechanism with a set of pre-computed metrics is required [GN-DQ-2].

Implementing the entire application from scratch, providing support for the majority of browsers is a challenge. JavaScript is executed as a single thread, albeit sand boxed, shares the resources with all other browser tabs; changing the browser tab or bringing the browser window out of focus halts the JavaScript execution. Furthermore, to ensure the smooth execution, functions are preferred to be executed asynchronously, either in form of callback or a promise.
The performance is achieved by a fine balancing of the tasks, e.g. downloading large number of small files will massively slow down any browser rendering functions; the decoding functions should not interfere with the for telemetry frames scanning of the radio's bitstream.

With the main focus on the data processing, figure 8.7 shows the sequence diagram of the proposed application. Although not explicitly indicated by the figure, the binary stream operations, such as frame detection, ought to be run as web workers [100], facilitating (virtual) multithreading approach.
The application shows an asynchronous behaviour with regards to storage and processing, but synchronicity towards processing. This is required to ensure the latest processing scripts, arguably, solvable by running additional webworker.

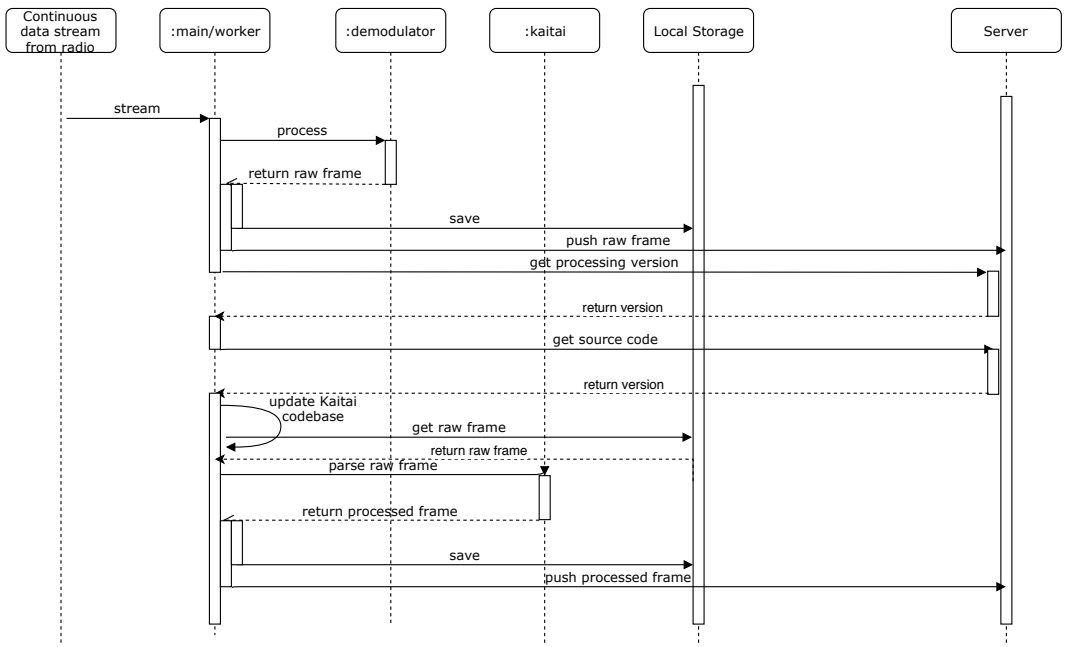

Figure 8.7: Simplified sequence diagram

Requirements
For the storage abstraction, the system ought to:

1. Operate in-browser
2. Store structured and unstructured data
3. Provide ability to query the stored data on attributes
4. Ability to cache/pre-query results
5. Ability to execute filter/scan queries on the data
6. Data should be available, following offline-first methodology
7. System provide ability to distinguish users groups and shield data access off accordingly
8. System should be able to update the processing and visualisation source code

With the given set of requirements, only a small selection of systems, available at time of writing, satisfy the selection criteria. The available solutions roughly fall into three categories, storage systems that provide no native synchronisation options, systems that synchronise manually, or systems that synchronise automatically. The non-synchronised options are often seen in the performance optimized systems sych as LokiJS and AlaSQL, utilising in-memory storage and query engines. Due to increased RAM footprint, these system opose limitations on volume of stored data and may disallow persistence. The storage system allowing persistence are often combined with external data store syncronisation. Due to distributed nature and inherent unknown networking latency, the system cannot achieve Consistency-Partition tolerance, without introducing unbound locks across all client applications. Alternatively, opting for Availability-Partition tolerance, bound by the networking, only lazy Slave-replication is achievable. The eventual consistency may lead to high inconsistencies in data set across client applications. Due to append-only nature of Delfi data sets, the chance of data collision is low, but should be addressed. Unsurprisingly, the majority of storage systems avoid active synchronisation, exposing an API to perform CRUD operations on the server, or requesting dataset (snapshots) from the central data store (YDN-DB, Minimongo). In contrast, PouchDB allows lazy replication, both from server to client and client to server, hence allowing a Multi-Master Two-tear deployment.
Two-tear systems implies two types of storage systems, permanent and temporary. The permanent systems are permanently connected to the network and contain the complete data sets, while the temporary systems may lose connection and/or the entire data sets. The PouchDB library provides the tools to operate the temporary node, while the permanent node is left to purpose built CouchDB data store.
The PouchDB storage system was selected due to the hands-off data synchronisation with the server, providing an abstraction for the PUSH and PULL requests shown in the figure 8.7, while the Multi-Master deployment allows (eventually) consistent operation due to implicit data-collision strategies.

8.3. Implementation

Based on the architecture proposed in the sections 8.1 and 8.2, this section focuses on the high level system design formulation. As introduced in the previous section, PouchDB allows native synchronisation with remote database, introducing the client PouchDB instances into the distributed storage cluster. Bound by the networking delays and geographical distances, the system operates in eventually consistent, i.e. lazy replication approach. The deployment view of the architecture, illustrated by the figure 8.10, shows two distinct types of nodes: permanent on-premise and provisional off-premise infrastructure. Under normal operations, the on-premise nodes are permanently available and connected to the network, while the off-premise devices can connect and disconnect at any moment in time.
CouchDB, a Document-based data store, is utilised for the permanent on-premise data storage. With options to be deployed both in stand-alone as in a decentralised Multi-Master (MM) manner, system is easily scalable, preventing single point of failure. CouchDB is discussed in section 8.3.1.
The PouchDB is an in-browser implementation of CouchDB, providing access to set of the CouchDB features. Both PouchDB as CouchDB define client permissions moderating the data access as well CRUD operations. CouchDB natively supports a per-user database, allowing a personal data store of each individual user or client application. PouchDB is discussed in section 8.3.2.

8.3.1. CouchDB

The CouchDB is a 'web-inspired' document-based data store designed to operate in a stand-alone or in a distributed deployment [101]. The stored information, i.e. the documents are represented by discrete objects consisting of fields and attachments. Operating in append-only mode, the documents are sequentially written to disk-stored database file. The Update of CRUD, is achieved by adding new versions of the document, and changing the pointer to the newest release. The process is handled by B+Tree based engine, indexing the documents based on the unique identifier and the sequence number. The CouchDB is subject to the ACID properties [102], but deviates from RDMBS READ locks by redirecting the write-concurrent reads to the previous version of the document. This is achieved by use of Mutliversion Concurrency Control that facilitates both the Consistency, by preventing the partially written documents from been read, as well the Isolation making document Read and Writes independent of each other. The Durability is satisfied by append only database write, that cannot be corrupted or lost by the application during a malfunction. Any document submit to the data store is either written or rejected, handled by the extendable Update Validation functions, ensuring the Atomicity property.

Being a distributed system, CouchDB is bound by CAP, actively forfeiting consistency for the availability. In this case, the document updates propagate throughout the database cluster, eventually converging to a consistent state, hence nicknamed "eventually consistent". CouchDB supports replication, allowing data sets or replicas, to be present on multiple database instances while ensuring their synchronisation. The documents in the replicas can be edited on any of the nodes, leading to possible conflicts. It should be noted that all conflicting documents are stored in the database, and left out of the Views as long the conflict is not resolved. Due to eventual consistency, the conflicts are not avoidable and treated as a normal occurrence in CouchDB system.

The querying of a collection containing both unstructured and semi-structured documents, each with different set of attributes, deviates from RDMBS SQL-languages. The denormalized documents are self-contained and cannot be queried in table-column-field manner. Due to its distributed nature, query parallelization is important and combined with implicit per-document schema requires queries to operate on per-document level. The aggregation queries combine document extracted information into a View. The Views act similarily to the map operation and apply a javascript function sequentially to the stored objects. The View results are stored in as an additional B-tree [103] structured file, containing the references to the documents by a map (hash) function. On the initial creation, MapReduce transverse the entire data set, while the later changes solely updates the references triggered by the database sequence ID change. Next to JavaScript based View map functionality, CouchDB facilitates Mango Queries, a Cloudant developed API [104] for complete data set traversal.

As illustrated in the figure 8.8, CouchDB clustering can be setup as Multi-Master (MM) or Master-Slave (MS) with filtered replication. The Master-Slave architecture can consist of a combination of one Write and two Read database nodes each containing the entire dataset. The deployment model allows all databases nodes to operate independently, acting as a single stand-alone node, reducing the complexity of the replication system. This design allows database nodes to be added and removed without the need for resharding. In term of performance, the view queries traversing entire datasets, are run independently and synchronously on a single node and are therefore bound by the node performance. The load balancer regulates the node utilisation on redirects Write and Read requests to the designated nodes.

The Multi-Master deployment is achieved using two-way replication between all of the database nodes. In contrast to MS architecture, the dataset is split between nodes in form of replicated shards, and typically three replicas per shard are used [105]. Where in MS deployment, the (CRUD) operations are redirected by the load balancer based on the operation type, with a single node responsible for the entire transaction, the MM deployment utilizes the entire cluster. The CRUD operations are handled by the coordinator, assigned by the load balancer based on the node's utilization, and in the contrast to MS clusters, any node can be used. The process of data Read and Write in a MM cluster follows the following principle [106]:

1. Client Request is registered at the load balancer
2. Loadbalancer redirects the request to a node: coordinator
3. Using the shard map, coordinator redirects the request to the nodes containing the affected shards

4. Coordinator awaits for two responses for Write and one response for Read requests
5. Coordinator processes the responses and passes the results back to the requestee via the load-balancer

In contrast to MS single node operations, the MM approach, due to its distributed nature, performs View mapping paralellized over all nodes containing the given shard increasing the querying performance of the system.

The scalability of CouchDB database deployed in Multi-Master cluster depends on the number of shards. The product of the number of shards and replicas count determines the maximum number of the allowed nodes, hence limiting the cluster size. The sharding number is set prior to the cluster deployment and is immutable. Change of shards number require redefinition of the existing shards and reallocation of the existing data, unfeasible in production. This leads to the concept of oversharding, storage of high number of shards on small number of nodes, to facilitate for future growth.

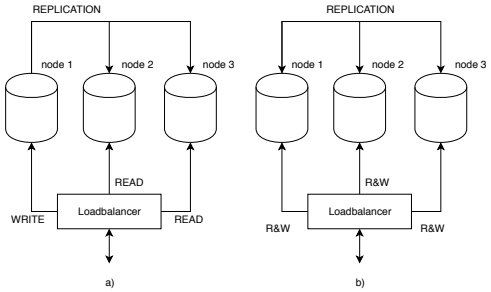

Figure 8.8: Master-Slave a) and Multi-Master b) CouchDB clustering

8.3.2. PouchDB

The PouchDB is an in-browser data store, emulating CouchDB database via the REST API interface. The PouchDB provides an abstraction layer for browser (or NodeJS data stores) and by default utilise webSQL, InnoDB or LevelDB (NodeJS) storage mechanisms. Furthermore, built in a modular manner, PouchDB allows use of custom adapters providing access to in-memory (RAM), local storage and other mechanisms.

PouchDB provides an unified storage and querying interface, independent of the operating system, browser or local installation (NodeJS). This increases the usability and reduces the impact of the browser changes on the application, by abstracting the changes within the PouchDB layer. For example, the FruitDown adapter was implemented in for PouchDB to work around Safari IndexDB bugs [107].

8.4. Distributed, Centralized System

By operating in a distributed manner, the CouchPouch system introduces new opportunities related to the data processing, potentially allowing recovery of otherwise lost frames due to bit-flip, not recoverable in the legacy application.

8.4.1. Document Design

Both the CouchDB and PouchDB are document-based data stores. From development point of view, the documents are abstracted to set of JSON objects, containing an unique identifier, document version number and a set of fields representing the stored data. Due to distributed nature of the data store, the documents ought to be self-contained, encapsulating all necessary information for the manipulations. This section focuses on the contents of the documents, focusing on the effects of schema-on-read system on the application implementation.

Stricture

The document-based databases a schema-less, allowing storage of arbitrary fields in the JSON based documents. The schema-on-read strategy provides flexibility on data insert operations, but requires

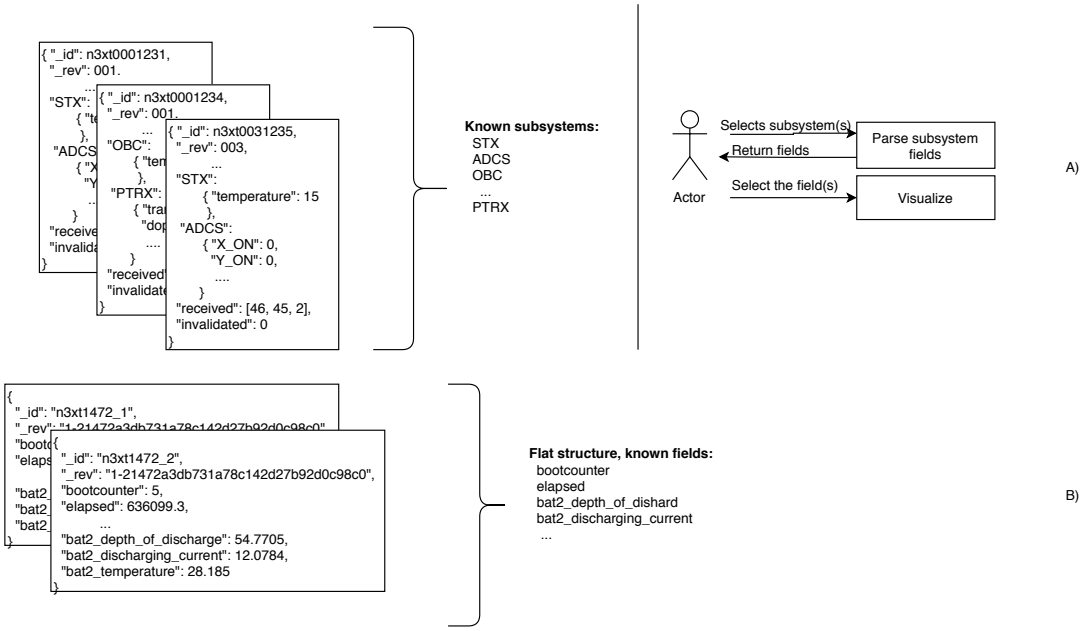

Figure 8.9: JSON encoding A) structured B) flat

the application handling the data to be aware of the documents' structure. This requires a level of sanitation to ensure the system compatibility. The schema synchronisation can be achieved in a variety of ways, and in authors opinion should be directly related to the processing scripts. With the processing code generated from Kaitai definition file that readily contains the available field structure along with per field documentation. The proof of concept structured the stored JSON document in accordance to the hierarchical structure of the KSY file. By storing the KSY file version, any document schema changes are traceable and can be decoded by the application trivially. It should be noted that the hierarchical structure can be denormalised trivially, as long the field keys are unique and descriptive.
While the automated data handling requires precise field name mapping, the data visualisation portion of the system can leverage the user, as the keys can be easily extracted from the local collection of documents. This allows user to dynamically filter the objects and data, as illustrated by option A in the figure 8.9. The choice between a hierarchically subsystem structured document data and a flat document structure ultimately depends on the system processing needs. For example, an aggregation of spacecraft temperatures requires traversal of subsystems fields, while in flat structure it can be performed as a single operation. Furthermore, use of external tools may oppose requirements on the storage, for example Kibana requires a flat document structure.

The design of the document structure is flexible and can be changed ad-hoc, but requires clear definitions to ensure long-term compatibility. Once written, the document fields containing satellite metrics should not be changed, as the immutability of these parameters contributes to the overall system consistency.
Where in RDMBS schema changes are handled by the database system, the CouchDB requires manual document-per-document edits. The impact of human-introduced errors is limited as B+tree provides version control, allowing roll back of transactions and restore of previous document versions. It should be noted that the garbage collection, either manually initiated or triggered by storage bloating, automatically repacks the stored documents, discarding all out-of-date document versions.

The design of the document should be performed at the final design stage of the system, when the processing as well the visualisation requirements are well known and defined. The proof of concept showed that both hierarchical as the flat document structure oppose no significant differences on the decoding and storage system. It should be noted that the document schema should follow KSY schema, to limit the amount of glue code required for the parsing. However, it should be stressed that the KSY encoding should not dictate the document encoding, but rather follow the telemetry frame structure set by the frame design.

Document identifier
To facilitate the data look-ups and increase the performance of the range selection queries, the documents id ought to be set either to the telemetry frame timestamp or the unique frame identifier, set by the OBC. The choice for timestamp may lead to use of time series databases, while frame id's are more suitable for the document-based stores. The use of timestamps is affected by inherent on-board clock drift. This can either be ignored, or adjusted prior to storage. Adjustment may cause a single telemetry frame to be stored under two id's or two different frames to be stored with the same id causing a complex data collision mode. Next, the document id should contain the satellite identifier, to increase the robustness of the application. Therefore, the unique document id should be set to the satellite name followed by the telemetry frame counter, in form of "<satellite identifier>:<frame identifier>" or "<mission identifier>:<satellite identifier>:<frame identifier>".

8.5. The Software Design

The software design introduced in the section, follows the architecture proposed in section 8.2 and software selection process discussed in section 8.3.

The designed system is characterised by the offline-first principle, requiring local storage of the software routines and the data on the client, i.e. the browser. The concept pivots on the PouchDB system, providing distinct purpose-driven databases for JavaScript scripts, as well for raw and processed data. As illustrated by the physical system overview 8.10, up to six distinct use cases can be identified: Ground Station, Radio Amateurs, Satellite Operators, Server-side Database System, Dedicated Processing System and Off-site backup; and grouped in three distinct categories: Radio Amateurs, Satellite Operators and Server-side functionality. The role of an authenticated user database defines the category and implicit sets the access rights to the CouchDB databases, controlling the Server-to-Client and Client-to-Server replication. The local databases, operated by the PouchDB and the browser, by design provide R&W privileges to the client, requiring the replication control to be handled server-side.

Based on the two-way replication, and depending on the level of trust with respect to the users, two distinct methodologies have been proposed: Compartmentalised (per-user) databases and Shared databases. The Compartmentalised and Shared Server-side database systems are illustrated in figure 8.11 by D) and E) sub figures.

The shared database approach limits the users' data access by filtering the replication of the central store to the local PouchDB instances [108]. For the Delfi-satellite operations two data stores are required: RAW and Processed, containing the unmodified received telemetry frames and the processed parsed data respectively. With the document-based store, the use of JOIN operations in unadvised, hence combining the RAW and Processed databases would require merging of the RAW and Processed data, increasing the number of the document edits and information to be transferred to the client. The filtered replication [108] is partially implemented by the CouchDB (hooks) and can be extended to accommodate for the user role-filtering needs.

In terms of security, filtered replication is a weak measurement, allowing adversaries to flood the database with fake telemetry data. With the "merge" data collision strategy, the fake data may dominate the correct reception, leading to potential data loss. As a measure to reduce the impact of single-user actions, the databases can be compartmentalised via CouchDB built-in database per-user functionality, allowing an individual RAW (and processing) database for each of the authenticated users. Furthermore, by setting the storage quota's, the flooding attempts will solely affect the users' (application) local data and not the server storage.

As illustrated by figure D), compartmentalisation requires data aggregation to combine the documents from different receivers. The aggregation step can be merged with the filtering step, allowing bit-flip recovery in conjunction. In operation, the Compartmentalised approach prevents data collision as duplicate frames will be stored on the individual user databases.
The use of compartmentalisation increases the storage requirements, as duplicate telemetry frames are expected to be presented in the multiple personal databases. Following the Delfi-n3Xt trend, the expected duplication is expect to be withing 40 % mark. With impact on the RAM use, a further research is required to determine the impact of per-use databases on the server performance.
Based on the increased storage requirements, as well limitation with regards to distributed processing, the compartmentalisation approach was discarded.

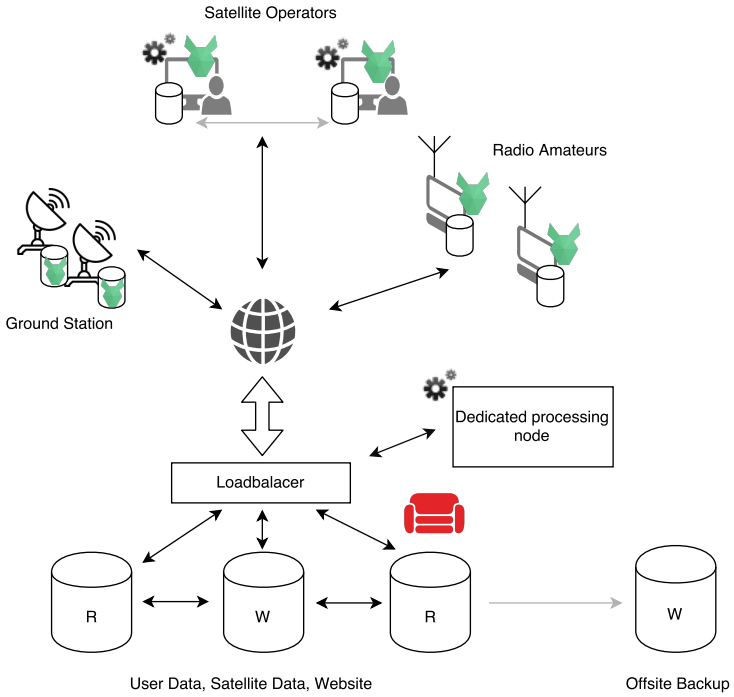

Figure 8.10: Physical system view

8.5.1. User Based Description

Following the user groups and accompanying requirements, this section researches whether the requirements are met per user group and use cases.

Radio Amateurs

Dictated by the requirement GN-UI-GEN-02, the application ought to visualise the received telemetry frames for each individual Radio Amateur. While the Dude and Rascal applications perform visualisation in parallel with the telemetry reception, hence showing only the last received frame, the proposed architecture allow an overview of all client's individually received data. Furthermore, with use of shared RAW and/or Processed databases, the newly received data can be replicated to the clients, providing a live satellite status link. Although not required, the approach may serve as an extra incentive to keep PouchDB website/tab active, therefore increasing the resources for the processing scripts as well number of active nodes in the system.

Regardless the approach, the RA_DB database, containing the demodulation scripts, is ought to be replicate as shown in the figure A 8.11. Depending on the Trust metric, the data processing scripts can be allocated to the user, along with Processed data Write access.

Satellite Operators

Executed with Satellite Operator user privileges, the application allows access to the historic satellite data via replications to the local database. Initially, a subset containing short-term historic data is replicated from Processed database, which can be expanded via on-demand data retrieval. This operates on the assumption that only short term data is required, and operations are visualisation-centred.
To deviate from Excel-based tooling, the front-end application allows local querying, based on PouchDB provided MapReduce and Mango query APIs. At time of writing the required operators queries are unknown, due to its inherent mission specific nature. The proof of concept, limited to selection and on-demand plotting of spacecraft parameters, provided no significant information on excess computational power. This lead to uncertainty with regards to the impact of replication and on-demand queries, that lead to the separation of (kaitai based) processing from operator nodes in the design.

Non-human users

The primary goal of the server-side application is the support of the client applications and permanent

storage/retrieval of the telemetry and user data. Due to the distributed architecture, the PouchDB or CouchDB based nodes can be added at will, for example, to assist with computations with use of a dedicated processing node (figure C).
The server side functionality is (or can be) split across multiple nodes, relaying on CouchDB for the separation. The proof of concept is based on non-sharded, one-way replication between server-side based databases. It should be stressed that the sharding is required for the scalability, specifically for the scaling of the write operations. At time of writing and POC implementation, the system use and the load were unknown, leading the selection process for clustering to later stage when system requirements are better understood.

8.5.2. System Components

This section provides a brief high level overview of the system components.

Databases

The system is based on CouchDB and PouchDB synchronisation, requiring a number of task specific databases.
System Database
In order to provide the offline-first functioanlity, the majority of the code as well the application website(s) should be stored locally in the clients browser. To limit the browser dependency as well as a measure to future-proof the application, the client application code ought to be stored in PouchDB. The POC showed ability to load and run Database stored scripts, in form of document attachments as well with PouchDB extension PouchDB-load.
The replication of the system database allows ad-hoc code adjustment on the server and quasi-instantaneous code updates across the system. Additionally, with per-document set privileges custom scripts and views can be developed for each user group.
On the high level, three types of scripts should be stored: the website related code, the demodulation and the processing scripts. The website code is user-group specific and instantiates the PouchDB databases, as well the the demodulation and processing scripts. The demodulation script is responsible for RF and low-level frame manipulations and is considered to be out of scope of the project. The processing scripts, generated by Kaitai compiler ingest the binary encoded telemetry frame generated by the demodulation code, and return a JavaScript object that can be stored (semi-) natively in the PouchDB database.
Data storage databases
Regardless the selected design, two telemetry databases can be identified: RAW and Processed. The RAW database contain the documents with received binary telemetry frame, indexed by the frame embedded counter and satellite identifier. The processed data, consists of documents containing the processed frame, along with receiver metadata and validation fields.
It can be argued that the separation of RAW and Processing data is not required, as raw and processed documents can be merged to a single entity and therefore be stored in a single database. This is unadvised due to two considerations. First, the merge action introduces additional document revisions, bloating the storage. Second, by merging the documents, the immutable RAW data is modified, which is unadvised due to possible human errors.
In order to support multiple spacecraft missions, the duplication of RAW and Processed databases per spacecraft might be considered. This is in author's opinion not necessary, since the frames are uniquely indexed by satellite name and carry the necessary meta data to determine frame origin to perform per satellite queries. Additionally, the use of high number of databases (500>) is unadvised, as replication load increases non-linearly.
User database
The user database contains private user information required to operate the replication filters and over system authentication. The latter is ignored from the discussion, as design was halted on the proof of concept phase.

Distributed Processing

The distributed system is designed with two considerations. First and the main consideration is to ensure system robustness. With the use of the client-side storage, i.e. allowing client applications to form a distributed cluster, any server failure is recoverable, as data is stored on the remote nodes and

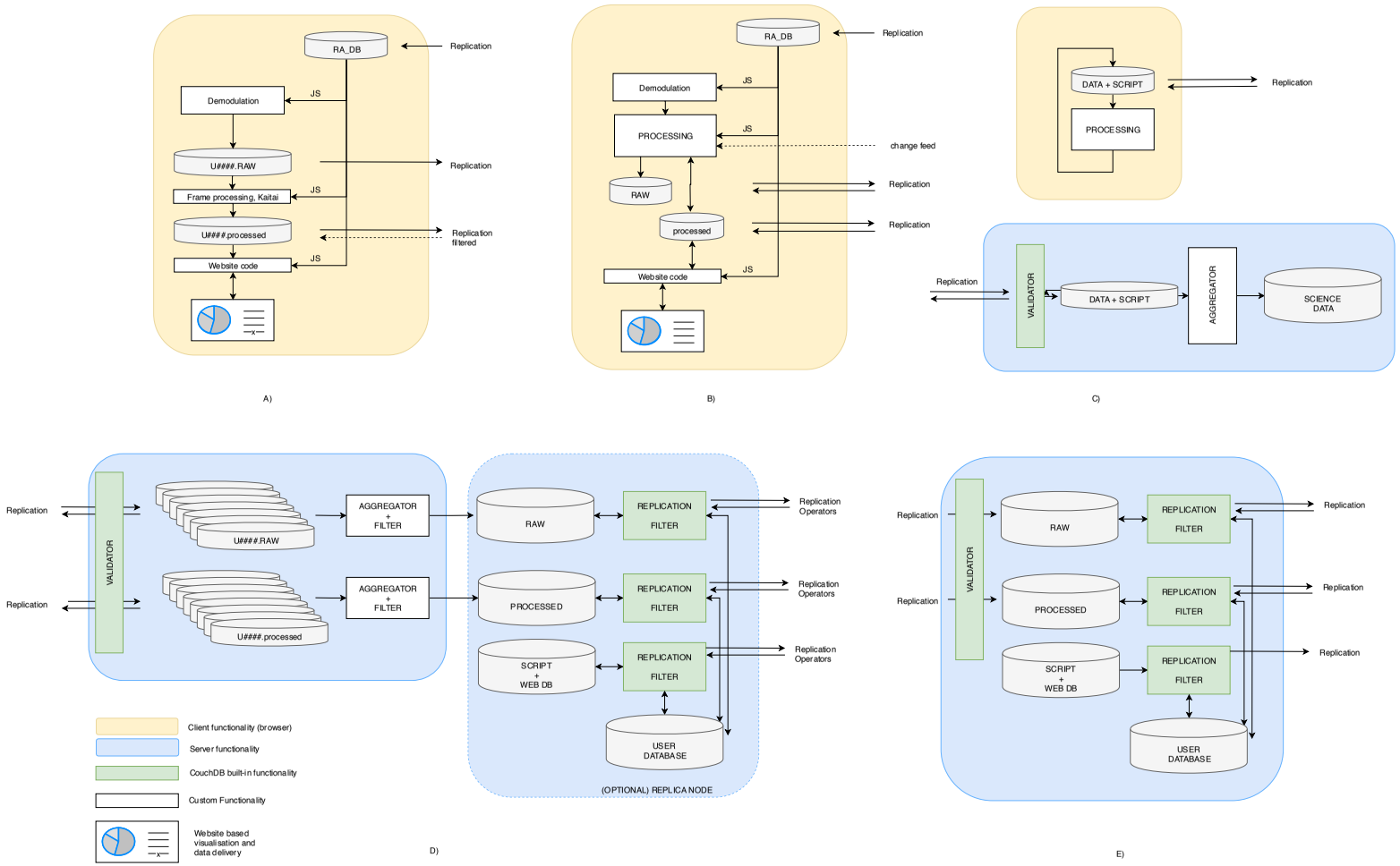

Figure 8.11: System Designs

will be replicated automatically on the server recovery. Secondly, the remote nodes, i.e. the client applications, can be leveraged for the ”swarm”-processing, allowing otherwise computationally expensive bit-flips recovery.

Client-side processing
The client-side processing is by design allocated to the RA's (if trusted) and the automated GN's. Further research is required to determine the computational impact, therefore the Spacecraft Operator group is ignored from the discussion.

The logic of the client-side processing is illustrated in the figure 8.12. The approach relies on the shared RAW and Processed databases, allowing participants to Read and Append data to the documents. It should be noted that the append operations are reversible, as each modification results in an additional copy.

The processing is performed in three major steps. First, the system ought to identify whether the telemetry frame is correct. Then, determine whether the telemetry frame is already received by the swarm. Lastly, system ought to append the outcome of the previous steps to the database. Additionally, the distributed system allows attempts to reverse the bit flip sequences. If multiple incorrect versions of the frame exists (analysis showed that 40 % of data is duplicate), the required bits to brute-force can be reduced significantly.

With a shared data store, the client application can validate whether the received, partially incorrect dataframe is readily available in the system and determine whether error correction is required. The approach hinges on the correctness of the frame identifier, the counter field in the housekeeping portion of the data.
The received frames are perceived as out-of-order by the server applications.This is a consequence of time synchronisations issues between the clients, but expected to arise more frequently due to SDR popularity in the future. The latter is especially interesting, since it allows the client application to select frames for bit-flip recovery based on their value. In case of Delfi-n3Xt and Delfi-C3 the frames received during a satellite pass, broadcast the instantaneous state of the space-craft, therefore it can be argued that the completeness of transmission sequence is not important. This is partially true, since certain mission states require higher number of measurements, e.g. thruster experiment. The variance in a single Delfi-n3Xt pass data is low, therefore, brute-force calculations are not necessary. Considering the future missions, potentially broadcasting the historic and whole-orbit on-board stored data, the complete frames sequence can be required.

Storing the partially incorrect frames (PIF), allows the brute-force a bit-error correction. With multiple versions of PIF from multiple receivers, the amount of brute-force work can be potentially reduced by aligning and comparing the bit fields of the frames, showing the frame differences. A further analysis is required to determine the correctness and feasibility of the approach, as identical bit errors can be received by multiple stations.

8.5.3. Considerations

With the entire system depending on CouchDB-PouchDB replication, the failure modes have to be investigated. Based on the HTTP based API, utilising HTTP GET, POST, PUT methods the PouchDB and CouchDB are well suited for the web browser applications, and by use of non-blocked standard port 5984, the connectivity issues due to networking are unexpected. The CouchDB was formely known in CouchApp deployment scheme, leveraging the design documents for HTML and CSS storage, delivering JS code as attachment. The CouchApps deviate from ”the right tool for the job” approach and are criticised by the developer community. The proposed architecture, utilizes the database as a storage and delivery system, but Deviating from the View and Querying mechanisms used in CouchApps, the PouchDB-CouchDB deployment, store the code base in form of documents, deviating from the CouchDB-View based hacks.
Where the locking, the inability to perform the Read or Write transactions, was the SQL based system limitation, the replication consistency is the CouchDB equivalent with regards to the system operation. Due to eventual consistency methodology, any document can be edited by any of client applications, causing data collisions. The strategy towards replication is simple: merge. As illustrated in the figure 8.12, all data manipulations can be resolved by merging the changes.

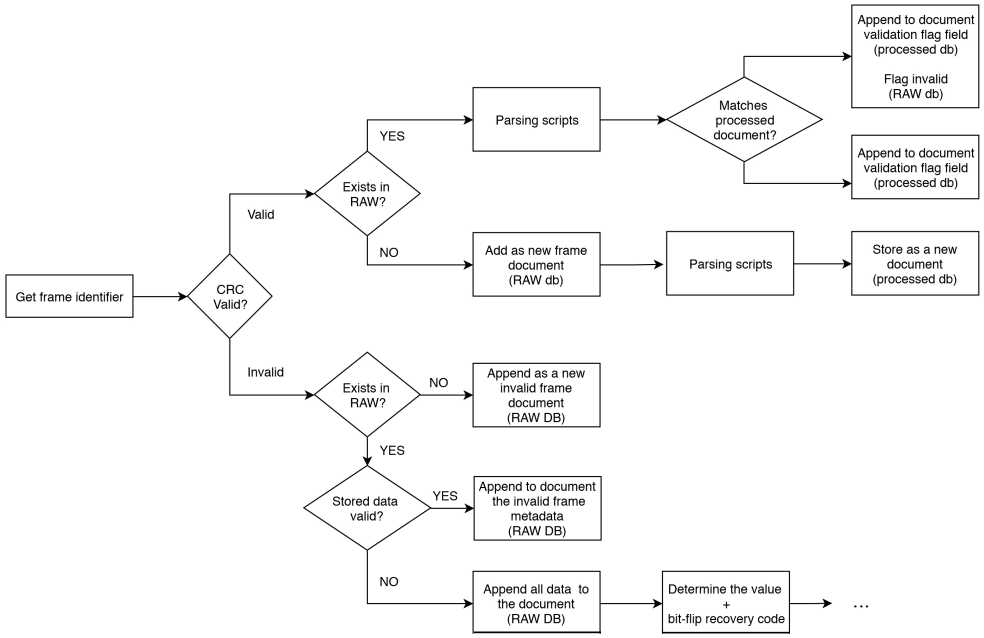

Figure 8.12: Client-side processing

8.6. Proof of Concept and Requirements Overview

The client-based processing and storage system concept presented in this chapter is built on the hypothesis that the use of remote clients will reduce the load on the central server. The PoC presented in this document addresses the client-based storage, as the computation load of the frame processing software is readily investigated and discussed in section 5.5.

8.6.1. Experiment Hardware and Software Setup

The hardware setup consists of a single server node and two client nodes. The server software consists of CouchDB (version 2.2.1) application, a Python built-in web server used to serve the webpage containing the client application and a monitoring plug-in. The monitoring is based on Prometheus project and provides a UI for data aggregation and graphing. Due to time constraints and Prometheus limitations, the axis on the graphs could not be modified for the report and are therefore different for each figure.

The client application is based on PouchDB (version 7.0.0) and is executed on a Windows 7 (SSD and 16GB RAM), and OS X 10.13.5 (SSD and 8GB RAM) hosts using Google Chrome (67.0.3396), Safari (11.1.1) and Firefox (60.0.2) browsers.

The server is connected with Windows host via a network switch, while the Unix host requires additional network hop: network router. It should be noted that the PoC did not attempt to investigate the networking effects, and setup is coincidental.

8.6.2. Experiments and Results

To investigate the feasibility of the data storage as a cluster with remote data nodes two phenomena have been investigated: the effect of the number of clients and the effect of data flow rate (burst vs continuous). Both are investigated by the series of the experiments:

1. Determine the baseline ingestion rate of CouchDB
2. Determine the replication rate of the server to remote client databases
3. Investigate the effect of simultaneously connected clients on replication speed
4. Investigate the effect of a ”burst” type of data ingestion
5. Investigate the effect of a ”burst” server to client ingestion
6. Investigate the client to the server replication rate

The data ingestion experiment, pushing 10,000 frames into the CouchDB at the maximum accepted rate, increases the system load, a combined metric of the CPU, network and drives queued process, to the value of two, as shown in the figure A 8.14 and fig:exp1fig2. The reason for the load increase during the experiment is the consequence of the RAM caching, as can be seen by the continuous increase in the RAM usage. The equivalent file size of 10,000 frames is 70 MB, roughly comparable to the RAM use. At 3min 15seconds mark, the RAM usage reduces significantly, followed by a spike in disk write operations. This is likely result of storage optimization, for example, data compression. The measured average data ingestion rate is on average 33 frames/s.

Figures B in 8.14 and 8.15 show the server metrics during database replication from the server to two separate clients. During the test runs, 10,000 documents containing raw telemetry frames have been transferred. The main observation is the lack of Disk Read activity hinting towards memory-based optimisation. High system load, but a low CPU utilization hints towards queuing and single thread processes, as seen in the figure B 8.15.

The experimentation with limited datasets showed a preference for memory-based storage, as indicated by < 1 Kb/s disk reads. The sudden decrease in RAM consumption at 5min50s is not an effect of the data read operations, but a reoccurring (caching) phenomenon that can be observed both at idle and during high load phases.

The observed network 'out' rate is 144% higher than network 'in' rate during the ingestion phase, however, the time required to transfer 100,000 frames is increased by 46 %. The increased traffic is most likely the consequence of sharding processes, requiring continuous updates of both client- and server-based database states. This is supported by the network receive rate. No significant difference between Chrome (3-4) and Safari (5-6) based applications could be observed during the experiment with the exception of the execution time.

It should be noted that throughout experiments Chrome was found to be 20% faster than Firefox and Safari. Furthermore, Safari was eager to perform memory-saving measures, reducing and temporary terminating replication as seen at 19min20s.

The combined Safari and Chrome run illustrated in the figures A 8.16 and 8.17 show a 42 % increase in CPU and 57 % increase in the system load. The comparison with a single connected application runs (3-4 and 5-6) show no significant change in RAM use, proving that the replication performance is primarily CPU and Network driven. Doubling in the number of connected clients did not increase the network traffic linearly. Consequently, the time required to replicate 100,000 documents to two clients applications simultaneously was increased by 20%, for example in case of Chrome from eight to ten minutes.

To simulate a close to a real-life scenario an experiment was conducted with two "virtual" operators (I, II) launching the application to monitor the spacecraft, while the new telemetry data is pushed by a radio amateur at the rate of 2 frames per second. Experimentation not presented in the report showed that the replication to the databases added to the cluster simultaneously or within a small (<1 minute) time window, is done with identical batches, allowing CouchDB to synchronize both either simultaneously or with a batch-based offset. Hence allowing the system to prepare each batch once and reuse it for each consecutive database. Therefore, to better simulate unpredictable real-life use, at 7min mark new application (III) was introduced to prevent re-use of the replication batches. The result is visible in the CPU and the load graphs of the figure B 8.16, showing an increase of 86 % in CPU and 77% in the load. The queuing of the Disk and Network can theoretically explain the non-linear change, i.e. 11% difference. However, none of both were observed, leading to the conclusion that CPU use is the primary source.

The replication speed from CouchDB node to application III was found to steadily decrease, with an average rate of 31.7 frames/s between measurements 11 and 12, dropping to 24.4 frames/s at point 13, and 20.5 frames/s at point 14. At measurement point 15, the data ingestion of 2 f/s was terminated, showing that all client databases were up to date both in terms of on-screen frame count and network (out) activity.

Figures 8.18 and 8.19 combine four different experiments. First, between the measurement points 1 and 2, a burst of data (1000 frames) is sent to PouchDB. No residual hysteresis was found in terms of Load or CPU, and with no connected clients, as well as the networking. The increase in RAM can be accounted to memory optimization of the database. Between point 2 and 3 no experiments were performed, showing the idle state of the system. At measurement point 3, four clients were connected with a time offset of two seconds. All of the clients required the complete dataset replication. During

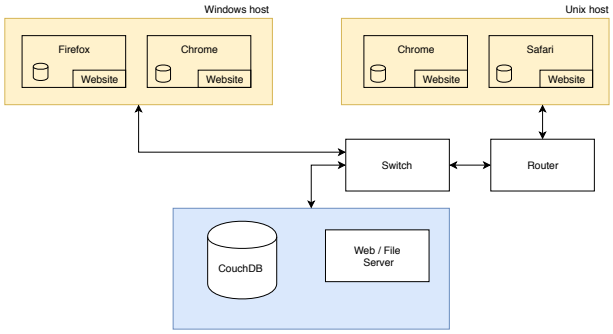

Figure 8.13: Experiment setup

the run, a steady increase in the system load can be observed hinting towards queuing. The RPi network interface is capable of 11.5 MB/s, however, during the experiments never exceeded the 1.5 MB/s. The measurements points indicate the completion of replication by Windows hosted Chrome and Firefox applications for points 4 and 5, and Unix based Chrome and Safari application in points 6 and 7. The application executed on the same host show similar results in replication speed, however, the difference between Windows and Unix hosts was substantial. On average, the Unix hosted applications lagged by 800 frames at 10min mark and increased up to 1000 frames close to measurement point 4. Furthermore, between measurement points 6 and 7 CouchDB "lockups" were observed, temporarily terminating the database connection. The experiment between point 3 and 7, replicated database with 11.763 frames to all of the connected clients.

The third experiment focused on the study of Client to Server replication, simulation RA contributions. At measurement point 8, the CouchDB and PouchDB databases were reformated. This removed all previously stored data freeing the server memory. At measurement point 9, telemetry data was placed on one of the PouchDB (500 frames) clients and replicated to the CouchDB server until measurement point 10. It can be seen that the replication to the server places the documents directly on disk, and utilizes the memory once replication is completed. In practice, this increases the performance as replication is much faster than direct data ingestion from python clients that places data in memory first. A similar experiment is performed between measurement points 14 and 15 with a larger dataset of 3000 frames.

The fourth experiment investigated the query performance of the CouchDB. The initial hypothesis, that the use of remote clients will reduce the load on the central server can be expressed mathematically as: $Load_{replication} < N_{queries} \cdot Load_{query}$ With $Load_{replication}$ the server load due to replication to a single client, $N_{queries}$ the number of expected queries throughout mission life per client and $Load_{query}$ as the average server load by a query. The server load due to queries is shown on the figure 8.18 and 8.19 as measurement points 11 and 12, and executed continuously between points 12 and 13 with a 10sec interval simulating multiple connected clients. Each query pulled 3000 frames from the database and took 2.3 seconds to make data available on the client side.

1. Determine the baseline ingestion rate of CouchDB
2. Determine the replication rate of the server to remote client databases
3. Investigate the effect of simultaneously connected clients on replication speed
4. Investigate the effect of a "burst" type of data ingestion
5. Investigate the effect of a "burst" server to client ingestion
6. Investigate the client to the server replication rate

8.6.3. Experiment Conclusion

The experiments did not prove the hypothesis of a reduced server load by data replication. The instantaneous server load due to query execution is comparable to replication load but much shorter in duration. A query pulling 3000 frames has an average runtime of 35 seconds, while replication of similar dataset would require 2.4 minutes. It should be stressed that once replicated, the queries can

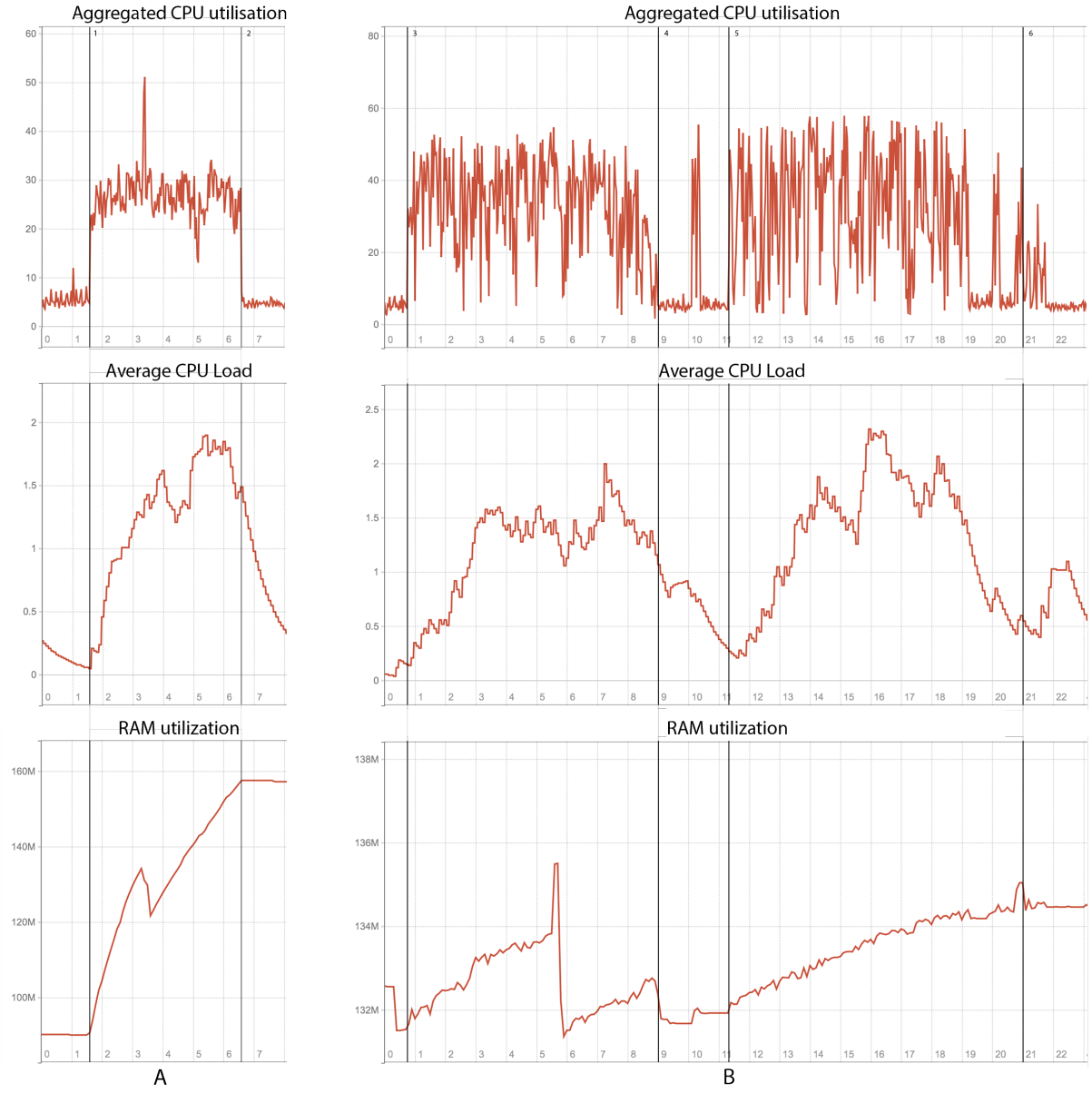

Figure 8.14: PouchCouch PoC experiment one and two combined

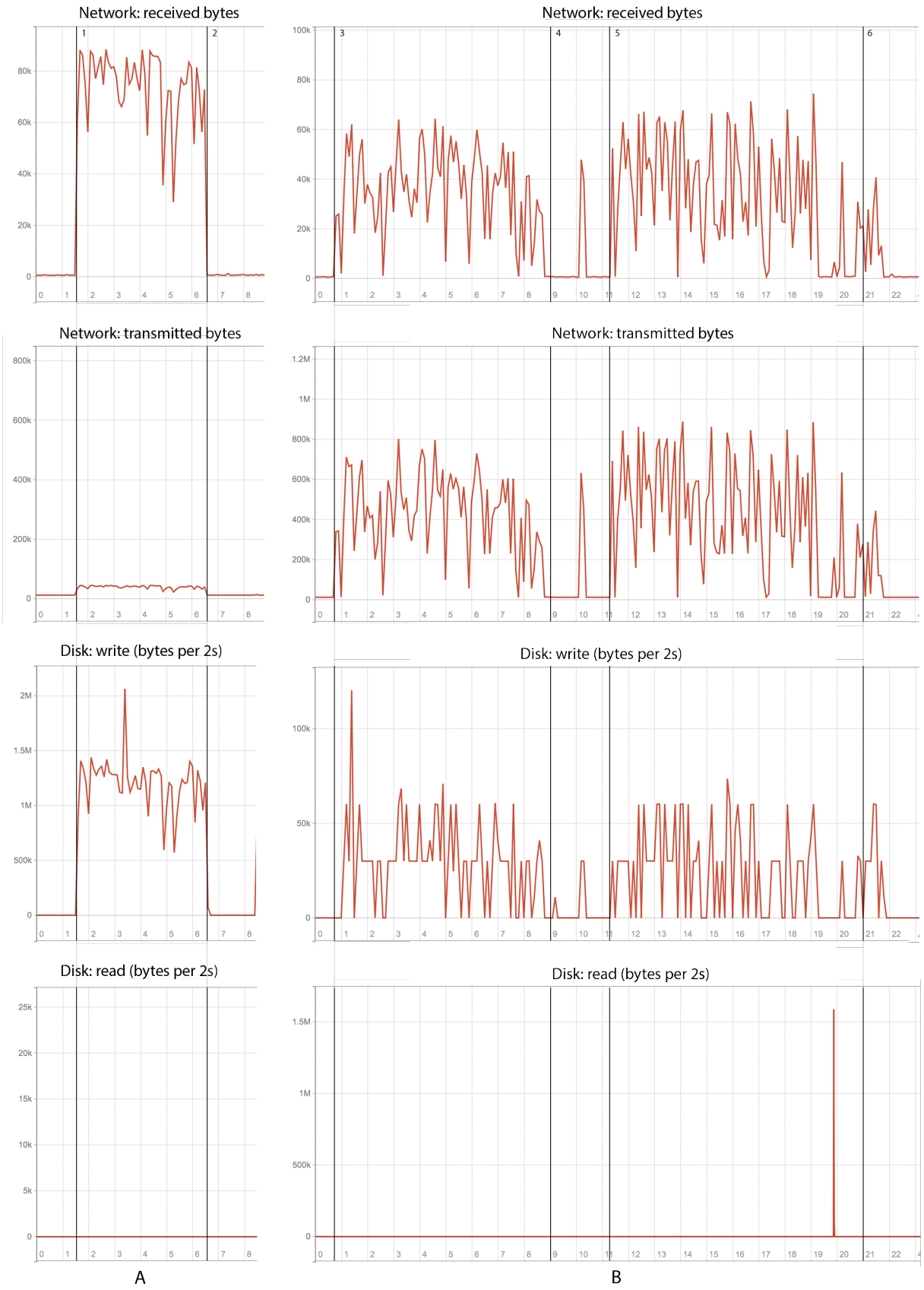

Figure 8.15: PouchCouch PoC experiment one and two combined [cont.]

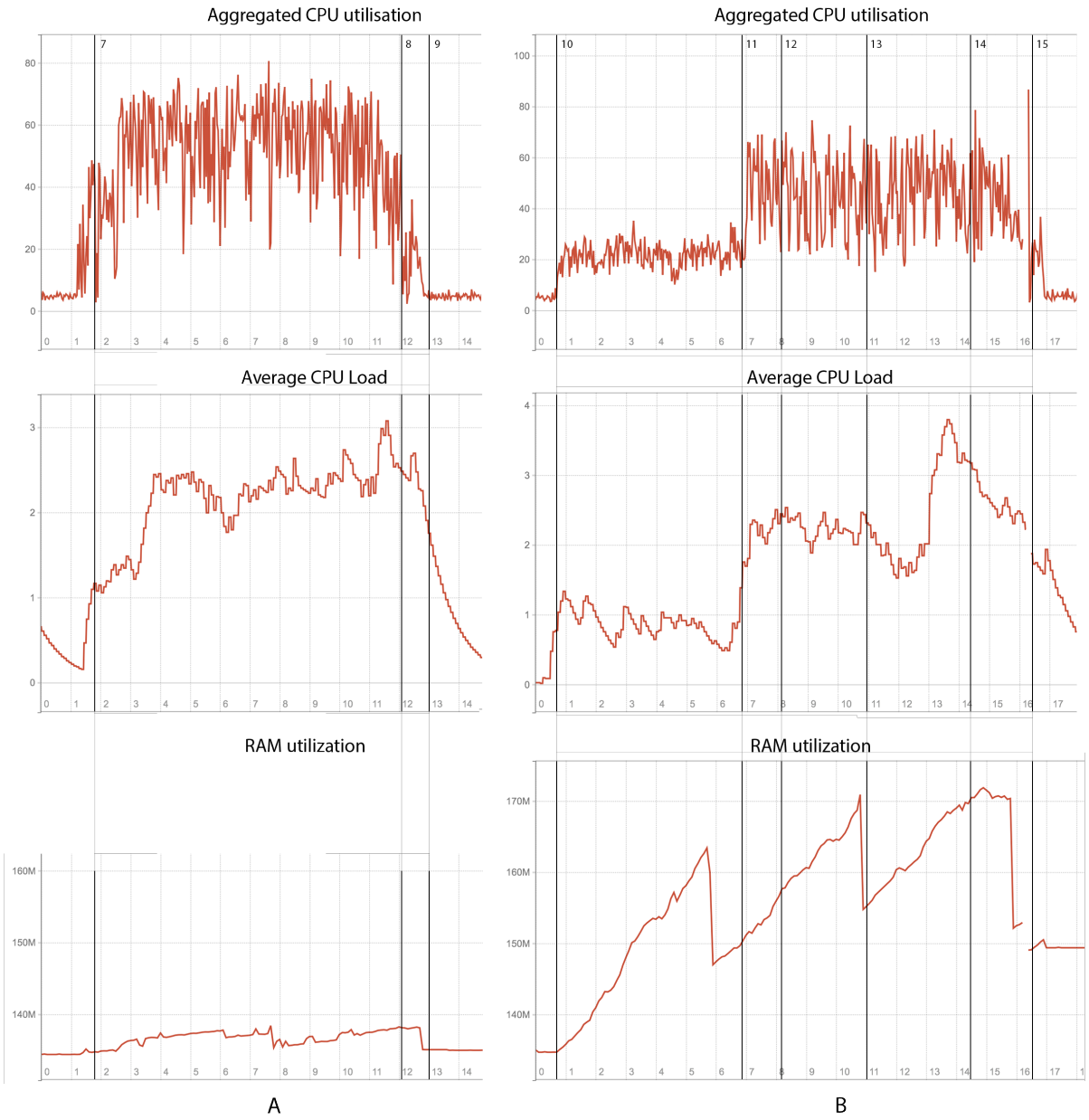

Figure 8.16: PouchCouch PoC experiment two and three combined

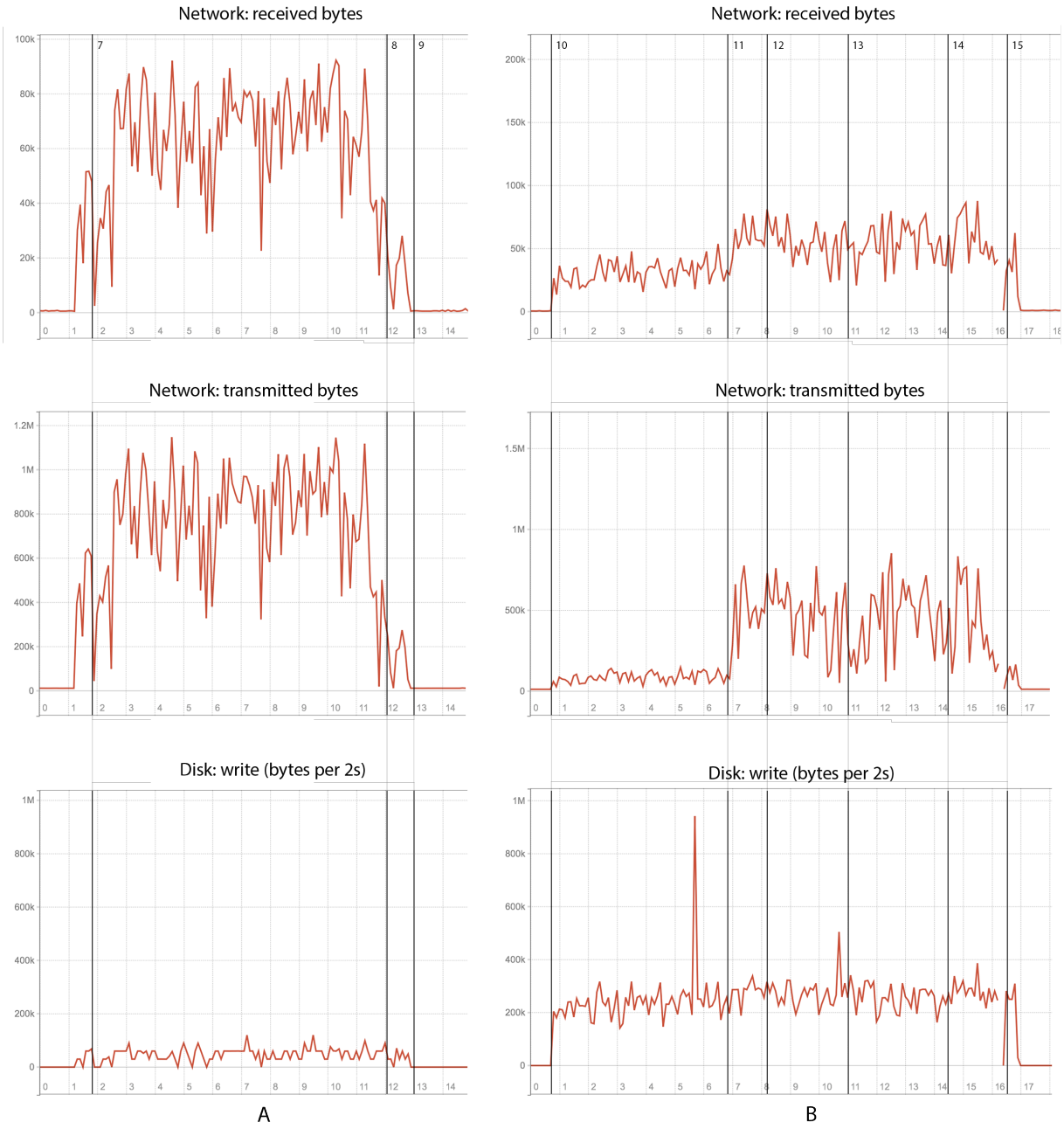

Figure 8.17: PouchCouch PoC experiment two and three combined [contd.]

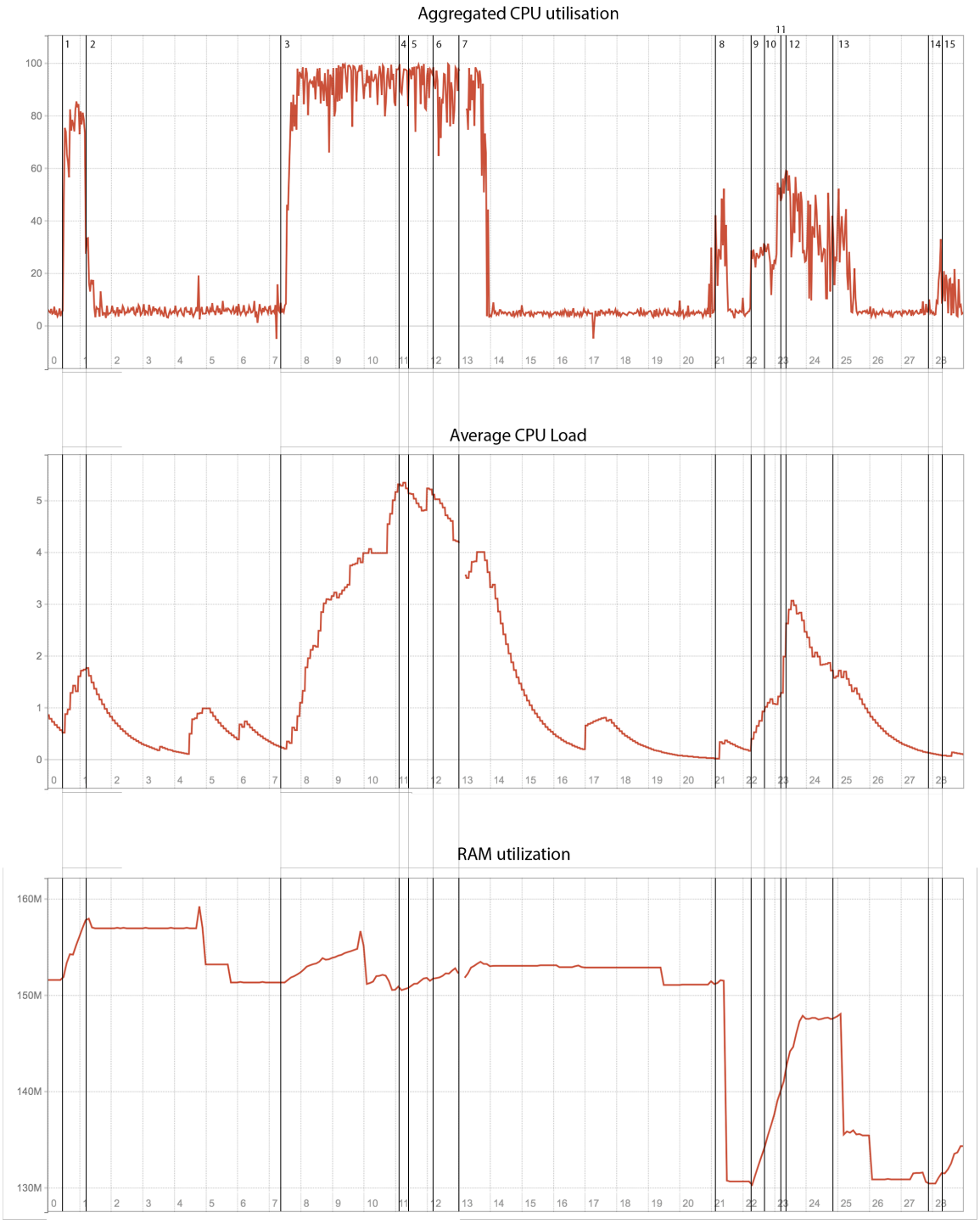

Figure 8.18: PouchCouch PoC experiment four

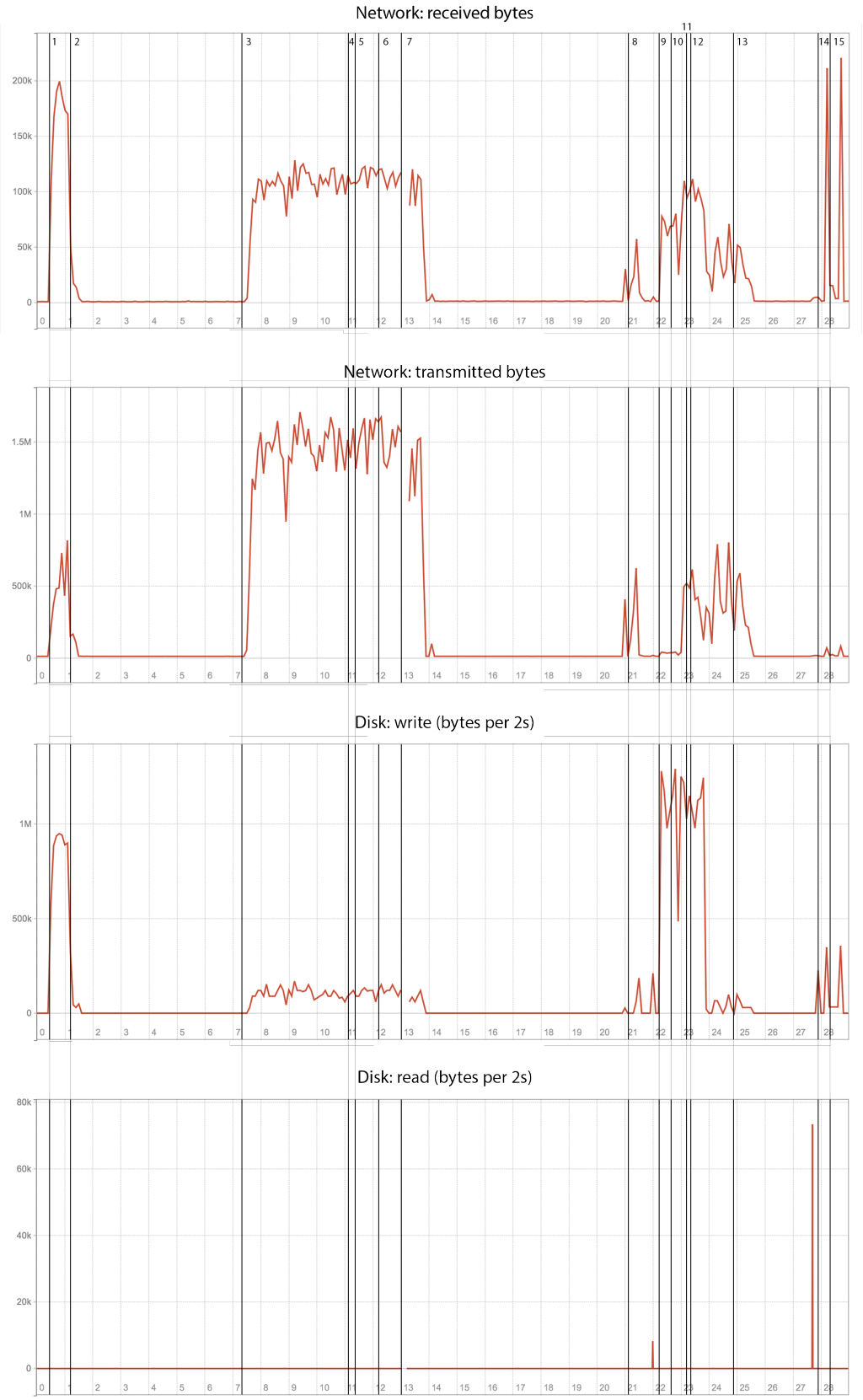

Figure 8.19: PouchCouch PoC experiment four [contd.]

be performed on the client side without further impact on the server operations. Therefore ideal for the workloads requiring frequent data transversals such as machine learning model training and analytics.

In case of Delfi missions, the processing of telemetry frames has a low computational impact. The entire Delfi-n3Xt dataset can be decoded and processed in minutes, while replication of 220,000 frames to a single connected client would take approximately 2.9 hours.

The experiments focused on small (182 % smaller than Delfi-n3Xt) datasets, therefore placing data directly in memory, hence providing the best possible performance. Even under these conditions, the performance is lacking. It can be argued that this is due to selected hardware, which is true, however, validation runs on a high-end Windows machine did show a limited increase in the replication performance for CouchDB system. A number of optimizations can be designed, such as data compression to increase of the batch size for each replica shard. However, the load to performance ratio is high and much higher than querying or frame processing. Furthermore, placing clients outside the managed network, as was the case in the experiment, will degrade network performance and introduce uncertainties, i.e. frame drops, routing errors etc.

Based on aforementioned it was concluded that bringing data to processing nodes, i.e client machines, is computationally expensive when database replication is used. Different methods of data delivery to the client applications can be considered, however, at the end of the day, networking will become the bottleneck and with limited computational requirements processing server side is both faster and more robust.

8.7. Lessons Learned

With the user data stored both locally as on the server, each RA is able to review the data locally, without server-side querying. Useful for the personal usage statistics, the technique provides an unique insight on the user stored information that exists on the server. With minor adjustments, system can be expanded for advanced retention policies, for example linking the personal user information to reception frames, as long as data is stored on the user side. Therefore natively implementing the right to be forgotten.

The client-driven architecture proposed in this chapter proved to be a complex research topic from both the users as the system point of view. Leveraging the client applications, in theory, lessens the computational needs for data processing of the server application. Furthermore, the PouchDB based proof-of-concept provided a high degree of flexibility, providing required features at a low development cost. This section aims to answer two questions: 'Why the PouchCouch based system was discontinued' and 'Why the Client leveraged architecture was discontinued', addressed in sections 8.7.1 and 8.7.2.

8.7.1. PouchCouch Implementation Flaws

The PoC of the client-leveraged system introduced in this chapter was found unacceptable for the further development. This section aims to provide the reason behind the decision and to refine the requirments for the following project iteration.

Security

Deployed as a browser application, the data store and the source code fully accessible to the end users. The initial assessment provided in section 8.2.2 focused primarily on the data integrity, such as users (accidentally) modifying the stored telemetry frames. However, this is only a small fraction of the security topics that need to be addressed.

The PouchDB application requires CouchDB system to be publicly accessible over the internet. All data, such as raw and processed telemetry frames, processing and visualization scripts, private and public user information, is exposed to the internet. In authors' personal opinion, the software is considered insecure unless proven otherwise. Secondly, the application security is time before adversary is able to abuse the system. Albeit left out of the discussion, VPN-based networking was concluded as one of the possible alternatives to resolve the problem.

The second major security threat is the PouchDB system deployed at the end users (clients). The data stored in PouchDB is synced with the CouchDB directly. Implementing security features on front-end is pointless, as the user is able to modify the source code at runtime. Implementing security features on the back-end may be a lost cause since the data readily resides in the database. Filtering can check for unexpected data types, but not the content, allowing users to submit maliciously crafted

data sets. Deleting data from CouchDB is a complex process, any deleted file is marked for deletion first, propagate through the nodes and deleted once the garbage collection has been triggered. Manually wiping a CouchDB document is not enough, as it can be replicated back from any other node.

The attack vector from the point of view of an authenticated user is immense. First, the malicious user is able to replicate large volumes of arbitrary data to CouchDB, wasting storage, causing DOS, potentially discarding the new incoming data. The second attack is the remote code execution. With write permissions for the CouchDB databases, the malicious user is able to replicate crafted JavaScript object. At the time of writing, one remote code execution (RCE) vulnerability is known for CouchDB system [109][110], allowing privilege escalation attacks on the CouchDB host server. Lastly, depending on the server-side implementation and the acquired user privileges, the malicious user is, theoretically, able to replicate malicious code to all clients.

All of the issues mentioned above are critical, hard to detect and potentially impossible to recover.

Implementation

The Pouch-Couch application relies on CouchDB for all of the back-end services and PouchDB for the majority of front-end services. Once implemented, nor PouchDB nor CouchDB can be replaced, actively locking into the CouchDB en PouchDB ecosystem. It should be noted that the commercial CouchDB (Couchbase) alternatives are available, however, due to open-source requirement [GN-AUX-LIC-01] cannot be considered for the production.

The application is designed around on PouchDB as primary data and source code delivery to the client application. Changing to a different data store is possible, however, the business logic, as implemented in the proof-of-concept, needs to be rewritten from scratch, as no CouchDB-compatible browser-based data stores were found.

As a disclaimer, the available browser-based natively sync-able datastores, such YDN-DB [111] rely on an API (REST), allowing both the client (YDN-DB) and the server-based database to be changed. Alternatively, the database may rely on custom sync algorithm such as Loki-JS.
The use of these systems in PouchCouch deployment would require extensive customization to facilitate the PouchDB and CouchDB database replication.

The PouchDB browser storage proved to be unreliable, as some versions of Chrome deleted the persistent storage after closing the browser. Furthermore, offline use of data created collisions and conflicts in document versions, unexpected due immutable nature of the telemetry data.

The PouchCouch system is limited on the storage provided by the client application. It is clear that dataset cannot be replicated entirely to the clients, hence, limiting the data query scope. The filtered replication allows control on the replication, however, applying on all client applications. [failure GN-DF-06/07]

Architectural Design Flaws

The failure of the PouchCouch system can be credited to the lack of separation of concerns. Albeit providing the required functionality and scalability, the architecture is bound to a single ecosystem built around CouchDB. The system is responsible for the data storage, querying, delivery, client-server communications, user authentication and security. Failure of a single of the component renders the entire system inoperative.

The proposed advantage of the remote query execution, as an attempt to outsource computational (e.g. data science) tasks to the remote clients, provides to be less useful than anticipated. The majority of tasks require data aggregation, leading to complex data operations between multiple clients. The orchestration between users is possible but limited due to unpredicted behaviour, due to unknown the user session duration: the user may close browser window at any point in time.

Use of PouchDB and CouchDB requires the application to utilize document-based data store. Furthermore, the data is eventually consistent, hence providing different query results across the clients.

The use of browser deployment requires frequent codebase update to support new browser versions. Practically, this means that the system is not future-proof and requires continuous support throughout the mission lifetime. Meaning that PouchDB has to be updated periodically.

8.7.2. Why not Client Based Applications

The research provided by the PouchCouch application showed an implicit dependency on the user. Having a user in the loop, not only induces the chance of human-errors but also manifests as a new

vector of attack. The user-induced vulnerability is well popularly known as ”Don’t trust the user” [112], generally applying to the sanitation of the input parameters such as website forms. The distributed system requires processing validation, which on its own is a weak band-aid solution. Furthermore, use of PouchCouch system introduces complexity, not in tooling but in the use of tools.

The PoC showed the system capabilities on a limited scale, expanding to 100s of users inevitable cause unforeseeable secondary effects. The server-client replication proved to be computationally expensive for the CouchDB server, therefore to benefit from a distributed, client-leveraged system, the client processing should match the server efforts. The assumption of gain from client-processing was disproven by the PoC, as the processing of 3-months worth of Delfi-n3Xt data could be decoded under two minutes in Kaitai and Python on a single machine.

It was determined that a distributed system is well suited for the computationally intensive, but data extensive applications. Which is not the case with the Delfi data sets. The data science tasks, such as model training requires large and often complete datasets, and due to client storage limitations (it is not possible to require contributors to provide 300GB of hard drive space), the majority of these tasks are not feasible on the client side.

Another PouchCouch observation that applies to the entire architecture is the server load due to data replication. Any client connecting to the cluster introduces a load on the system while the data server and client data stores are synchronizing. The phenomena require further investigation, however, the initial assumption that leveraging of the self-sufficient client application reduces the server node does not hold.

8.8. Conclusion and the Next Project Phase

The chapter proved that use of client applications for the computations is not advised due to the limited scope of the computations. Hence, replication of data has a higher load on the system in comparison to the nominal data request.

The proposed PouchCouch system showed a weak architecture with a high level over interdependencies between critical system components. The long-term use of the system is questionable and any systems changes are error-prone due to complex multi-database operations.

To solve and prevent the problem in the next project iteration required further research in Software Engineering (SE). It should be stressed that the structure of the report may indicate the SE as an afterthought of the project, leading to the conclusion that SE tools were considered once the PouchCouch system was determined unfeasible, this is however not the case. The Separation of Concerns (SoC) introduced in the section 6.1, was studied and applied prior to the PouchCouch system but was placed in the chapter to improve the readability. From the point of view of SoC, the software components are well separated and connected via well-defined interfaces. Even more, the processing components can be removed and tested separately, as the rest of custom written software.

9

Server side processing

9.1. Introduction

Based on the analysis presented in chapter 7 and refined requirement extracted from PoC of the PouchCouch system, the server side processing required further investigation.

The DPS is considered to be the system connecting data ingestion API, i.e. the abstraction for the client application to the data delivery API. The scope of the project, therefore, encapsulates the functionality between both API's, as illustrated in the figure 9.2.

9.1.1. Client application function

The analysis of the legacy systems use cases presented in section 3 showed two primary goals: data acquisition and data visualization. The telemetry processing is added to reduce the latency of visualization and reduce the load on the server. Furthermore, as shown in figure 8.4, historically the client application deployed in this configuration were not updated timely by the RAs. Therefore the system requires the ability to auto-update to ensure the correct results of visualization.

To increase the applicability of the designed software implementation for the future missions and limit the dependency of the client system, thus reducing the possible data inconsistencies and errors as found in the Delfi-C3 system. The proposed architecture is Delfi-n3Xt inspired and requires the client applications to submit raw telemetry frames to the server for the processing. Whether the processing is done in parallel on the client (DuDe) or whether the data is pulled from the server application as shown in the figure B 8.2 requires further research.

9.1.2. System Scalability: key for the architecture

The straightforward solution to comply with the potentially-unbound data growth [GN-ING-12], and redundancy requirements [GN-DS-1], is a multi-server deployment. With the use of multi-server systems, the scalability concerns have to be addressed. First, the use of a single satellite mission as Delfi-C3 require fewer server resources than multi-satellite missions. While it is assumed that more simultaneous satellite mission will be operated in the coming years, the opposite is equally feasible. Hence, the system should be able to grow or decrease in scale, without extensive reconfiguration or adjustments.

The PouchCouch proof of concept showed the importance of the architecture design towards the supportability and system complexity. The limited maintainability, due to the complexity of operations and the high risk of failure due to the PouchDB dependency is in many ways comparable to the limitations of the Delfi-n3Xt implementation, the system that was initially found to be hard to maintain and develop C.

The scalability can consequently not be added as an afterthought, as it will increase complexity (as was the case Delfi-n3Xt frame processing that led to data loss) and reduce the ability for modifications to facilitate new satellite missions.

9.1.3. The approach

The architecture design methodology is based on the Clean Architecture introduced in section 6.2 and applied as a system architecture tool. The tool intrinsically leads towards a high level of system decomposition, and hence, abstraction. It can be argued that the use of highly abstracted components leads to a reduced interdependency of the subsystems but an increases complexity due to hidden assumptions and number of components.

Section 9.2 reiterates the requirements, data structures and requires processing. The lead to three architectures discussed in section 9.3. The applicability of Stream based processing is then assessed in section 9.4 which lead to conclusion to apply Lambda Architecture as described in section 9.5.

9.2. Problem analysis

This section aims to provide a brief summary of the system requirements, lessons learnt from legacy systems and PouchCouh PoC. First, data transformation is summarized in section 9.2.1 focusing on the conversion of the telemetry frames to data. Querying of the processed satellite data is discussed in section 9.2.2.

9.2.1. Data transformation

The primary data transformation seen in all Delfi mission is the conversion of the received binary satellite frames to a set of parameters describing the satellite state. The data processing is described in section 3.5 and leads to the use of Kaitai framework introduced in section 5 as the telemetry frame parsing and decoding tool. As discussed in section 5.4 the approach hinges on the use of Kaitai decoded objects as the abstraction of raw telemetry frames, and wherever possible the use of one of the following programming languages: Python, C++/C#, Go, Java, JavaScript, Lua, Perl, PHP or Ruby. If an alternative programming language is required, wrappers can be used, but that would not be preferred.

As discussed in the section 3.3.1, the RAs are not allowed to broadcast telemetry frame reception acknowledgement to the satellite. Therefore, the data cannot be split arbitrarily across multiple frames with the use of AX.25 segmenter, as discussed in section 3.3.2. The aforementioned guarantees that the satellite frames are autonomous and decoupled from the sent sequence.

The data transformation is to a high degree independent of the selected architecture as summarized below and in figure 9.1A.

1. Consumes the satellite telemetry frame and receiver (RA) metadata
2. (Optional) Pre-process frame: bit ”unstuffing” (see section 3.7)
3. (Optional) Pre-process frame: byte-wise manipulations (see section 5.2)
4. Asses telemetry frame validity [GN-PR-5]
5. Decode and parse the frame ([GN-PR-4,6,7, GN-PR-RAW-1] see section 5.4)
6. Aggregate and store the data

The design of the decoding components, as illustrated by the figure 5.2 is flexible and may be changed at any point in time.

With the reception date and time present in the metadata, the TLE method [GN-PR-RAW-2] can be applied to allow correlation of the satellite location to the frame. It should be noted that storing location per frame is a potential resource waste and should be re-evaluated. Furthermore, the TLE calculations require fetching of TLE file from online resources [GN-PR-INT-1]. The fetching should be decoupled from low latency queries, as additional processing time degrades query performance.

Analysis resented section 3.3.1 shows the presence of data duplication in the Delfi-n3Xt dataset. Data duplication is the consequence of RA participation and is caused by the overlap in the reception area within the European region.

The duplicates intrinsically carry information and should not be deleted. However, the requirement [GN-PR-19] demand aggregation of the metadata. It can be argued that the aggregation is an analytics query and ought to be discarded from the Data transformation process. However, the presence of duplicates in the processed database is a weakness and should be avoided. The de-duplication can be achieved by running a whole-data query, hence re-processing the telemetry frames as a batch layer, as illustrated in the figure B 9.1.

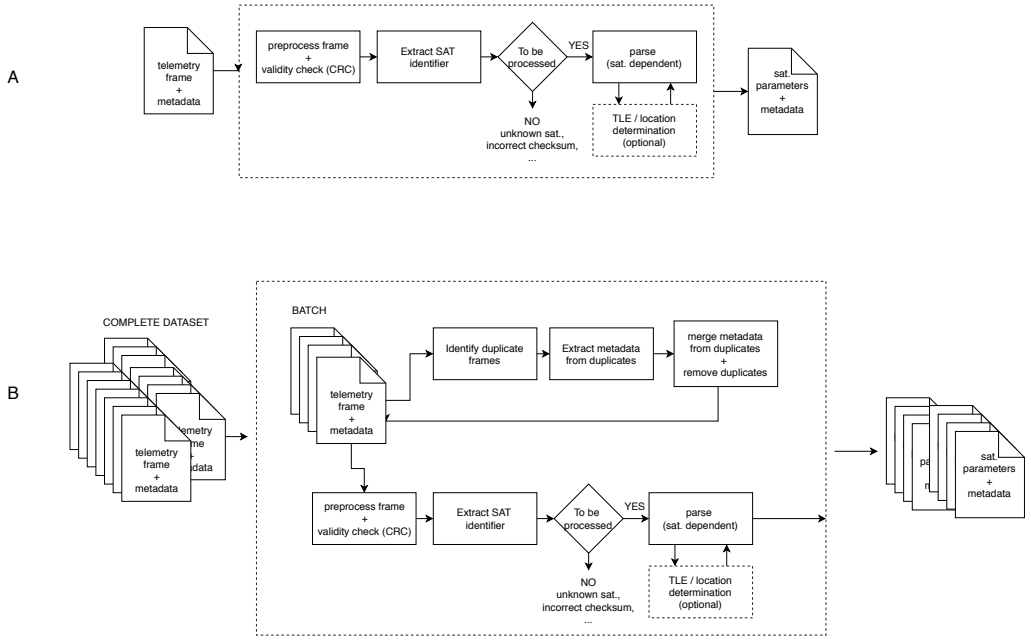

Figure 9.1: Telemetry processing queries: a) Stream or Batch b) Batch

9.2.2. Data analysis

During the mission lifetime, the TPS is responsible for data storage. The system is tightly coupled to the data ingestion system receiving data from the client applications (e.g. DuDe). Upon mission termination, the legacy systems are solely utilized for the data storage.

In the legacy applications, database based storage was utilized, allowing for data querying which is used directly in the web application. The use of database systems potentially allows for data analysis, however, limited to hard-coded queries, primarily focusing on visualizing the datasets. The actual analysis of the satellite data was performed offline on researchers local machines, using COTS software tools (Excel/Matlab).

One can distinguish between three types of data queries. First, queries can be applied directly to the raw incoming data. The requirement [GN-UI-RA-01] states that the system should facilitate per Radio Amateur statistics, related to the frame reception. This can be seen as the aggregation of the metadata, as can be seen in figure 9.1.

The second type of queries is recurrent queries. For example, UI or web application visualization. Secondly, the requirement [GN-UI-GEN-04] can be satisfied by training a model on the past historic data, accommodating for the RA's geographical location, position and the equipment. Hence, not only calculating the next pass but predicting the chance of reception and to be received frames. The model application is, in this case, a recurrent query.

Lastly, one should consider ad-hoc, random queries. Queries originate from the user interaction and can originate from on-demand graphs used in UI or analysis of the data.

9.2.3. Data characteristics

Study of the n3Xt dataset showed a discrepancy between frame time appended by the spacecraft and server reception time. This was determined to be a consequence of three facts.

First, due to the distributed nature of the RA's network, the telemetry frames are likely to be received with a delay and out of sequence. Additionally, as discussed in section 3.7, the use of SDR (data replay) may result in the frame submissions hours, days and possibly months after the initial reception.

Secondly, differences in server and client system time were observed, that can be accounted by clock drift, disregard of automatic time synchronization or use of non-standard time servers.

Lastly, the distributed network RAs has an leads to an overlap in reception, leading to telemetry frames having multiple reception timestamps.

9.2.4. Architecture assessment

Processing latency Applies to data transformation. The raw telemetry frames have to be processed as soon as possible and preferably near real-time. According to requirement [GN-PERF-4], the latency measured from ingestion by the server to become available in visualizations should be less than one second at data ingestion rate of 10 frames/second [GN-PERF-2]. The data analytics queries latency is driven by the client application and UI and require further investigation. The data science, as well as analytical queries operating on the complete unprocessed dataset, have unknown performance requirements [GN-UI-OPS-01] at time of writing.

Data size Dictated by the requirement [GN-DS-7], the system should be able to store at least 1 TB of raw, unprocessed data. The data is unbound in size with incremental growth. In theory, the system should be able to store data from the previous missions. Optionally, the system should be able to store and process non-satellite data, for example, generated by the test software, by Earth-based instruments used for system testing, calibration and analytics.

Policies with regards to flow control The minimum ingestion rate per satellite, as dictated by [GN-PERF-2], is ten frames/second. When exceeded, the system should store incoming data continuously, potentially, by throttling the processing scripts to mitigate the data loss.

Data duplication Data duplication in the processed database is allowed for near-real-time processing, however, the duplicates should be removed and aggregated as metadata [GN-PR-19].

Out of order data The raw and unprocessed data should be stored in the order of reception by the server. The processed data store should store the data in accordance with frame unique id, hence ordering in terms of the sent date by the spacecraft.

Processing guarantuees Wherever possible, an effort has to be done to ensure 'exactly-once' policy. The system should not introduce additional data duplication except that introduced by the RA community.

Scalability and Elasticity Due to in-house deployment requirement, the scalability has to be achieved on the locally maintained servers. The elasticity, scaling up/down is not feasible in the classical Cloud deployment sense and requires further research.

Integration, Extensibility The system should facilitate extensibility, allowing integration with other system components. The vendor lock-in should be avoided at all costs. In accordance with the Clean Architecture, the major system components should be exchangeable. Moreover, the metric encapsulates the easiness and ability to make system or component changes to accomodate for new mission needs.

Accesibility The system should only rely on broadly available, open-source systems that are actively developed and supported by the community. Furthermore, well documented and easy to use tools are prefeed.

Hardware Failure Being deployed on-premise, the hardware failures are likely to occur. The system should be able to tolerate failures and wherever possible to (automatically) recover from the errors. Failures in the storage systems shall not cause data loss. The system should remain operational with data storage node(s) failures. Failures in the data ingestion systems shall not result in data loss. Ingestion system should operate in high availability mode and tolerate node(s) failures. Failure of data processing subsystem should be tolerated by the system and shall not cause any subsystem faults. The data delivery node (presentation layer) failures and requirements are considered outside the scope.

Fault tolerance The system should be recoverable from human errors in the data analytics and data transformation. The system shall tolerate errors in telemetry such as bit flips. The system should tolerate subsystem failures.

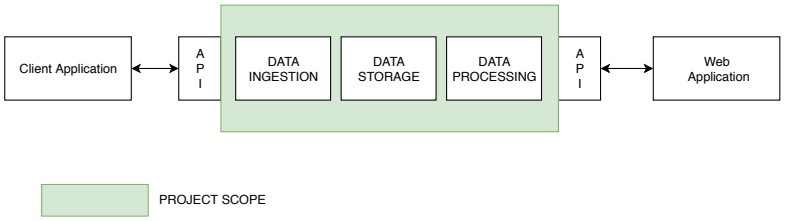

Figure 9.2: Project Scope

Tracability The system should generate logs for debugging and problem investigation purposes. The system should store all sent the telecommands to the satellite. The system should allow appending extra fields to the data for traceability purposes (e.g. timestamps, user notes)

Machine Learning The system should facilitate Machine Learning and iterative query design.

9.3. Related work

9.3.1. Abstraction

The challenges in the satellite data processing in the educational institutions are not new. In 1995, Berczuk [14] proposed a pattern-based approach to increase the software quality on organizational and programming levels. The issues identified by Berczuk are attributed to the distributed nature of the ground segment (the reception component), small distributed development teams and a limited layover of the personnel between the projects [14].
On the programming level, Berczuks solution is identifiable by loose interfaces and factory-patterns for the data parsing and interpretation subsystems. Berczuk argues that the use of software patterns ensures the longterm compatibility, while the loose interfaces facilitate the re-use of software.

The combination of Berczuk's method with the Clean Architecture methodology leads to a blueprint of a system with a limited (one way) dependencies, decoupled components and standardized interfaces. The latter two facilitate the maintainability, as system components ought to be exchangeable, allowing for the change of the components. The stand-alone components are preferred since, in theory, they provide the best specialization with the lowest complexity. By decoupling the system components and providing well-defined interfaces, the system can be designed, modified, implemented and tested independently, increasing the overall quality by reducing the individual component complexity. This sparkled the research towards a highly decoupled and (potentially) distributed system.

9.3.2. Microservices / Service orientated architecture

One of the side effects of the PouchCouch system (see chapter 8)) was the ambiguity of the system with regards to the server-side and client-side operations. One can, therefore, argue that the client-application, acting on the data input is a "service." This is related to the service-oriented architecture (SOA).

In SOA architecture, the functions are performed by a discrete application, i.e. services, by ingesting the data in the form of a message and passing it through to the next service in accordance to the established business logic. Furthermore, the microservice architecture decouples the services on the network level by deploying them as stand-alone containers. The information and data transfer between the services is the task of a messaging system, providing queuing of the messages and ensuring the delivery. Figure 9.3 illustrates the possible microservice components for the data ingestion portion of Delfi system.

It is important to realize that in the microservices architecture, all services ingest data, perform an operation and pass data to the next service asynchronously, thus without an acknowledgement to the sender. Due to use of the queuing/messaging system, the services can be removed and added to the system while in operation, for example by using Docker Swarm deployment, thus promoting the system elasticity. The service decoupling and due to the use of the defined interface, i.e. the messaging system, makes each service fully autonomous and self-contained, thus allowing the use of mixed languages, frameworks and data storage methods.

The microservices are in many ways comparable to the method calls in Object Oriented Programming

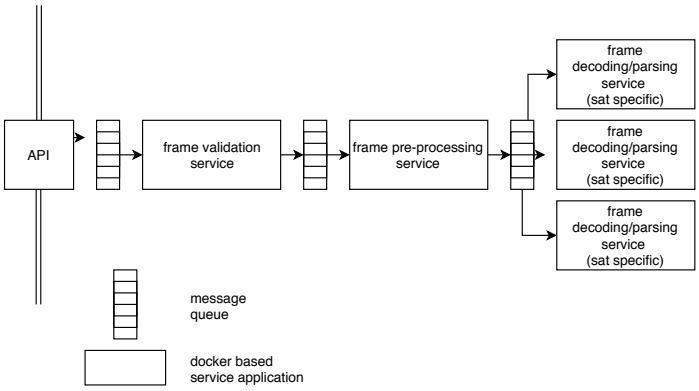

Figure 9.3: Frame parsing in microservices architecture

(OOP), and, therefore, provide familiarity with the subject. However, one should keep in mind that the service calls are executed asynchronously, with limited control over execution order or time. Each parallel service will likely execute at different rates due to server utilization, the hardware, network distance (hops) and many other transient factors. Therefore, the microservice system is by design following the eventual consistency principle.

The second consideration against the microservice architecture is the processing latency. Decoupling of all individual functions to a single microservice is not feasible, as it processing time doubles with the number of services due to messaging queues and network latencies.

9.3.3. Lambda and Kappa architectures

As discussed in section 4, the distributed storage systems are bound by CAP, practically, limiting the storage policies either for the Availability or the Consistency. Manifesting either in the locking of the Read/Write operations or potential inconsistencies of the retried data due to the eventual consistency paradigms. The primary consideration against the eventually consistent systems is the need for Read-repair (which relates back to NoSQL schema on-read paradigm) in the form of the data collision prevention and version conflict resolution. A different approach was introduced by Marz [113], audaciously called "How to beat the CAP theorem," attempting to guarantee both the Availability with Consistency. In contrary to popular belief, the proposed architecture is not new [114] and was introduced in 1983 by Lampson [115].

The Marz's proposal is based on two assumptions. First, the data is immutable; thus any changes or updates to the existing data, generate new data. Secondly, the data is inherently time-based. This leads to the conclusion that the whole-dataset queries can be computed on the complete unprocessed dataset, albeit with at high computational latency, and that any subsequent changes in the datasets can be expressed as incremental updates of the query results. This holds true since changes are stored as new data (immutable) and occur after (time-based) data is present and thus appended later in time.

The approach is known as the Lambda architecture, due to the resemblance of the Greek Lambda symbol of the two-stream architecture 9.4. To facilitate the proposed dual-querying architecture, two separate systems are required. One system for handling of the large 'batch' datasets, responsible for computations of the "base" query results. And the second system for incrementally 'real-time' update queries, to accommodate for the newly ingested data. A limitation recognized by Marz is the reoccurrence of the eventual consistency in the increment updates, hence, a possible reintroduction of the divergent results due to data version conflicts. The solution, however, is deeply embedded in the methodology, requiring the batch system to perform base query recomputations regularly, overwriting the real-time layer results. This forcing the consistency of the historical data, with limited, inconsistent results found in the real-time data.

Marz argues that by recomputing the queries, the complexity of the storage (e.g. updates of shards, eventual consistency of the data) can be resolved. However, the complexity is re-introduced in the tooling as two distinct systems are required: large volume - high latency ('batch') and a low volume-low latency ('real-time'). Furthermore, it should be stressed that in its original form, the approach applies to the recurrent queries, hence, queries that are predictable and feasible to be computed beforehand.

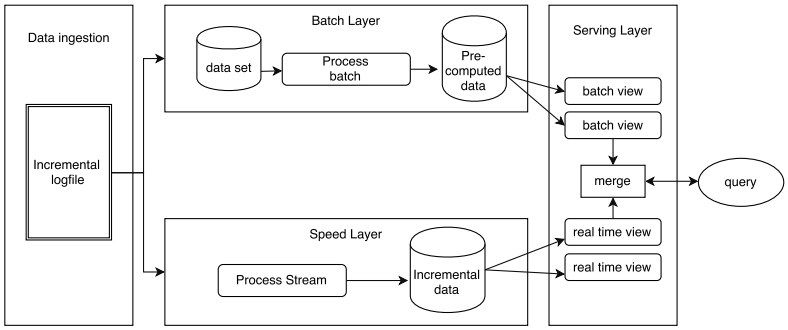

Figure 9.4: Typical Lambda Architecture

The popularity of the Lambda Architecture can be explained by two factors. First, at time of the introduction, Hadoop was the most popular ”BigData” framework. Hadoop natively utilizes HDFS storage system, that at time of introduction used MapReduce query system. MapReduce is excellent for the batch query types, allowing complete dataset traversal but comes at the cost of high computational latency. The second reason for the popularity is the presence of humans in the loop. With the increasing data volumes, and high computation costs and latencies, human-introduced errors are expensive. Marz [113] recognizes two types of human-errors: ”buggy queries” (either in streaming of batch layers), writes of bad data (batch) and general errors in the real-time layer. It can be argued that the former types of errors are interlinkable, as both buggy queries and good queries with ”bad” writes lead to incorrect data, thus incorrect results of the queries. However, both errors are addressed natively by the ”overwrite” nature of the batch layer, as buggy queries and bad writes can be resolved, and the previous (incorrect) real-time results are eventually overwritten.

It should be stressed that the data storage system is duplicated, with one system containing the raw, unprocessed data and two separate systems containing the query results as illustrated by the figure 9.4. The query results are stored in two separate systems, requiring an additional system to select the right data store and perform the querying. In a typical enterprise setting, the query retrieval system is likely combined with an API, obfuscating the querying logic from the end users. This is required since the system is likely utilized by a number of applications that should not be modified in case of datastore changes. In case of Delfi missions, the need for an API for the query results needed to be investigated but considered unnecessary for the proof of concept as data retrieval logic is encapsulated in the DAO of the application.

Another major drawback of the Lambda architecture is the duplication of the processing, which due to different requirements (throughput vs latency), leads to the use of two separate systems. This implicitly results in the duplication of the business logic and implementation in two processing languages (batch and stream). To address the problem, Summerbird [116] was introduced, generating MapReduce jobs usable in both Batch as Steam layers. At the time of writing, MapReduce has been surpassed by Apache TEZ [117] doubling the computation performance, hence rendering Summerbird obsolete for newly designed systems.

Additionally to the code base duplication, the resource waste should be considered. The results of the Stream layer are temporary and overwritten by the batch layer processing. This leads to the question, whether the changes in the immutable data are frequent enough and whether stand-alone stream processing can be utilized. To continue the Greek letter naming scheme, the stream-based processing architecture was introduced as Kappa architecture by Kreps [118]. The decision is not arbitrary since Kreps is one of the authors of Kafka, a system that dubbed as ”everything is a stream.”

The stream of data from the RA client applications can be seen as an append-only queue or a log. Each frame in the log is an update of the satellite parameters, thus the satellite state, and adds new information to the system. At any point in time, the database provides a snapshot of the satellite state, which is updated continuously with the information added from the log. Thus with a complete transaction log, one can reconstruct the state of the database for any point of time. This means that removal of the batch layer, not necessarily results in historical data being lost.

While both Lambda and Kappa are both heavily optimized for data processing and analytics, an analogy can be observed in Kappa and (micro) services example illustrated in figure 9.3. Applied for the ingestion-optimized system, it is clear that microservices are no match for purpose designed Stream processing frameworks, both in terms of scalability, but more importantly, Consistency.

The main consideration against Kappa architecture is the sensitivity towards out-of-sequence data as well as data duplication. Most notably both issues are automatically resolved in the Batch or data presentation layer in Lambda Archtiecture. The query or the data transformation is a function that receives data as input and returns data as output, preferably without external looks up. This is required for scalability, i.e. to facilitate the parallel processing. Query function is not aware of other query functions, the previous, the next or concurrent queries. Therefore, the query function cannot determine whether the given piece of data has been processed previously. The use of ”cannot” can be overruled, however, additional data retrieval in a query function will result in performance reduction of the querying system. It is important to note that the Stream queries, operates on a single data object, i.e. a single telemetry frame, while the Batch queries can span the entire (potentially in Terra-Petabyte range) dataset.

The Delfi missions that depend on Radio Amateur participation contain a high degree of data duplication, that should not be ignored in the analytics [GN-UI-RA-01]. The aggregation or the whole-data queries are historically the domain of batch processing and lead to complexity in the Stream layer.

In contrast to the data transformation point of view, i.e. Stream and Batch, Lin [114] argues that the Lambda and Kappa architectures should be considered as ”right tool for the job” and ”one size fits all” solutions respectively. Specifically, Lambda can be applied for the complex querying, that is potentially unavailable in the Streaming layer, hence ’the right tool for the job.’ Lin argues that the software development, in general, tend to (over)popularize one over the other in a repeated manner, comparable to a ”swing of a pendulum.” The ”right tool for the job” with increase specialization comes with an increased complexity and integration costs, while the ”one size fits all” attempt to simplify the system, thus achieve generalization by reducing of the tooling [114]. Lin recognizes abstractions as a plausible solution for the generalization/specialization issue, which comes at the cost of increased complexity of the system.

At the time of writing, the BigData tool landscape is expanding rapidly, and only a limited set of tools can be considered mature of the production. One of the industry proven tools is Apache Spark. As discussed in detail in section 10.1.4 Spark is designed for the Batch layer and can be utilized in Stream Layer (see section 10.1.3) with Spark Streaming; thus providing an abstraction for the (day-to-day) data processing.

The Spark capabilities hinted for a blueprint of a system with an active Streaming layer and infrequent batch updates. By following Clean Architecture system is allowing the Spark Streaming to be replaced with any other capable systems, for example, Storm. This need originates from the on-premise deployment constraint, and therefore, it should be investigated whether the Spark (batch) and Spark Streaming can reside on the same cluster, hence reducing the server requirements (number).

Table 9.1: Summary

	Microservices	Lambda	Kappa
Processing latency	+	+	+
Data size	?	+	+/–
Flow control	?	?	?
Data deduplication	–	+	?
Out of order data	?	+	+
Processing guarantees	–	+	+
Scalability and Elasticity	+	+	+
Integration and Extensibility	?	+	+/–
Hardware failure	+ / –	+	+/–
Fault tolerance	– –	+	-
Tracability	? / –	+	+
Machine Learning	– –	+	? / +

The content presented in the section is visually combined and applied to the assessment criteria in

table 9.1. The microservice architecture was concluded undesired due to increased risks regarding the system reliability. Furthermore, the research focused further on Lambda and Kappa architectures.

9.4. Stream Processing: Feasibility assessment

The section focuses on the applicability of Kappa architecture to the Delfi mission. The Kappa, begin a subset of Lambda architecture, requires Stream processing and limits the set of operations that can be performed to the dataset. As discussed in section 9.2.2, the data processing primary function is telemetry frame transformation into a set of telemetry parameters. With the use of mutually independent telemetry frames, it can be argued that the processing Stream processing is applicable without the need for batch processing.

9.4.1. Stream, Log and Table

The Stream, in the scope of stream processing, is defined as an unbound sequence of the messages: telemetry frames. Where the entirety of the received stream messages is the log. As a rule, the log is immutable and append-only.

Looking on the stream from another perspective, Helland [119] argues that a stream of changes and updates to the database generates a transaction log, with the instantaneous state of a database table merely a cache of all previous states changes. With a transaction log at hand, the state of the database can be computed for any given point in time. Thus the transaction log is the ground truth of the system.

The use of transactional log is deeply embedded in many database systems, for example, MySQL. As discussed in section 4.4.1, two-phase commit operates on the transactional log which is consumed by the database tables.

In the case of Kappa architecture, the Stream processing operates on the log and performs analytical operations directly on the sequence of data rather than using the cached version (= database table).

The second feature of Stream processing is the decoupling of the processing units, allowing multiple executors to operate on the log independently. This allows a high level of parallelism as executors require no synchronization.

The use of a single log of Telemetry processing for Delfi mission is advised. As any data operation would require re-processing of each individual frame in the log. Therefore, the stream has to be converted to another log. The second log would contain all processed frames in the sequence of processing. Further research is required to investigate the effects of parallel executors with asymmetric performance on the sequence of data in the log. In case a single executor is used, deduplication of ingested data may still have an effect and should be studied.

9.4.2. Stream processing native deduplication methods

The distributed network of RA and the use of SDR inevitably leads to out-of-order frames in the log. The problem is not unique for Delfi missions, and within the stream processing, one can distinguish between following (standardized) strategies: Session, Time-agnostic, Approximation, (fixed) Windowing by processing time and Windowing by the event time.

It should be noted that discussion assumes that the uniqueness of the telemetry frame can be assessed, for example, by the frame-counter or frame-time appended by the OBC of the satellite.

Time-agnostic

Time-agnosticism implies processing of data without taking the event or the processing time into consideration. Hence, ignoring the reception time by the RA, the frame OBC time and the time when the server system observed the event. This means that the frames are processed and aggregated based on the reception sequence. It should be noted that the approach is applicable only if the target system, handling the output of the stream processing system, handles the duplicate and out-of-order entries.

Fixed Window and Sliding Window

Within the framework of windowing, one can distinguish between two strategies: Fixed and Sliding. In fixed windowing, each log entry is contained in one segment and is therefore processed and retrieved once. The Sliding windowing retrieves log entries in the sequence of segments, as illustrated in the figure 9.5. The sliding windowing is defined by a length (number of entries) and period parameters.

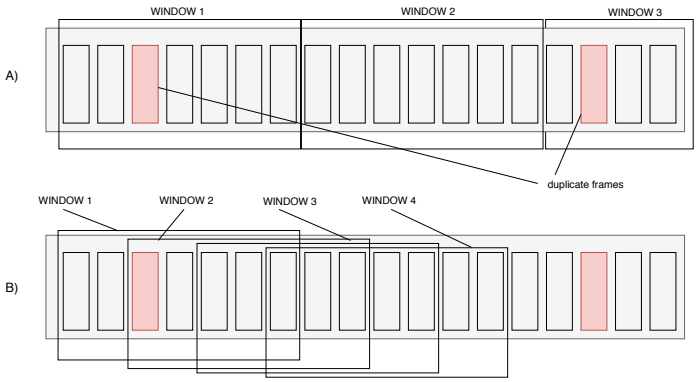

Figure 9.5: Fixed (A) and Sliding Windowing (B)

With the sliding window period larger than the processing time, the windows overlap, mimicking a sliding motion. The overlap depends on the ratio of the frequency of execution and the size of the window.

As illustrated in the figure 9.5, the out-of-sequence entries are not guaranteed to be contained in a single window and de-duplication may fail. Furthermore, the data duplication is likely to change over time, as it is to a higher degree dependent on the client application, the behaviour of RAs and other uncontrollable characteristics of the system. Leading to a need for the continuous tune of the windows.

It can be argued that to prevent de-duplication failure the information on the sequence of the messages has to be retained between the windows, requiring the system to store the state of each window. Due to parallelism, the windows are not guaranteed to be read sequentially or run on the same node. Therefore, syncing the information between nodes increases the complexity and introduces additional failure modes.

Windowing by Event and Processing time
The event time can be defined either as the satellite time or the reception time by the RA. As discussed in the chapter 3, the time of the reception by the client applications is an unreliable parameter due to errors in time and timezones. Furthermore, the onboard satellite clock is unreliable too due to clock drift. Additionally, the future missions may implement features to adjust or update the onboard clock, leading to inconsistencies in data sets.

In the case of Delfi, the processing time is defined as the reception time by the processing system. The reception time is affected by the client application, use of SDR or recovery from a network failure will cause out of sequence messages. Therefore, nor the Event, nor the Processing time Windowing provide reliable means of de-duplication.

Data de-duplication and sorting at the API
Since de-duplication and sorting are not feasible in the stored log, it can be argued that frames have to be verified before being added to the log. This will actively filter out duplicates and out-of-sequence frames. The approach is discarded due to two reasons. First, any bugs or malfunctions in the filtering will discard incoming frames. Secondly, the duplication and out-of-sequence data carry useful information on the RA behaviour.

9.4.3. Data de-duplication by sorting

The incoming telemetry messages are stored in the sequence of reception by the server system. Due to the immutable nature of the log, the order cannot be changed. Therefore, the sort and filter (Map Reduce) operations are not feasible.

An alternative is illustrated in figure 9.6. The frames/log messages are exported from the log, and with frame id extracted, exported to a mutable datastore, followed by sliding window sorting. In theory, after few passes allowing duplicate frames, i.e. frames with the same id, to be easily identified and removed. It should be noted that the frames stored in the log are not processed, thus requiring an additional transformation.

There is one important observation to be made, figure 9.6 shows characteristics of a batch pro-

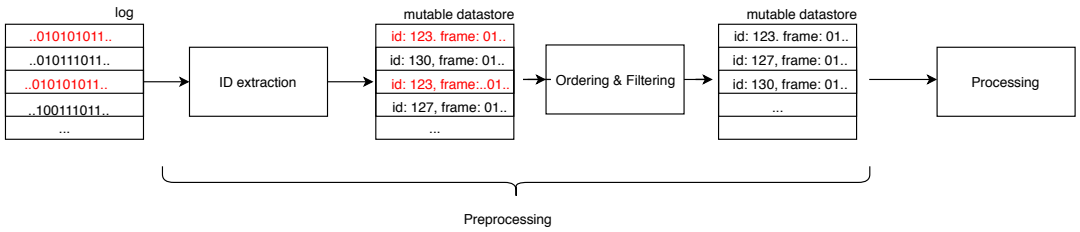

Figure 9.6: De-duplication in the processing

cessing: MapReduce, rather than a stream processing. First, a large set of data is extracted (=batch), the processed (id extraction) and transformed (ordered and de-duplicated). Secondly, the operation is performed on the complete dataset, rather than a small subset of data.

9.4.4. Data de-duplication in the delivery layer

As discussed in section 9.4.2, the de-duplication and out-of-sequence reordering are not feasible on the ingestion side of the application. The processing alternative, proposed in section 9.4.3 does not follow the streaming methodology and performs batch processing. Therefore, de-duplication on the delivery layer was considered. As a recap, the goal of telemetry processing software is to ingest, process, store and deliver data in TBD format to a mutable datastore (ElasticSearch in the proof of concept). Therefore, it is argued that by selecting a datastore that ignores or tolerates the duplicate values, and orders frames by frame_id, implying idempotency on the storage level.

Use of the delivery layer for de-duplication indicates two things. First, the delivery layer opposes requirements on the processing. Secondly, the core of the design methodology requires components to be easily exchangeable. This means that the processing system should not rely heavily on a feature of the delivery layer.

Furthermore, the de-duplication can be pushed further in the delivery layer into the logic of the application, precisely against the design methodology.

Conclusion

The data ingestion de-duplication is prone to mistakes that are unrecoverable. The analysis of the log-based deduplication showed that a large window size is required having an impact on the server memory (RAM) sizing. The processing de-duplication showed a class MapReduce functionality, thus hinting towards Batch processing. Lastly, the data delivery layer de-duplication was considered. Reliance on the data store or application logic is against clean architecture principle and should be avoided as the permanent solution.

9.5. Architecture discussion and requirements Assessment

In this chapter, the Kappa and Lambda architectures were studied. Application of Stream Processing to Defli-n3Xt data patterns showed weakness towards out-of-order and data de-duplication opportunities. De-duplication was found feasible with the use of an intermediate data store (topic), which when applied to Delfi context resembles MapReduce (Batch) operation rather than stream query. As shown in the table 9.2, on-demand querying is unfeasible as new data transformation has to be implemented. Secondly, the log is the primary data store. While offloading data to a different data store for back up porpuses is feasible, restoring the log from an external source is not straightforward. Furthermore, the Stream processing components operate on the incoming data and cannot consume an external data source.

In stark contrast to Kappa, Lambda Architecture provides significant higher applicability both in terms of on-demand querying (Pig, Hive) as well in use of the external data sources (HDFS, PouchDB and others). The use of incremental processing with Streaming and Batch layers ensures both high performance (near-real-time) as well as the data reliability (de-duplication, filtering and other transformations). The redundancy in terms of storage and processing decreases the efficiency of Lambda architecture but increases robustness significantly. The functional duplication allows for extensive experimentation and system development while in operations without the risk of data loss or processing

Table 9.2: Kappa architecture trade-off

Advantages	Disadvantages
Only one to two technologies required: Ingestion System and Stream Processing	Cannot be used for on-demand queries
One stream processing job is sufficient for one satellite mission	Application restart may require to reprocess all data in the log
The presentation layer has only one source	For aggregation transformations, data has to be serialized and written into the topic
No need for aggregations in presentation layer	Log corruption has critical consequences and loss of data processing
Potentially reusable in Lambda Architecture	Possible to backup, but hard to restore (timestamps lost)
	Not possible to use backup data store as the source of the processing
	Data migration (backup) requires additional processing job
	All processing jobs are incremental, application restart may require to reprocess all data in the log
	De-duplication not feasibile: may lead to unpredictable results
	Less mature than Batch processing
	The result of a stream processing is either right or wrong; results are not corrected
	Processing reacts on the incoming data (event), rather than the query (request)
	Data schema and processing changes are abrupt

errors. The results of Lambda Architectures are combined in table 9.3.

Table 9.3: Lambda architecture trade-off

Advantages	Disadvantages
Very versatile and scalable (up to petabyte datasets)	At minimum 3 technologies
Tolerance for human errors (bugs, operational mistakes, data loss)	Originally designed for MapReduce, but can be changed for other systems
Any types of queries possible	Inefficiencies in storage and processing
Running computationally intensive queries/jobs, with long execution time, will not affect other processing jobs (live processing is separated, application data queries are separated)	On-demand queries on small dataset can be slow. Data presentation layer should be used for the application queries.
Faulty (ad-hoc) queries are unlikely to halt the system.	In some cases the transformation logic is implemented twice (queries: speed/batch), each has own different code-base
Append-only datastore ensures that all changes (where, when, what) are encapsulated.	Stream layer processed data is overwritten
Schema migrations are simple as data is overwritten incrementally	Requires additional logic to deal with the dual data source in the presentation layer
Database system changes are simpler, as data is overwritten incrementally	
The RAW data is stored immutably. Should never be modified (append-only). Provides backup	
Processing can operate on backup data directly or any other data source	
Applies extremely well for the data de-duplication use-cases	
Works extremely well for analytical operations (number of frames per RA)	
Batch processing can be used for Machine Learning: model training.	
Machine Learning models can be applied in stream processing	
More mature than Kappa	
Jobs can be added and removed ad-hoc	
Easy to migrate and test new technologies	
Redundancy in processing and data. Ideal for novice users	
Can exchange components without downtime.	

III

Results

10

Proposed Architecture

The discussion presented in chapter 9.3 identified Lambda Architecture as the optimal solution for the Delfi future missions.

The chapter will focuses on the selection of the components and tools required to implement the system. Section 10.1 will address the major components of the generic Lambda architecture illustrated in the figure 9.4, discussing the functionality, requirements and selected technology for each component. The used technologies are further discussed in section 10.2 leading to the summary of the interfaces and implementation discussion in section .

10.1. Processing system Components

Lambda Architecture proposed in section 10, consists of four distinct components aiming to solve the following concerns: data ingestion and storage, real-time processing, error-recovery (de-duplication) processing and data delivery.

As discussed in section 2 the technologies used to solve the needs are exchangeable and preferably independent. The focus will consequently be placed on ”how” technology solves a particular need, rather than the trade-off of the available technologies.

The PTS requirements and correlating needs derived throughout the document and MSc thesis have been summarized in the table 10.1 and will be used in the following sections.

10.1.1. PTS global requirements

The initial project iteration presented in chapter 4 identified distributed NoSQL data storage as the optimum trade-off between reliability and complexity. The need for a distributed storage system led to PouchCouch system presented in chapter 8.
The PouchCouch PoC showed that distributed storage is feasible and applicable for Delfi type of operation given the that the system is self-governing, requiring no manual operations.

The study of legacy systems showed recurring errors in the processing definitions or code itself. In the section 5 Kaitai framework was identified as the most applicable telemetry parsing and decoding system, allowing unified telemetry definition and decoding across different platforms and programming languages.

The complexity of the legacy system directly correlates to reduce reusability, which led to the spiral of new PTS for each Delfi mission. Furthermore, due to the complexity of PTS, the system cannot be implemented within the MSc Thesis allocated seven month period.

Globally, this leads to a global need for the system to be reliable and robust. Reliability can be assured with duplication of components, allowing single component failures. In theory, this allows for components to be added and updated, without losing processing and storage capabilities. The use of independent components, as guided by the Clean Architecture, allows independent component development, allowing for incremental system design.

By the use of distributed systems, scalability can be achieved with lesser effort, allowing for the system to increase capabilities and performance when required.

The major lesson learnt from the legacy and the PouchCoudh PoC is the use of contingency. The use of new technologies increases the risk of errors in configuration and bugs in the code. The subsystems should, therefore, be able to operate independently in case of a failure and provide the minimum viable functionality at all times.

10.1.2. Ingestion System

In the proposed architecture, the API component is placed outside the Ingestion system. The API, in this case, is seen as a gateway and is responsible for the clients and client applications authentication, hosts blacklisting, and other functions that are not part of Telemetry processing. The API should not store the data, but pass it through the ingestion system

The ingestion system is responsible for data aggregation. For example, from different API services (API per user group Operations, RA users), API systems can be deployed on multiple machines; furthermore, the ingestion system could be connected to a test setup generating telemetry frames for the full system testing. With the ability to store and process data at a high rate, the architecture can be used as a data acquisition system for ground-based experimentation. This allows satellite-generated data to be used along with experimental data for example for satellite malfunction traceability [9].

Need

The need is summarised to:

1. Ingest data at a high rate from multiple data sources (parallelism)
2. Support for different data sources and programming languages
3. Create a buffer for the processing system.
4. Store data temporary, in case processing or storage system fails
5. High Availability systems are preferred
6. Should be able to scale (increased data rates possible)

Messaging and event log systems

At the time of writing, two categories can be defined: messaging and event-log based systems.
The messaging systems at its core consist of a queue where the messages are appended to by the producers. The system either pushed the messages to the connected clients (consumers) or allows consumers to pull data at own rate. Once pushed or pulled from the queue, the message is deleted.

The core of an event log system is the log, an append-only and immutable data store. In contrast to messaging systems, the messages remain unchanged in the log after being read by the consumers. To keep track of the read messages, event log systems use a reference pointer (offset) for each consumer. By working with offsets, the system not only guarantees that message is read (and processed) but ensures that each record is read and processed exactly once.

Selected technology: Kafka

Confluent Kafka was selected based on the best fit for the requirements, technology maturity, available documentation and ease of use. Kafka is discussed further in section 10.2.1, with the highlights summarized below.

The use of log, instead of a queue, provides message processing guarantees. The offset is updated only after the message is retrieved from the log, processed and stored. The process ensures that each telemetry frame is processed and stored only once.

Secondly, log allows multiple consumers, allowing Stream Processing jobs and Batch (in this case permanent storage) to consume the messages at their rates. As expected, the message consumption rate of processing is lower than of the batch sink storage.

Lastly, with an immutable log, any critical failures of permanent storage, or delivery layer issues are recoverable. The incoming will be appended and stored in the log regardless of the status of other

systems. Furthermore, any lost data can be recovered from the log, reprocessed and placed back in the delivery layer or even in the permanent storage.

From the non-technical point of view, both Kafka and Confluent Kafka are well maintained, delivering three minor releases a year. The popularity of Kafka in the enterprise environment hints towards high quality (bugs) and, theoretically, long-term support.

Furthermore, with expanding the scope of Kafka framework, utilization of the product allows further prototyping with stream processing, which might in future provide better results than the micro-batch based Spark Streaming.

The test of Kafka in production as will be discussed in chapter 11 proved Kafka's reliability but showed that intermittent hardware failures of more than the quorum might lead to data corruptions. This is perfectly acceptable since all incoming data exported from Kafka into HDFS data store as discussed later in this section. Furthermore, hardware and node failures are less likely on server-grade hardware than Raspberry Pi's.

10.1.3. Stream Layer

The Stream Layer is responsible for the near-real-time telemetry frame processing. As discussed in section 5, Kaitai framework is selected as telemetry parser and decoder. The functionality of Kaitai is limited to parsing of the data, and additional functionality is required to retrieve data from the Ingestion Layer and store the results in the Presentation Layer.

The Stream Layer should be able to operate on one single machine for development and testing while being able to scale to a large cluster without code adjustments. This requires abstraction of the processing.

The developed routines (jobs) should be modifiable, ensuring that errors and bugs are resolvable. Ideally, the system should facilitate multiple simultaneous jobs, allowing each satellite or mission telemetry processing to be independent. Furthermore, parallel jobs should not pose requirements to each other and should be able to terminate or start independently.

Need

The primary needs for Stream Processing are summarized below:

1. Able to run on a cluster and on one single node system
2. Able to utilize resources of multiple machines
3. Able to tolerate and recover from processing node failures
4. Able to add new processing routines (jobs) without downtime.
5. Support one of the Kaitai supported languages
6. Provide Job monitoring and administration interface
7. Be well documented

Selected technology: Apache Spark

One of the significant disadvantages of Lambda architecture is the complexity, hence the use of different technologies requires expertise in various applications. With processing separated in Stream and Batch processing, the business logic is split in two, thus twice as expensive in development, maintenance and testing.

The use of Spark Framework allows the use of the same processing code in both systems, hence cutting the complexity in half. The micro-batch processing approach is also operationally tested in the Delfi-n3Xt server application, that relies on fixed interval running processing jobs. With Apache Spark used in the batch processing, the use of Apache Spark Streaming allows re-use of business logic: telemetry processing. This minimizes the amount of programming effort and allows better testing for software defects.

The Spark Streaming is best perceived as micro-batch processing, treating the incoming stream as micro-batches: small sets of data. The micro-batches are executed at fixed time intervals, for example, every second. The process of data filtering and transformation performed by the micro batches is called Spark Job. Spark Jobs contain a programming interface API for a number of programming languages,

Table 10.1: System requirements summary

	Tolerate Uncertainty
GN-PR-09	Inconsistencies in data should be resolvable (i.e. Stream errors fixed in Batch)
GN-PR-08, GN-PR-21	Ability to add new functionality while in operation
GN-PR-08, GN-PR-21	Ability to add/remove components while in operation (load data from MySQL or other sources)
GN-PR-01	Ability to use any data source for processing (backup data)
GN-QA-MAIN-04, GN-QA-MAIN-05	Ability to export data or migrate to another implementation if needed
	Ensure Reliability
GN-PR-01, GN-QA-MAIN-01	Do not rely on a single framework/system
GN-SAFE-03	Error recovery (node failures, network failures)
GN-SAFE-01	Data loss recovery (disk failures)
GN-QA-REL-02, GN-QA-REL-03, A	Designed with subsystem/node/server failure in mind
GN-ING-15	Back pressure should only affect data ingestion system
GN-QA-MAIN-02	Proven and further developed technologies
GN-PR-09	Reliable results (de-duplication, adjustments of clock drifts)
GN-QA-REL-04	Ability to test the system (replay data, each component is testable)
GN-AVAIL-3	The more parallel machines, the lower the risk of the complete system failure
	Ensure Robustness
	Errors tolerance
	– Human error tolerance
GN-PR-8, GN-SAFE-2, GN-PR-21	Allow processing jobs errors to be resolved
GN-SAFE-2, GN-PR-25, GN-PR-26	Allow incorrect data to be overwritten
GN-AVAIL-3, GN-QA-AVAIL-01	Ensure that subsystem adjustments are not disruptive for the system
GN-AVAIL-3, GN-QA-AVAIL-01, GN-PR-1, GN-QA-REL-03	Tolerate errors in configuration: single subsystem should not terminate the entire system
GN-SAFE-1, GN-ING-16, GN-DS-01, GN-QA-AVAIL-01	Prevent data loss (human mistakes)
GN-QA-AVAIL-03, GN-SAFE-03	Auto-recover whenever possible (no 24/7 support)
GN-AUX-DOC-1	Documentation should be sufficient for further development
GN-PR-12-rev1	Re-usable business logic is necessary
GN-QA-MAIN-3	Well-structured programs (abstractions) are preferred
GN-QA-MAIN-3	Comprehendible system design
GN-QA-MAIN-3	A more functionality with less code
GN-QA-MAIN-3	A ability to run processing locally (Acceptance tests before deployment, debugging)
	– Hardware failure tolerance
GN-SAFE-03, GN-QA-AVAIL-01, GN-QA-REL-03	tolerance to server(s) loss
GN-SAFE-03	tolerance to disk loss
GN-SAFE-03, GN-QA-AVAIL-01	tolerance to network loss
A	– Use software to prevent errors (IDE)
A	– The easier the tool to use, the less error-prone
A	– Use widely known programming languages
GN-QA-MAIN-3	Does not require in-depth system knowledge to operate
GN-QA-MAIN-3	Can scale up and out to increase performance
	Ensure reusability
GN-QA-MAIN-3	Does not require in-depth knowledge to implement new jobs
A	One way dependencies (limit scope of modifications when changing components)
GN-PERF-12-rev1	Re-use existing components (business logic)
GN-QA-MAIN-2	Community supported projects preferred (re-use of code, free support)
GN-QA-REL-04	Allow parallel development (add functionality without removing old ones)
GN-QA-MAIN-3	Ability to pre-test and debug
GN-QA-MAIN-3	Provide an overview of the running tasks, components, system status
GN-PR-1, GN-QA-MAIN-4, GN-QA-MAIN-5	System components are decoupled and replaceable. I.e. use more applicable or better-understood technologies
B	Advanced use cases: machine learning
C	Tasks, components implementable in less than seven months
	Ensure Scalability
D	Easily scaled up and out, when needed
D	Redeployment possibility to hosted cloud systems
D	All components should be designed with scalability in mind
D	The performance increase is predictable (i.e. linear scaling)
D	No code changes required for scaling, only configurations (operations)

such as Scala, Java and Python. Within the framework of the project, Python has been selected due to direct compatibility with Kaitai decoding framework.

The processing can be performed on a single node, or distributed on a cluster, where the allocation of resources and scheduling are being continuously monitored by Yarn and Spark. Failure of cluster nodes is automatically recovered, with minimal effects on end-user experience.

From the technology and Spark framework point of view, use of Spark in Batch and Stream allows utilization of Spark MLlib libraries allowing, machine learning in Batch layer (for example kNN for classification and outlier detection) and application for the trained model in the Streaming Layer allowing instant results.

10.1.4. Batch Layer

The Batch Layer has the purpose of bulk data processing and retention of the immutable data set.

Permanent Storage

While log-based storage in the ingestion layer is feasible, a dedicated permanent storage system was added to the architecture. This ensures that data is stored reliably and can be accessed via other systems than stream based processors. Secondly, with the use of dedicated storage, the ingestion layer can be exchanged for a different technology without a need for data migration.

The discussion and trade-off for storage technologies can be found in section 10.2.2 and lead to the selection of the HDFS.

Batch processing

As discussed in section 9.4.2 telemetry frame de-duplication is not feasible in Stream Processing and requires batch processing to remove data duplicates and sort out-of-sequence entries.

The batch processing is responsible for :

1. Load data from HDFS structure
2. Clean-up HDFS structure
3. Process data

TThe data is exported periodically from Kafka to HDFS storage. The frequency depends on the level of trust placed on ingestion layer and may vary from milliseconds to hours. For PoC 15 minute interval was selected. The process is performed as a recurrent job, storing and structuring data in HDFS folders based on mission-satellite-year-month-day parameters.

The batch job is run every 24 hours consuming data stored in HDFS (since the last the run) combining to a single per-day file. This is required since the storage of small files is ineffective in HDFS, directly affecting HDFS performance.

The second batch processing job consumes all per-day HDFS files, extracting and decoding the binary telemetry frames with Kaitai framework, removing duplicates and storing them in the presentation Layer (ELK).

Apache Spark was selected due to versatility in the processing, allowing the use of Python and Kaitai framework without modifications. Spark is scalable from one node to a cluster, without the need of code changes due to abstractions of the API. Spark Framework furthermore provides native support for HDFS, and with use of Structured Streaming, the Pandas like syntax can be used for the analytics queries.

10.1.5. Presentation Layer

The presentation layer consists of the data store where the processed data is written to. Following Clean Architecture principle, the system should be replaceable, and with Layer out of the scope of the proejct, no trade-off or selection process will be performed. These decisions stem from the fact that the data store should be best suited for data use. For example, a web application queries and data access patterns may prefer the use of SQL (random read queries) over NoSQL systems.

It is assumed that NoSQL provides the best fit for the problem and ElasticSearch was utilized for the proof of concept. This was primarily driven by Kibana UI compatibility, allowing visual data representation and easy debugging for the PoC testing.

10.2. Used Technologies

This section provides an overview of used technologies.

10.2.1. Kafka

A popular solution for the problem is Apache Kafka. Kafka is a distributed messaging system for log processing that provides an interface for pushing messages to and an interface to pull messages from at arbitrary rates. Kafka is both scalable vertically, as RAM and Storage Speeds will increase the performance, and horizontally, as incoming messages are split accross multiple brokers in a linear fashion.

In practice, Kafka can be seen as data streaming pipeline, connecting various real-time data producers and consumers, allowing connectivity abstractions. Due to the streaming nature, Apache Kafka is heavily utilized in the streaming archtectures, and is some cases deployed as a complete processing system.

Kafka Architecture and Components

Illustrated by the figure 10.1, Apache Kafka consists of three major components, Kafka Core API, Kafka Connect API and Kafka Stream API. The Core API provides hooks for custom Producers and Consumers, used for data ingestion and retrival respectively. Apache Connect API provides open-source, ready to use connectors (Producers and Consumers) for data ingestion and retrieval for Kafka Core. The Connect API can be utilized to connect database systems for data transfer and migration. The final component, the Streaming API provides an API for data transformation, and follows the logic and principles explained in section 9.4.

Originally designed by LinkedIn, Apache Kafka is released as an open source project and is developed by the open-source community. Additionally, lead by the original Kafka designers, Confluent provides an enterprise-grade platform built around Apache Kafka. The Confluent platforms contains the Kafka Core, Connect and Stream API, denoted as Kafka, and Kafka Stream Registry and Kafka Rest API as illustrated in the figure 10.2.

The Kafka Stream Registry provides a service for AVRO schema registration, allowing schema to be stored in a consistent and immutable manner. In this deployment form, the AVRO schema defintion is omitted from the message, reducing the message size. The dedicated service can be used by the Consumer API to request AVRO schema's at will, but not required to do so.
While the Consumer and Producers utilizing the Core API are primarily developed in Java, the Kafka REST API allows use of alternative programming languages with use of REST services. Additional component, not part of the Confluent ecosystem is Landoop UI. This optional component consumes the REST services and provides an user interface for Kafka monitoring, such as overview of topics, data in the topics, and configuration interface for the connectors and more.

Basic Operation

This section provides a high-level overview of Kafka Core operation and illustrates the use of Consumer and Producer API.

Following the concept introduced in section 9.4, Kafka acts as a ledger, storing the sequence of messages in an immutable fashion, required to ensure the data validity for the stream processing. The state of the system is updated by every ingested message. Where the database is the "cache" of the current state, in case of a streaming system, the state is expressed in the entireness of the stream. The complete (immutable) dataset is therefore considered the ground truth of the system.
Kafka message is the means of data trasfer between the source (producer) via the Kafka broke to the target (consumer).

Topics, Replicas and Offsets

Similar to the database systems, data is separated. In case of Kafka, the separation is done on the level of streams, each defined as a topic. The system scalability is achieved in a manner similar to database sharding, by splitting the topics into partitions, that are distributing across the cluster as illustrated in the figure 10.3.
To ensure availability, the partitions are replicated. When multiple replicas of a single partition are present, one will be elected "leader" and will receive, store and send messages for the partition. The in-sync replicas (ISR), will update the messages in the log and will be on standby if the leader will lose

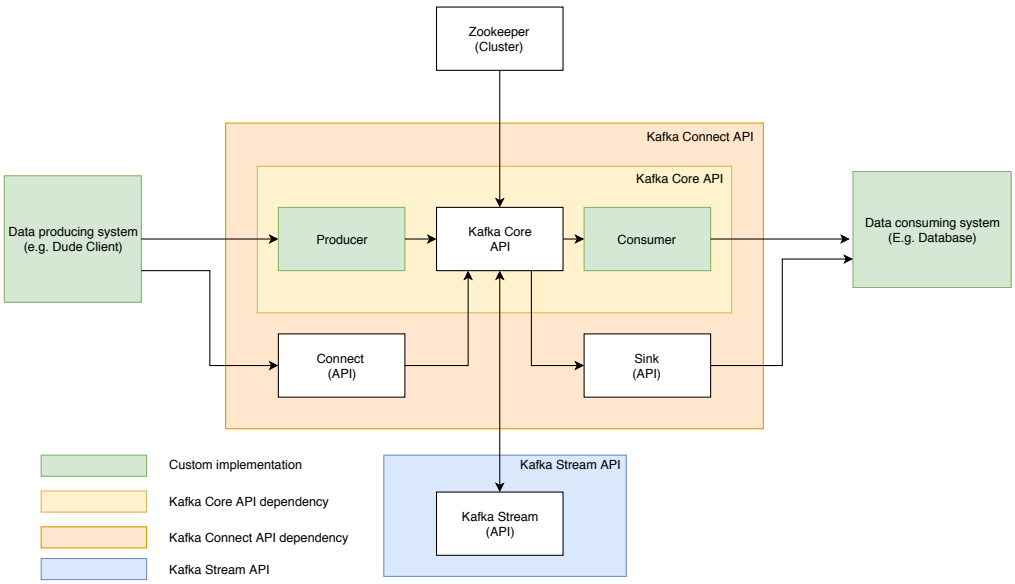

Figure 10.1: Apache Kafka Architecture

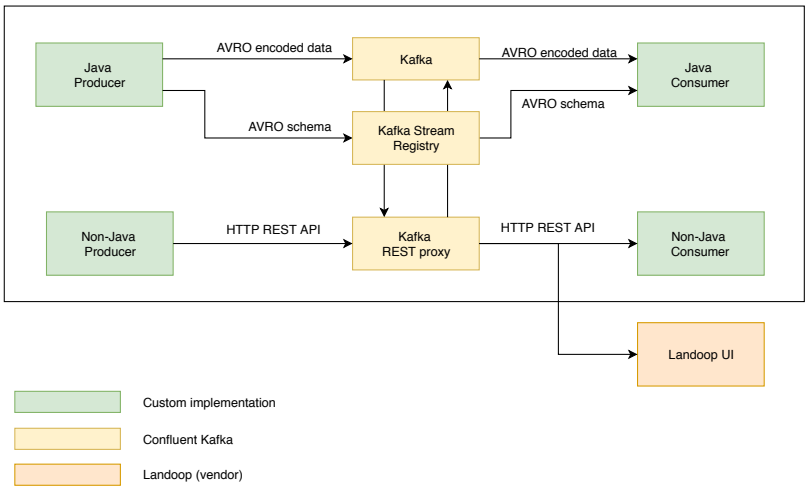

Figure 10.2: Confluent Kafka Architecture

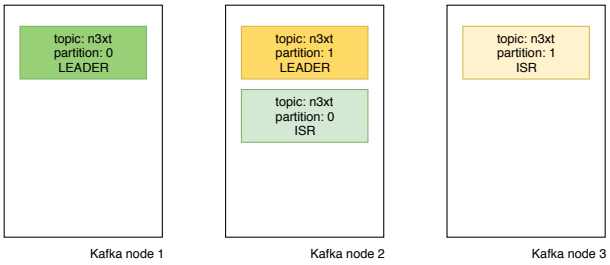

Figure 10.3: Kafka topic partitioning

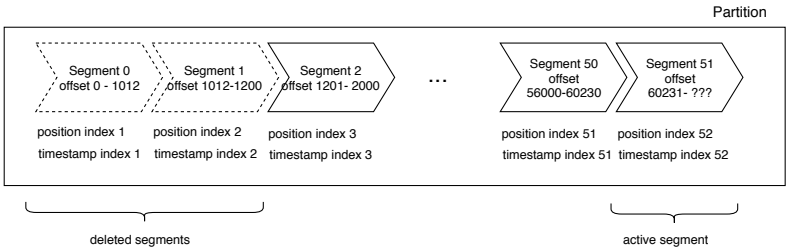

Figure 10.4: Kafka Segments and Indexes

the connection. With a replication factor of N up to N-1 Kafka nodes are allowed to fail simultaneously without data loss or downtime. Logically, the replication factor cannot exceed the number of nodes of the cluster.

The partition leader allocation is done by the Zookeeper. Zookeeper is responsible for configuration management, group id and offsets tracking and many other features of Kafka. Zookeeper is deployed as a standalone cluster, with a single elected Master node that keeps track of client nodes via a heartbeat mechanism. Internally, Zookeeper provides a hierarchical namespace functionality, consisting of nodes, called znodes, that are most comparable to files and folders in a filesystem [120]. When the heartbeat is lost, a new znode branch is created of all available cluster nodes, each identified by a sequence number. The elected leader is the node with the smallest number [121].

Segments and Indexes

Kafka topics consist of partitions, which are either unique or replicated. Similarly to the messages that are identified by a message-offsets in a topic, each partition consists of segments containing the data and indexes linking messages to the segments. The incoming messages are appended to the active segment of the partition. New segment is created when previosly active segment exceeds a threshold in size or in time (by default one week). As illustrated in the figure 10.4, the segment indexes are defined in incremental digits as timestamp counters. The indexes are used internally by Kafka for data cleaning up, as each message is configured with an expiration date.

Because the segments are stored as files in the file system, the segment size may require fine-tuning. This is the consequence of the number of open files setting of the operating system. Having a large number of files degrade the system performance, as clean-up is run on the per-segment level. Furthermore, OS will terminate the program if the number of open files threshold is exceeded. Therefore testing is required to determine sufficiently large segments, but not too large, so the seek time of the messages is not affected.

The segments are stored directy on the file system. Since segment reads are sequential, it facilitates high read speeds. It should be ntoed that Kafka relies on the OS file system caching. In practice, this results in the frequently read segments to be copied to memory. [122] In case of Kafka failure, data readily resides on the disk, reducing the chance of corruption.

Clean-up policy

The clean-up daemon is triggered every 15 seconds and can start message clean up if the message is expired (retention time) or when partition volume exceeds a preset threshold. The default policy dictates that the messages are deleted from the topic after 168hours (one week).

Alternatively, the messages can be retained indefinitely, but compacted to save the disk space. The

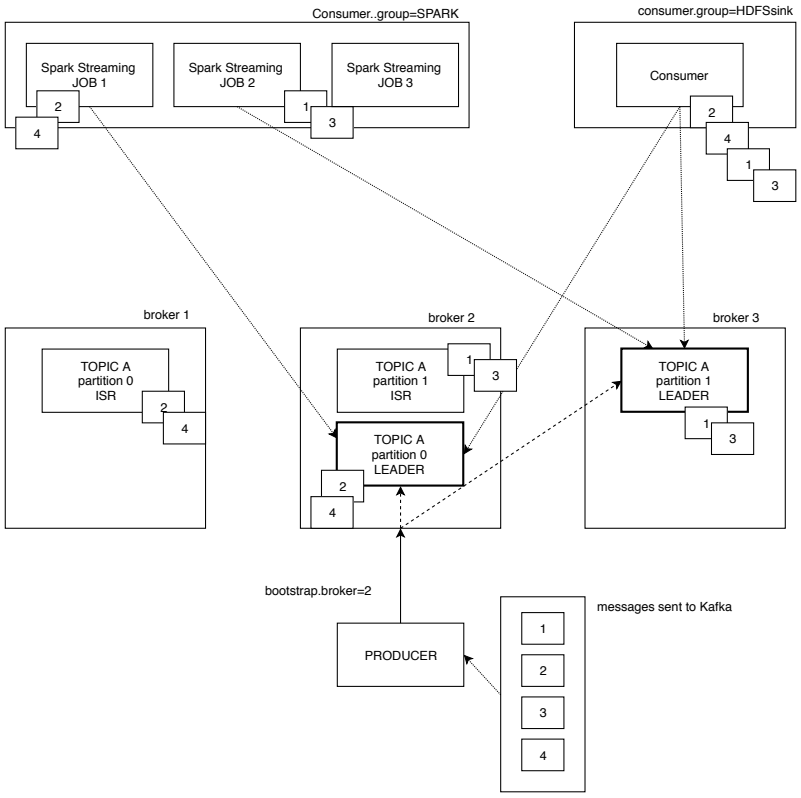

Figure 10.5: Producer and Consumer Groups

compaction policy does not compress the data, but deletes the older messages sharing the same key. This relates to the theory covered in section 9.4, where the state of the system is presented by the updates of the log. Hence, by keeping the last version of each key a database-like snapshot is created.

Kafka Producer
The key-less incoming messages of a topic are randomly assigned to a partition, this is required to ensure the load balancing and to spread the load across the cluster. The producer can opt to submit messages without acknowledgments (ACK=0), hence not knowing whether the data was delivered. To prevent data loss, ACK=1 can be configured requiring the connected broker to acknowledge message reception. In extreme cases, ACK=all can be configured, requiring acknowledgement from the leader and all replica partitions. Next, to the topic of separation, Kafka facilitates key-based separation, allowing messages sharing the same key to be placed in the same partitions. This can be utilized to increase data retrieval performance, as data can be retrieved per partition.

Kafka Consumer
The messages are retrieved from the topics as a pull request, on demand by the data consumer. The data consumers are typically aggregated into groups, called consumer groups. As illustrated by the figure 10.5, within a group, one consumer can connect to many partitions, but a single partition can only be connected by a single consumer from the group.

The messages within a partition are identified by an offset. The offsets are guaranteed within a partition, but not across the topic. The consumer offsets, the last read messages, are stored in Kafka in consumer offset topic. When a consumer group is a user, a consumer group offset will be defined. The consumer group offset is synchronized across the cluster and prevents previously consumed messages to be accessed by the consumers of the group. This allows consumers within a group to operate independently, as Kafka will ensure that only previously unprocessed messages will be available.

Kafka Delivery Semantics
The data duplication can occur due to duplications on the client side, for example, due to use of SDR

with DuDe client. Another source of the duplication and data loss are the nodes failures of the cluster. With use of 'at most once' strategy, under normal operations, a message pulled from the Kafka topic causes offset update of the partition, whereafter the message is processed by the consumer application. In case of a node failure occurred after the message retrieval, the message is lost as the offset is already updated. To resolve the data loss problem 'at least once' strategy can be utilized. This requires the consumer to acknowledge the message reception and processing prior to the offset update. However, this causes a new problem, as a failure between message retrieval and offset updates will cause the message to be processed twice, leading to duplicates. The solution to the problem is idempotent processing, meaning that processing of the same data twice should not cause duplicate results. This is achieved trivially by verifying whether the processed data exist in the data store prior to storage.

Message Key

As discussed in the previous section, each message sent and consume from Kafka may contain a key. The key is optional and is represented by a String object. Unless Kafka is repartitioned, all messages sharing the same key will be placed in the same partition. Furthermore, as illustrated in the figure 10.5, Kafka consumers pull messages per partition, thus reading the messages per key in the sequence of the reception. Additionally, consumers can be configured to pull data from specific partitions.

When working with multi-satellite data, the key can be set as satellite identifier, thus separating satellite per partition. This would result in each Spark processing job (executor) pulling and processing per satellite, in the sequence of reception. This requires fine balancing to ensure that each partition is equally utilized. The processing job, pulling Kafka messages, should, furthermore, be satellite-agnostic and be able to process all telemetry frame. This approach is applicable for twin (or more) satellite constellations, with an identical set of telemetry frame definitions.

When working with multiple missions, requiring different processing logic and telemetry definitions, multiple topics can be used. This provides separation, as each Spark job can subscribe to a specific topic, thus, the specific mission job is designed for.

When working with a single satellite mission, one can distinguish between three alternatives. The first alternative is key-less. When no key is present, the messages will be placed in partitions randomly. This works for time-agnostic processing and will fail for windowing. Furthermore, when messages are dumped into HDFS, the sink will consume partition per partition, resulting in a quasi-random sequence of the messages.

The second alternative is an RA or GS specific identifier. This will group messages (telemetry frames) per receiver, allowing simpler grouping and filtering, but more importantly more performant RA statistic operations. For example, frame counter per RA can be performed on a single partition, hence by a single job, without inter-node aggregations. Perfect in theory, this approach will fail in practice, as the frame count per RA is vastly different.

The third alternative and the proposed solution for the single or multi-satellite missions is keying per receiver per day. This ensures that the entire reception sequence is contained within one partition allowing windowing while providing enough balancing between the partitions.

Deployment Kafka

The deployment of Kafka requires at minimum an assessment of the number of required nodes (brokers), number of topics, partitions and replication factor (replica partitions).

The replication factor (RF) increases the disk space use and reduces the performance as more data has to be copied. The RF can be set to 2 (minimum), allowing one broker to fail, or 3 allowing 2 brokers failures. Depending on the failover policy, failure of a leader in RF 2 deployment may lead to corruption of offsets of both topics. This lead to the decision of replication factor 3.

The number of partitions determines the scalability (parallelism) of the consumer groups. With 2 partitions, only two parallel spark processing jobs can be executed. On the other side of the spectrum, a high number of partitions leads to increase in the number of open files that degrade the overall performance of the server. Furthermore, by doubling the number of partitions, the latency increases significantly as the amount of data to be replicated is increased six times. Further operational testing is required, however, at minimum 3 partitions are needed to accommodate three nodes Spark Cluster. It should be noted that the number of partitions can be changed at any time, however, requires topic downtime. Topic repartitioning additionally resets the key-to-partition allocation.

The number of Kafka nodes and partitions is furthermore right to the data ingestion rate. By increasing the number of nodes and partitions, higher level of parallelism can be achieved both on

Producer as Consumer sides. Due to moderate to low data ingestion rates observed in Delfi-C3 and Delfi-n3Xt missions, two to three node Kafka cluster is sufficient but requires operation testing for the validation.

10.2.2. Immutable Storage

To facilitate human-error tolerance, satellite data should be stored in the original and unmodified form, permanently. This ensures data can be reconstructed, as a means for the recovery for processing errors or critical failure of the delivery layer storage medium. The Permanent Storage ought to be append-only and should not be used for data modification, for example, the timestamps adjustment for clock sync of the client. The Permanent storage is read infrequently by the Batch processing (whole-data operations) or for Kafka replays, for example for the streaming tests. The systems are evaluated based on the system complexity, storage guarantees (redundancy) and the flexibility.

Kafka based storage

The use of Kafka in the data ingestion layer provides a short-trem data retention. Meant as a buffer, topics can be modified for long term storage by changing the data retention period.
At time of writing, no explicit longterm support plan for Apache Kafka has been proposed [122]. The project follows a time based release methodology and aims at three releases a year, guarantueeing rolling upgrade (upgrade without system downtime) for releases up to on year in the past. Each updates includes new features and require system updates to faciliate newer consumers and producers.
In terms of technology, Kafka provides sufficent means for long-term storage. Data is partitioned and replicated, which reduces the risk of dataloss. Due to OS based caching, the increase in stored data is unlikely to affect performance.
The long term storage in Kafka is heavily advertised by Kreps (Confluent vendor) [123], however, it has one large (and hidden) limitation. Relying on OS caching, read data will be placed in memory. Therefore, accessing a large set of old data in a topic, will result in large memory use, as OS at runtime cannot know the life expectation of the read dataset. The performance of the topic read operations will degrade significantly once memory limit is reached. Therefore, the whole-data operations, such as Batch jobs will become more expensive in terms of memory over time.

Database storage

As an alternative for Kafka storage the Database, File System and Object Storage were considered. The databases, are the most complex systems from the list. The use of SQL system, requires structured data and schema defintion. Serving multiple spacecrafts and missions, the use of SQL systems requires additional data management system to redirect messages to the tables in accordance to the spacecraft, mission, or other parameters.
The NoSQL systems with schema on read behaviour are more suited for the application, but increase the complexity by use of additional features. For example the in-memory caching is not required for infrequent data reads but present in the majority of the systems. As discussed in section 4.4.2 achieving redundancy and scalability of database systems in terms of disk space, is not trivial. While the performance of the write operations is not critical, the read performance is. Used for recover from delivery layer failure, the system should be capable of replaying the recent ingested data at a high speed. For example by pushing data back to the Kafka broker, triggering the Stream processing or by running a Batch job with a reduced scope.
Maintaining a low footprint for write operations is preferred as a measure to reduce the operation cost. In contrast, to ensure a high read throughput, the NoSQL system either keep reference to the data or pre-cache in memory, which is unfavorable.

Object storage

An alternative for the Dabase systems is the Object storage. As discussed in section 4.3, the Object storage requires a unique identifier for data retrieval. Since frames are accessed in bulk, and not necessarily in one-per-one fashion, the use of object storage increases the complexity for data storage and retrieval.

File system storage

The final considered alternative is the File System storage. To facilitate redundancy, only distributed

system were considered. This leads to the hardware solutions such as SAN, discussed in section 4.2 and purpose designed software applications such as NAS systems, HDFS, GlusterFS, Google Filesystem and many others. The SAN availability for the project requires further investigation, however, is generally associated with an increased cost of the hardware. The NAS solutions are considered standalone hardware, however, in practice, NAS is simply a server running a file system.

For the purpose of proof-of-concept, HDFS file system was selected. This is driven by the best fit to the flexibility, both in terms of deployment hardware or OS, as well the storage, i.e. files, directories, as the database file, python file, binary blob, parquet, AVRO etc... Storage guarantee, as the use of distributed store, replication and fault-tolerating design methodology ensure data protection.

In terms of processing, HDFS has a low write-operation footprint and high read-operation throughput at higher latency. Simply put, HDFS requires time to prepare read operation, but once prepared, it is done at a high rate. Additionally, HDFS provides high compatibility to Spark ecosystem, allowing data to be loaded and processed in memory on the same node increasing the Batch processing speed and allowing parallel Kafka producers for data replay purposes. Secondly, HDFS can be used as Spark cache allowing fast recovery in case of Job failure.

Another feature of Lambda Architecture not previously discussed is the data replay. Kafka Connect allows data to be read and ingested into a partition from a database or a data store. More specifically in case of Delfi missions, a Delfi-C3 and Delfi-n3Xt datasets can be stored in HDFS and be ingested into Kafka topic. This open doors for many applications, from Machine Learning applications, such as model testing, To Stream and Batch Processing system verification.

Implementation
With the Hadoop Distributed File System is discussed in detail in section 10.2.3, this section provides an overview of the data structure and funcionality to transfer data from Kafka topics to HDFS-based files.

The message retrieved from Kafka broker contain the original payload sent by the producer, the system API. The payload will depend on the mission and setallite and should not be edited in any way. At minimum, the payload contains: the raw telemetry frame in TBD format, user information, reception_time_client, reception_time_server (appended by the API), and client application information.

As discussed in section 10.2.3, HDFS is designed and optimised for large files. Storing small files impacts memory usage. Therefore, the telemetry frames contained in Kafka messages should be aggregated and stored as single files. Due to inherent uncertainties in timestamps of the client reception time and onboard clock drifts (see section 9.2.2), the aggregation is performed based on Kafka reception sequence. The use of partition sequence is unadvised as it is driven by the message keying, as discussed in section XX. Due to this reasons, Kafka ingestion timestamp is used. The ingestion timestamp is embedded in Kafka message as ingestion.time-parameter when message.timestamp.type is set to LogAppendTime in Kafka configuration.

As discussed earlier, storing each message in a file is feasible, but degrades performances and increased memory use. The messages are therefore aggregated in files. Manual aggregation to txt, xls, files is possible but is error-prone.

Three alternatives were considered, SequenceFile, AVRO and Parquet. SequenceFile is a row based key-value data structure, generally utilized in MapReduce jobs and therefore native in the Hadoop (HDFS) framework.
AVRO is a serialization, de-serialization framework that stores the schema and the content, the telemetry frame and metadata, in a compressed binary format. Storing the AVRO encoded binary, containing the contents of the message directly in HDFS does not solve the problem. However, library AvroTools provide Concat tool, allowing row-based multiple file aggregation. This is a multi-step process as HDFS does not allow update operations.
Parquet is a general-purpose columnar data store. Where AVRO appends and stores data as entries per row, Parquet stores the data per column in files. And is therefore optimized for column reads.

AVRO is the best fit for the problem, since, messages will be read sequentially per row. Secondly, the metadata and telemetry frame can be extracted from the file directly for the Batch Processing. AVRO is not locked in HDFS like sequence file and can be exported and ready in any other system.

10.2.3. Hadoop Distributed File System

Hadoop Distributed File System (HDFS) is a distributed file storage designed to be deployed on commodity servers. The commodity servers are defined as general purpose, low-end servers that are both cheap and easily replaceable. This directly correlates to heterogeneous hardware in term of hardware specifications and capabilities, and software, i.e. the operating system, within a single cluster.

The HDFS cluster consists of a master NameNode and number of slaves Datanodes distributed on a potentially high number of machines. With a large number of nodes, a higher chance of component failure is expected [124]. With this principle in mind, HDFS is designed to tolerate hardware failure, providing error detection and automatic recovery procedures.

In contrast to file systems like GlusterFS, HDFS does not comply with POSIX requirements. Borthakur argues that relaxing POSIX requirements enables streaming access to the stored files [124]. The most notable is the lack of "Update" operations, leading to immutable storage.

The HDFS system is designed to operate on large datasets consisting of large files (> gigabyte). To facilitate data manipulations, HDFS is optimized for high throughput but forfeits the quick data access (seek). Furthermore, to ease massive datasets manipulations, HDFS provides API for data access from MapReduce, Spark, and many others.

Architecture and Data Management

The HDFS is, ideally, deployed in a cluster with at least four nodes: one NameNode and three DataNodes. The NameNode is the central controller of the HDFS cluster and exposes the file systems namespace to the client applications. The provided API allows file and directory creation, name updates or deletion. Most notably the stored files cannot be edited.

NameNode

On the NameNode, the configuration of the cluster is stored in two files: EditLog and FsImage. EditLog is a transaction log, containing all changes to the files and file system. The FsImage contains the namespace, the mappings of blocks and files and properties of the HDFS cluster. The FsImage is kept in memory to increase the performance of the system, hence driving the server memory requirement. The required memory directly correlates to the number of blocks, with 3GB found sufficient for storage of 1 million blocks [125] [124]. If the file size is smaller than the block size, a complete block will be allocated. On the DataNode, the memory footprint of a thousand kB sized files will equal to that of a thousand 50MB sized ones. The second consequence is the performance reduction of the Read operations, as reading multiple files would require more DataNode operations, because HDFS is optimized for block per block operations [126]. One of the available optimizations is the aggregation of small (kB range) files to a single block. Published test runs yielded in a 45x performance increase and 3x reduction in memory use [127] comparing to block per file storage.

On the cluster start, the NameNode reads the EditLog, and when available, combines it with the existing EditLog (on disk) to create an in-memory FsImage object. As discussed in the streaming architecture, the EditLog is the ground truth of the system, containing all transaction, where the FsImage is the cache (or table database) of the current system state, that in HDFS terms is called checkpoint. In the default configuration, the NameNode is the single point of failure, requiring additional functionality to recover from the failure. Secondly, the EditLog and FsImage corruption is the second critical element, requiring manual operation for cluster recovery. To resolve the problem, secondary EditLog and FsImage can be configured, providing redundancy at the cost of performance. This is achieved in HDFS High Availability (HA) deployment, running a secondary NameNode in sync proving direct layover in case of primary NameNode failure. This feature will be discussed in the Experiment section. Alternative to Secondary NameNode is the CheckPoint node. The checkpoint contains a snapshot of the HDFS namespace state aggregated from EditLog and FsImage with the goal to reduce the size of EditLog and when combined with Backup services provide additional redundancy.

File Write and Read

The files stored in HDFS, are split into blocks with a fixed preconfigured size and replicated across DataNodes. With Federation deployment, the HDFS can be rack aware and place working and replica blocks on different racks, facilitating higher intra-rack performance and inter-rack redundancy. The File blocks are fixed in size and default to 64MB in size. Larger files will be split into multiple blocks of equal size, except for the closing block that will be dynamically sized to fit. To determine the file

size, and pre-allocate the required blocks, the HDFS client application API caches file locally on the client machine. The cached file Write-request (DistributedFileSystem API) from the client application is handled by the NameNode, that determines the available DataNodes for data storage and provides information to establish RPC interface for writing (FSDataOutputStream) directly on the DataNode(s). On the NameNode, file information is appended to the file system permanently once the file-block was stored on the DataNode and acknowledged by the client application. The written file blocks are replicated from the original DataNodes in accordance with the replication policy. When replication is completed, the acknowledgement will be issued via FSDataOutputStream to the client writing the file, which will issue the close command to the file and acknowledge the completion to the NameNode. Failure to acknowledge file-block write will terminate the operation and discard the file. The file Read following the same principle, the FSDataInputStream provides access to the DataNodes containing the file blocks. The blocks are read sequentially and aggregated on the client.

DataNode
The DataNodes store blocks as files, purposefully abstracting them from the original files they refer to. The DataNode runs as a java based application that stores and retrieves block-files from OS file system. The block-files are stored in a tree of directories, not related to the HDFS directory, to prevent OS file system paging bottlenecks. To keep track of block-files and directories the block-files are placed in, DateNode relies on a Blokreport object. Similar to other storage and transport protocols, the file-blocks contain a checksum, to ensure data integrity.

Designed to cope with hardware failure, the NameNode keeps track of active NameNodes by use of heartbeats, status messages sent periodically (3seconds) from the DataNodes. Nodes with missing heartbeats are considered as failed and will be removed from the HDFS cluster. This likely triggers replication of the blocks on other nodes to maintain the initial replication factor. The deletion of files or forced replication due to hardware failure will likely disbalance the cluster, leading to overutilization of some of the nodes. In practice, HDFS built-in balancer ensures equal utilization or the disk capacity across the DataNodes without consideration for block utilization (frequency of access).

Deployment in Production
As discussed in section 10.2.3, HDFS is designed to be deployed as a cluster of servers. This prevents data loss due to hardware failure and ensures system availability. Due to the ”moving computaiton” methodology, the data and processing framework should coincide on the same hardware. The minimum number of nodes is driven by the replication factor, which is often set to three. This results in three DataNodes and a single NameNode.

The HDFS should be deployed on dedicated servers, separated from Presentation Layer and Ingestion Layer nodes. While in PoC phase contarization provide best testing results, the HDFS container deployment should be avoided. In accordance to Hortworks best practices, Hadoop ought to be deployed on barebones hardware.

10.2.4. Apache Spark
The spark cluster consists of worker nodes and a single leader, named driver, that is elected manually. Spark depends on the cluster manager, such as YARN for resources allocation for Jobs and ensures that hardware resources are not exceeded. This is required since Spark is operarted in-memory, pushing the data to nodes RAM. While in stream processing the data sizes are moderate small, in batch Spark can operate on the entire data set that likely exceeds available RAM. The data is therefore split in smaller sets called Resilent Distributed Datasets (RDD). Operating as a distributed application, Spark splits the job in accordance to the required transformation and attempts to limit the intra-node operations.

The RDD are manipulated directly by the Spark API and are available programmatically. The field of (BigData) distributed processing is not yet matured and more technologies are been interoduced. So, version two of Spark introduced a higher level RDD API allowing SQL-like processing and structured streaming.

Kafka Pull
Data ingested from Kafka broker, stored in Spark Resilient Distributed Dataset (RDD), which consists at least of three parts or elements. The first is the Kafka-key. Key is not required, and if not provided, will be set to ’null’. It is important to keep in mind that Streaming applies micro batches, pulling data from

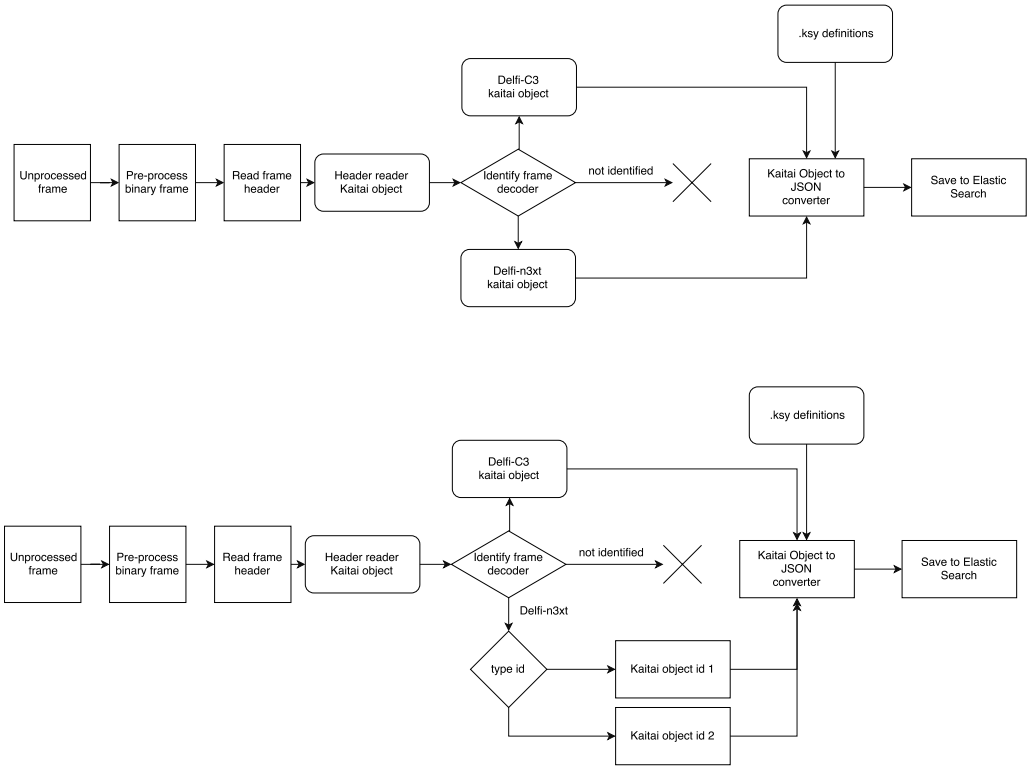

Figure 10.6: Kaitai

Kafka log. This means that if the log does not contain any data, an empty RDD will be placed in the Dstream, and been executed on fixed intervals, storing and processing empty RDD has to be avoided.

The Dstream object, contains the RDD and can be both stateless, allowing RDD-per-RDD operations or statefull, allowing aggregated operations (related to windowing principle).

The second element of the RDD is the receiver (RA, GS) metadata, for example, the reception timestamp or client application version. It is passed directly to the storage and is ignored by processing. Therefore it is important to encode data in an acceptable format, for exmaple JSON.

The third element is the telemetry frame. For the sake of simplicity of the PoC, data is sent in the String format. This is enables direct Delfi-n3Xt data ingestion for the testing purposes. For the final design and implementation, String encoding is unacceptable is a major waste of resources.

Processing

The third element of the RDD is the original, raw received satellite frame. The satellite can be seen as a binary encoded hashmap, with predefined untagged fields. In case of Delfi-C3 and Delfi-n3xt, the frame is adaptation of AX.25 frame definition, with loosely defined payload field and reduced header size. The header consists of start flag, used to mark begin of the frame, address and control fields to identify the spacecraft and PID control fields. The header is invariant, and is fixed to the spacecraft. The payload definition depends on the mission. In case of Delfi-n3xt, the payload contains an id field identifying the encoding of the bulk of the payload field. This should be read in order to correctly parse the field.

Tne parsing is performed by Kaitai library. The key decision moment of the processing script lies in the logic of the payload schema selection and be performed by the Kaitai object or python spark logic as discussed in section 5.5 and illustrated in figure 10.6.

Delegating the frame schema detection to Kaitai requires a complete job overhal for any additional processing frames. First, ksy file has to be adjusted to accommodate for the new definition, then new python parser object has to be generated and transferred to the spark directory. Job has to be restarted to take the effect. In case B, the frame type id, would require job to python read file with preallocated names.

The conclusion is that option A is useful for missions with fixed frame definitions like Delfi-C3, where

the frame id is invariant, and Delfi-n3xt, since no new data is been produced. In theory. Option B is useful for active missions, as it allows frame definitions to be added dynamically. The feasibility of this approach require further investigation, as each job is deployed as a single container.

10.2.5. Apache YARN

Apache YARN or Yet-Another-Resource-Negotiator is resource management and a scheduling tool for a distributed system. YARN consists of two components, Resource Manager and a Node Manager per node. Resource Manager is the bridge between the client applications requiring job scheduling and Node Managers, responsible for node resource allocation and worker containers. The ResouceManager is the master and ensures that deployed job can be executed on the cluster in terms of resources: CPU, Memory, Disk and Network. The deployment, accepting of the job and allocation of resources is done by ApplicationsManager. The Resource manager furthermore contains Scheduler, ensuring job scheduling, but performs no monitoring or job restart.

10.2.6. Zookeeper

Where YARN ensured the ability to execute the job, Apache Zookeeper ensures that the job is completed. In essence, Zookeeper is a distributed hierarchical Key-Value store, that is used to provide configuration, synchronization and other functionality to the part of the system.

In Kafka, Zookeeper data store is used to keep track of partition leaders as well store of the offsets of the Consumer groups.

In HDFS Zookeeper is used to configure standby Namenode, ensuring that each DataNode is notified on Mater change.

10.3. Architecture

The proposed architecture, with the major components defined in section 10.1 is shown in figure 10.7. The figure shows two views of the system. A provides cluster overview, separating the technologies per Layer. View B shows the advised server allocation.

As shown in figure B 10.1, the advised server allocation requires a minimum of four servers. Three are required to ensure Kafka and HDFS High Availability, while only is needed for telemetry processing. If processing speed is found insufficient, additional processing nodes can be added to the Spark cluster at any time.

It is likely that the change in processing performance requirements, and hence the addition of extra data processing nodes, will correlate with higher data ingestion rate. This would require migration of HDFS systems from ingestion nodes (Kafka) to processors.

The allocation of HDFS to Spark nodes is driven by Spark memory optimizations, where the batch jobs would load the data in the allocated RAM for processing. Separating HDFS and Spark introduce network latency, hence adversely affecting the performance.

Placing Kafka and HDFS on the same nodes is a measure to decrease the cluster size (i.e. number of servers required). The co-allocation increases the chance of data loss as both Kafka and HDFS are responsible for the storage.

In the ideal scenario shown in figure A 10.1 the data ingestion Layer, Kafka can be deployed on virtual machines. With low required specifications (± RPI performance), the cost of operation is low. The data processing system should be deployed on hardware instances due to the use of HDFS. In the scenario B 10.1, the use of virtualization is unadvised due to HDFS presence on all nodes of the system.

The proof of concept introduced in section 11, and the discussion above, the requirement assessment was performed. The primary needs are summarized in table 10.1, and correlate to the requirements presented in appendix A. As shown by the overview of PASS/FAIL of the requirements in section A.7, all in-scope requirements have been fulfilled.

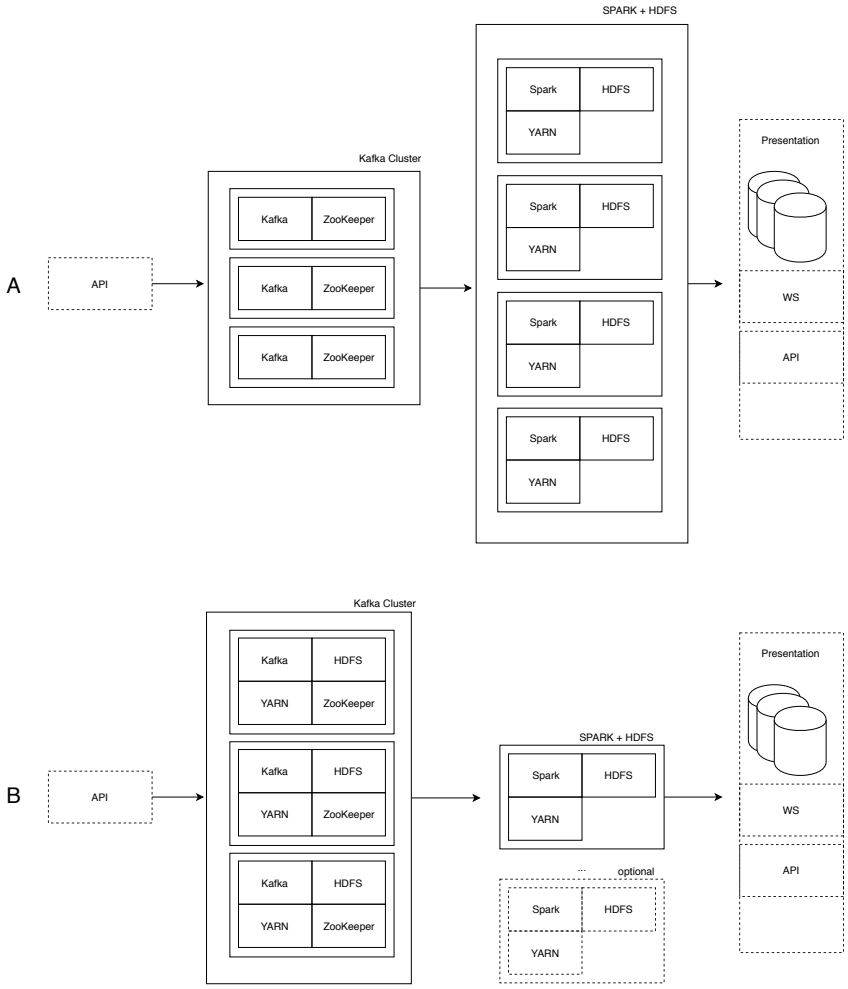

Figure 10.7: Telemetry processing architecture: a) Cluster b) Proposed deployment

11

Experiment and Research Questions discussion

The chapter aims to validate the requirements, proving proposed architecture applicability to Delfi mission, answering the research questions.

11.1. Scope of System Testing

The architectural design proposed in section 10 requires validation against the initial set of requirements.The functional requirements validation require proof of concept. Furthermore, to answer the research questions defined in chapter 1 hardware experimentation is required.

11.1.1. System Robustness Assessment

Robustness is defined as run-time tolerance to errors and inconsistencies. Within the proposed architecture, the following source of errors have been considered:

- Errors in ingested data: telemetry frame errors
- Errors in ingested data: errors caused by the client application
- Errors in processing: exceptions handling and anomalies in the computing
- Errors in processing: errors in the processing code
- Errors in communication between components
- Loss of components

The errors in processing, such as faulty input data (client application bugs) can be handled with Try-Catch exception blocks in the programming and requires no further investigation. Furthermore, the errors in the processing code are most likely used by human errors: bugs.

The use of distributed computing increases the risk of communication loss between components and single component failure. The experiment should therefore investigate the tolerance of the system to component loss and recovery modes of the system.

11.1.2. System Reliability Assessment

Reliability is one of the three classic hardware quality attributes: Reliability, Availability and Serviceability. In the scope of the software system, the reliability is a measure of the software failure rate in terms of downtime or incorrect results and is often modelled as a probabilistic function [128] or based on a fault-tree analysis [129]. Both approaches require the fully implemented functional system operating on the target hardware or historical data, both of which are unachievable within the scope of the project. Petrov argues that the reliability is affected primarily by the number of the errors in the code, and shows a linear relation between lines of code and number of programming errors [128]. When applied to automotive applications, Petrov's analysis indicates an exponential growth in error

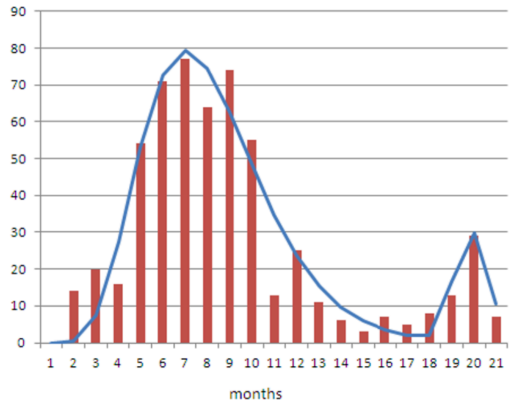

Figure 11.1: Defects discovered versus time [128]

detection in the first four to seven months of operational use as illustrated in figure 11.1. Which in case of Delfi, correlates with a typical internship and thesis duration.

Reliability is expressed as a programming error metric closely related to the robustness, and it can be argued that both are closely related. The relation is one-sided as the error-free and hence reliable software contributes to the robustness, but robust software on its own cannot guarantee reliability.

Monperrus suggests an inverted view on the problem by " ... thriving and improving when facing errors" [130]. The proposed antifragile concept relies on self-checking, self-repair and failure injection in the production system. While self-checking and self-repair software are not feasible within the scope of the project, the concept can be thought of as a reduction of uncertainty by introducing failures to the system. The latter is supported by Tseitlin who argues that "... [a] failure [in a distributed system] is not predictable and does not occur with uniform probability and frequency"[131], and that the theoretical prediction and probabilistic concept are not feasible [131] [132]. Both Tseitlin and Monperrus argue that first step in the process is software testing. In practice, this is adequately achieved through the use of unit testing or test-driven development. This correlates back to the architectural decisions, requiring independent components testing, as it is the case in the Clean Architecture principle.

Tseitlin furthermore argues that capturing the behaviour of secondary systems, i.e. data source or data sinks, is complex and unfeasible in practice. This leads to the concept of proactively introducing errors into an operational system to decrease the uncertainty. Simply put: learn from the mistakes. The concept does not support software reliability, but operational resilience, which arguably leads to the same results: reliable system. An example of the approach is Netflix Chaos Monkey [133], a service that actively introduces errors to the system: latency, network partitions, application outages, CPU overutilization, disk loss, etc. Regardless of type of error introduced, the introduction of errors requires fault-tolerance, either in the form of redundancy or robustness.

The second element to ensure the system robustness is the bus factor, a metric for the single point of failure within a team. Simply put, how many team members can leave the project ('run over by the bus') before the system is no longer maintainable. Due to the limited size of the team, the reliability will greatly depend on the quality and quantity of the available documentation of the system design, system operations, operation checklists," lessons learned" and implementation documentation.

The aforementioned leads to separation of Reliability into Software Reliability and Operational Reliability. While the Software Reliability can be seen as a measure of software quality (defects metric), the Operation Reliability is an aggregation of software and hardware redundancy, the capacity for Error/Failure handling, and Maintainability. In this case Maintainability is expressed as a metric for the long-term strategy with regards to handling, i.e. possible data degradation and data loss.

11.2. Research Question evaluation

The application of research questions to the proposed architecture is summarized in table 11.1. While the majority of the metrics presented in the table are deduced from the vendor documentation and information presented in the previous chapters, some of the attributes require further investigation. The following sections discuss the parameters and evaluate the experiments required for the validation.

11.2.1. Errors originating form client application

Use of client application for data reception introduces a wide spectrum of potential errors and inconsistencies. As discussed in the chapter 9 the client application is assumed to deliver raw and unprocessed telemetry frames. Due to error-mitigating nature of the software development observed in the past systems, the chance of data corruption by the client application is negligible.

In case of data corruption, the Kaitai processing will throw an exception ignoring the processing of faulty data. The recovery from extensive data corruption introduced by the client applications will depend highly on the specifics of the error but could be resolved with custom Spark jobs.

11.2.2. Effects of unstable networking and loss of nodes on Ingestion Layer

The ingestion layer is responsible for the aggregation of data originating from ingestion API, client applications and potentially other data sources such as data acquisition systems. Due to the separation of system API from the Kafka System, the loss of Kafka may result in data loss, depending on API implementation and producer semantics.

The loss of Kafka System inevitably terminates all dependant application such as Spark Streaming jobs and require stream processing job restart.

While failing under quorum, i.e. more than two failed nodes out of the cluster of three, requires manual intervention, the single node recovery could be performed automatically. The research on COTS service provider availability focus on three primary failure factors: upgrades (e.g. human errors in configuration), networking and software bugs [134]. Unfortunately, in the majority of the cases, the source of the system failure cannot be established. Furthermore, the system does not stop working but behaves unpredictably, exponentially degrading the performance, a process known as gray failure [135].

The aforementioned discussion leads to the following list of points of interest:

1. Effects of configuration errors
2. Effects of networking errors
3. Effects of software bugs
4. Effects of gray-failures

With the use of the in-house hosting, the configuration errors are highly likely to occur. However, the system load is initially low due to single satellite utilization, no operational upgrades or scaling out operations are expected for the first year(s) of operations. This gives adequate time to prepare for the required upgrades. The same principle applies to the software bugs, both are therefore removed from the list.

The networking errors or network partitioning is described as a frequent source of malfunctions [134] in a clustered environment. In case of ingestion system, Kafka documentation points to graceful node termination (handled by OS) in overcapacity cases such as disk storage, while lacking documentation on networking overcapacity effects. Finally, the gray-failures are hard to quantify and reproduce. To investigate the network effect following experiments will be performed:

1. Effects of network oversaturation
2. Effects of network failure (connection loss)
3. Effects of network failure recovery (connection recovery)
4. Attempt to reproduce gray-failure

It should be noted that the network traffic within the Kafka cluster is larger in magnitude than the traffic between the data source (Producer) and Kafka cluster. This is the consequence of topic replication, as replication rate of two increases the intra-cluster data traffic two-fold. Furthermore, the Kafka producer directs data directly to the appropriate partition leaders, efficiently splitting the traffic between the nodes. The ability to store all incoming data, regardless of the system load, is critical, while topic message retrieval rate is not. Therefore, the Kafka testing will focus solely on the data ingestion for the experiments.

Table 11.1: Research Question evaluation summary

Item	Solution
Robustness	
Telemetry frame errors	Exception handling (Python)
Errors originating form client application	Unknown at time of writing
Anomalies in frame processing	Exception handling (Python) for parsing
	Failed Spark jobs: Checkpoint & Recovery on HDFS
	Stuck Spark jobs: Manual termination, Checkpoint recovery
Connection loss (network partitioning)	Kafka node loss: unknown impact for ingestion layer, no impact on Stream processing as long quorum is maintained. Experiment required
	Kafka node loss under quorum: Kafka failure. Temporary loss of partitions, incoming data rejected.
	Spark node loss: low impact on stream layer, with higher latency in processing; possible delay in low-latency batch jobs. Experiment required
	HDFS node loss: no impact on Kafka, Spark or Spark Streaming as long quorum is maintained. Connected clients will be redirected automatically.
	HDFS node loss under quorum: No impact on Kafka, Spark batch jobs failure, Spark Streaming jobs failure if YARN is affected.
	Zookeeper node failure: no effect
	Zookeeper all node failure: Kafka terminated.
	YARN singe node failure: no effect
	YARN all nodes failure: HDFS and Spark new jobs allocation terminated
Programming errors in jobs	Pre-check by Spark Batch and Spark Streaming
Reliability	
Unit testing	No experimentation required.
Operations	Experiment required
Maintanability	Experiment required
Redundancy	See robustness: server node loss

11.2.3. Reliability in Operations and effects of Maintainability

In section 11.1.2 the system reliability was proposed as the team readiness for a system component failure. To assess the reliability, in form of the ability to debug and comprehend the system failure, the experimentation was implicitly performed during the configuration of the cluster and development of Spark jobs. The available documentation is excellent, with the majority of encountered error messages readily explained in online resources.

The maintainability, as a measure of the ability to modify system for changes, is excellent. Spark streaming job used for Stream processing was adjusted to perform the Batch job with limited changes (less than ten lines of code). The exchangeability of the code, for example, change of data stores from ElasticSearch to HDFS require minimal code adjustments. The use of Spark Structured Streaming and Spark SQL allows the use of Pandas syntax familiar to Python developers and SQL-like syntax used in the legacy Delfi systems.

11.3. Hardware Experiment Setup

The proposed experiments require saturation of the server resources, for example, the network card (NIC), to study the ingestion layer robustness. The gray-failures are hard to reproduce but are linked to high system utilization. Running these tests on an optimized hosted cluster or even commodity level server would require substantial datasets.

Therefore, the experiments were performed on a cluster of 4 Raspberry Pi's (RPI), each equipped with 1 GB of RAM and 1.2 GHz quad-core ARM based CPU running modified version Debian (Raspbian

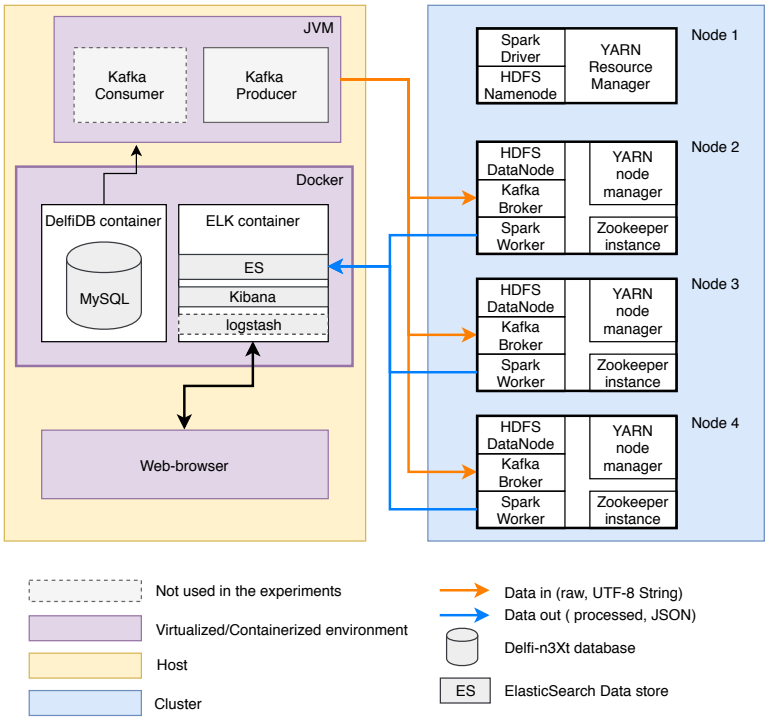

Figure 11.2: Experiment setup

Table 11.2: Typical RPI performance characteristics

CPU	1.2 Ghz (600 Mhz throttle)
Memory size	1 GB
Storage capacity	32 GB
SD card average (AVG) Write speed	21 MB/s
SD card AVG Read speed	40 MB/s
Network AVG speed	11 MB/s

Jessie) connectie via a wired 1 gigabit switch. The use of underpowered hardware allows the system to be tested under high load conditions. Secondly, the requirement for in-house deployment requires a level of expertise, that is not necessarily feasible in a ready-to-use cloud deployment. Use of RPI nodes allows fast cluster reconfiguration and reinstallation. Setting up and running on underpowered hardware allows more substantial tweaking and a better understanding of the effects settings have on the complete system.

All four nodes are configured to run HDFS, YARN (resource manager and negotiator for HDFS), SPARK, Zookeeper (required for Kafka) and Spark. For each experiment, only the necessary services were enabled unless stated otherwise. The HDFS NameNode (Master in the non-HA deployment) and Spark Master node coincide on node1 that does not contain the DataNode or the Spark executor node.

The cluster monitoring system is based on the Prometheus project, a client application submitting node status updates and the central Prometheus server responsible for data aggregation and visualizations.

For PoC, the presentation layer is implemented with ELK stack, which is installed outside the cluster and contains ElasticSearch storage and querying system and Kibana front-end application. The ELK separation from the processing logic is two-fold. First, ELK is not necessarily part of the telemetry processing system and therefore does not directly contribute to the research. Secondly, by following the previously established methodology, components are easily exchangeable and should be selected based on the system needs.

The selection of ELK stack for PoC is driven solely by the need for operational testing, requiring data visualization and fast lookups of processed data. Hence, ELK is merely used as a tool to validate the processed data.

11.4. Limitations

Use of Pi-based cluster has three significant limitations. First, the hardware has extremely limited memory and requires fine-tuning of components. Throughout experiments, YARN prevented overutilization of memory which resulted in high swap memory use. Due to use of SD-card based disks in RPI hardware, the substantial hardware wear was observed that likely lead to cluster instability found during last experiments.

The second limitation is the inherent sensitivity of RPI hardware to the power supply. Running on standard 5V USB power supply, a power consumption of up to 3.5W have been observed. Under continuous high load, power supply showed instabilities by reducing output voltage under 5V that combined with the voltage drop in USB cables lead to RPI CPU throttling and instabilities. The effects of power supply on RPI stability were determined late in the process and could not be resolved within the timeframe of the project.

Lastly, the RPI is based on ARM architecture. While no conflicting dependencies were discovered, the software may be executed in non-standard modes and may affect the results. Eventhough no conclusive evidence was found, it increases the uncertainty of the test results.

11.5. Experiments

In the proposed architecture, the loss of the data ingestion layer has the highest impact on the system operations. While the errors and malfunctions of any other system component may result in delay or temporary unavailability of data in the presentation layer, the telemetry data, either processed or raw cannot be corrupted or lost.

11.5.1. Kafka

As described in section 11.2 due to the distributed nature of the system, the networking is critical to the system operations. Two aspects will be studied: network saturation and intermittent connection loss.

Hypothesis : *saturated network (interface) will cause a high load on the system due to queueing of data, and subsequently reduce the data ingestion rate with potential data loss due to message rejection.*

One can identify two distinct network paths: from Producer to Kafka and networking between Kafka-nodes. The proposed experiment is built on the assumption that Kafka's single API node will be used, leading to a single Kafka Producer Architecture. The API, acting as the gateway to the cluster, is connected to the cluster network, hence guaranteeing substantially higher network speeds to the cluster than to the client applications. This leads to the conclusion that network saturation is likely to occur either in Client - API network or internally within the Kafka cluster. For the experiment, only internal Kafka node-to-node network is studied. This is the consequence of the replication as each ingested bytes in replicated, hence amplifying the network load two- or threefold.

As illustrated in figure 11.3, Kafka cluster used for experiments contains three nodes and utilizes one single topic. The topic consists of two partitions and two replicas. The number of topic partitions depends on the cluster size and the parallelism of data consumers. A higher number of partitions and replicas increase the cluster load, obfuscating the results required for the experiment.

Study of node failure effects requires at least three nodes cluster, as two node cluster would limit the replica number. More than three nodes will increase the network traffic, which is limited due to RPI hardware, hence affecting the experiment results.

The data used in the experiment is based on Delfi-n3Xt telemetry dataset with string UTF-8 encoded unprocessed telemetry frames as used in Delfi-n3Xt telemetry processing system. Kafka producer, the application responsible for data ingestion, operates in burst fashion inserting 100,000 telemetry frames with 100ms cooldown between the runs. The duration of each run is measured and presented in table 11.3.

Hypothesis validation : *A high volume of data is submitted to the Kafka cluster with monitoring of network use and metrics of all cluster nodes. Additionally, the node connection is terminated to study the reliability and robustness of the system.*

The experiment in split into two parts: A and B. The first part of experiment A establishes the baseline of the processor (CPU) and memory (RAM) utilization, load, network and disk use metrics, as well as the complete system throughput rate. The second part studies the effect of network loss

Table 11.3: Measured ingestion rate for experiments A and B

Experiment A			
Point	Run	Run duration [ms]	Speed [f/s]
3 - 4	1	32824.17	3047
	2	29272.35	3416
	3	25029.88	3995
	4	17703.70	5649
	5	15458.42	6469
	6	16460.83	6075
5 - 6	1	17936.66	5575
	2	17337.57	5768
	3	15777.85	6338
	4	18657.23	5360
	5	17488.76	5718
	6	15446.28	6474
7 - 8	1	16280.74	6142
	2	16063.81	6225
	3	19183.20	5213
	4	18319.08	5459
	5	19617.96	5097
	6	16833.55	5941
9 - 11	1	44771.06	2234
	2	26098.23	3832
	3	23109.68	4327
	4	22669.95	4411
	5	24843.33	4025
	6	24099.09	4150

Experiment B			
Point	Run	Run duration [ms]	Speed [f/s]
2 - 3	1	23449.37	4264
	2	20206.41	4949
	3	21256.78	4704
	4	21376.15	4678
	5	22417.51	4461
	6	54863.55	1823
3/4 - 5	1	17160.03	5827
	2	16282.99	6141
	3	15918.55	6282
	4	16435.50	6084
	5	15560.39	6427
	6	18532.77	5396
6 - 7	1	14475.35	6908
	2	13989.63	7148
	3	14809.06	6753
	4	14677.17	6813
	5	14337.86	6975
	6	14609.97	6845
8 - 9	1	13588.37	7359
	2	15266.28	6550
	3	14632.17	6834
	4	15962.94	6265
	5	14217.36	7034
	6	14903.42	6710

Table 11.4: Measurement point description for experiments A and B

Experiment A	
Point	Description
1	ZooKeeper start
2	Kafka start
3	start run 1
4	end run 1
5	start run 2
6	end run 2
7	start run 3
8	end run 3
9	start run 4 node 2 disconnected
10	end run 4
11	node 2 reconnected

Experiment B	
Point	Description
1	ZooKeeper and Kafka start
2	start run 1
3	end run 1
4	start run 2
5	end run 2
6	start run 3
7	end run 3
8	start run 4
9	end run 4

of ZooKeeper and Kafka partition leaders. Experiment A is discussed in section 11.5.1. Experiment B attempts to recreate a gray-failure of the cluster and studies the effects on performance and stability (as metric for robustness and reliability) of the cluster under failing conditions. Experiment B is discussed in section 11.5.1.

Experiment A: baseline and network failure

Experiment A consists of two parts. In the first phase, the baseline for data throughput is established. The second phase focuses on the study of the network loss effects.

The first three parts of the experiment, represented by 3-4, 5-6 and 7-8 in the figures 11.4 and 11.5, and tables 11.3 and 11.4, show a bell-shaped performance curve, with an initial increase in performance over the first bursts (runs) and decrease towards the end of the run. A difference in ingestion speed between first runs of experiments two and three (measurement points 5 and 7 respectively) indicate a system hysteresis as a longer cooldown time was applied in the former (two versus one minute). The difference is the most likely the consequence of a higher load of the node 2, as can be seen in average load figure 11.4 measurement points 5 and 7. As illustrated in figure 11.3, the nodes 2 and 3 replicate data to node 1. Figure 11.5 supports the theory and shows inbound traffic of node 1 that is approximately the sum of node 2 and 3 outbound data. Furthermore, the volume of data written on node 1 is approximately the sum of data read on nodes 2 and 3. Furthermore, it can be seen that Kafka messages are compressed in the transfer, as a disk throughput is higher than the network.

With the maximum network speed of 11MB/s [136], it can be seen that network interface (NIC) of RPI node 1 is fully saturated. The load metric in figure 11.4 encapsulates the CPU, RAM, network and disk queuing and shows a uniform load across the cluster, regardless of the ingestion rate. A positive trend is observed for node 1, indicating increasing network congestion and latency between leading Kafka partitions (nodes 2 and 3) and ISR on node 1. Furthermore, a 3% decrease in the ingestion rate due to replication queuing to the redundant node (from AVG: 5872 STD: 439 to AVG: 5679 STD: 487) was observed.

The second phase of the experiment focused on the effects of node loss in operations. In the experiment, node 2 hosts both the ZooKeeper as Kafka partition (0) Leaders. Failure of node 2 has, therefore, the highest impact, as both ZooKeeper as Partition elections have to take place.

As illustrated by the network figures 11.5, the replication queue to node 1 was resolved at the 15th minute of the experiment. The node 2 was not disconnected prior to prevent data inconsistencies and possible corruptions. While the graceful disconnect is unlikely to occur in the real world applications, the replication latency is not representative of the real world server centers with significantly faster networking as well. For example, a typical 10 Gbps network is roughly 200 % faster than the experimental setup used.

The second phase of the experiment consists of a single burst of 6 runs, as shown in the table 11.3. The node 2 is disconnected simultaneously with the burst start, delaying data ingestion at Kafka until the new ZooKeeper and partition leaders are elected. The ZooKeeper is instrumental for messages offset tracking, therefore delaying the data ingestion from Kafka producer, hence reducing the performance of the initial run. Figure 11.5 show an imbalance of traffic between nodes 3 and 1, which is the consequence of the partition 1 ISR on node 1. The outbound network and disk read activity on node 1 show replication to node 3, pointing towards a single point of failure. This is expected as the automatic partition rebalancing is disabled by default. As a side note, while repartitioning changes the key partition co-allocation, the rebalancing does not, allowing recovery from a permanently failed nodes. At measurement point 11, after the 600,000 telemetry frames ingestion (6x 100,000), the networking to node 2 was restored. This shows that 1.5x data size of the Delfi-n3Xt mission was replaced within 45 seconds between the nodes.

Experiment B: baseline and node failure

Experiment B focuses on intermittent node failure. The experiment is done in two phases; first data ingestion is initiated followed by ZooKeeper and partition Leader failure. The second phase investigates the behaviour towards lost node and while attempting to cause data corruption.

The node failure is initiated by reducing RPI input voltage close to the operational threshold (4.75V), causing Linux kernel malfunction. While control over the node (ssh) and high-level function was lost, the node continued to respond to network requests (ping) and submitted (faulty) heartbeat information.

Table 11.3 indicates the ”hard” failure in the 6th run of the first experiment, approximately at 3min30sec as can be seen in figures 11.6 and 11.7. The initial setup deviates from figure 11.3, with partition 0 leader on node 1 and partition 1 leader on node 2. The loss of node 1, allocated leaders to nodes 2 and 3 respectively and caused failure of the replication as can be seen in network activity in figure 11.7.

Multiple anomalies can be observed in figures 11.6 and 11.7, first, the time-stamp of Kafka producer is 15 seconds off from the cluster clock. Secondly, the measurement point 5 shows an earlier Producer termination time-stamp than the illustrated by the network and disk activities.

During the experiment, no data loss acquired and Kafka producer, responsible for data ingestion, did not encounter any exception or faults.

11.5.2. Spark Streaming

As discussed in section 11.2, the reliability of an application depends on the number of possible errors in the source code. By following this logic, one can argue that a reduction in the number of lines of code (LoC) will actively reduce the probability of bug occurrence and can potentially increase the software reliability. This is however a feeble measure of the reliability parameter as LoC does not account for code complexity.

The code below show the PySpark source code for Kafka topic offload functionality to HDFS. The topic messages are aggergated by 10 seconds intervals and stored in year-month-day folder structure:

```
spark = SparkSession.builder \
```

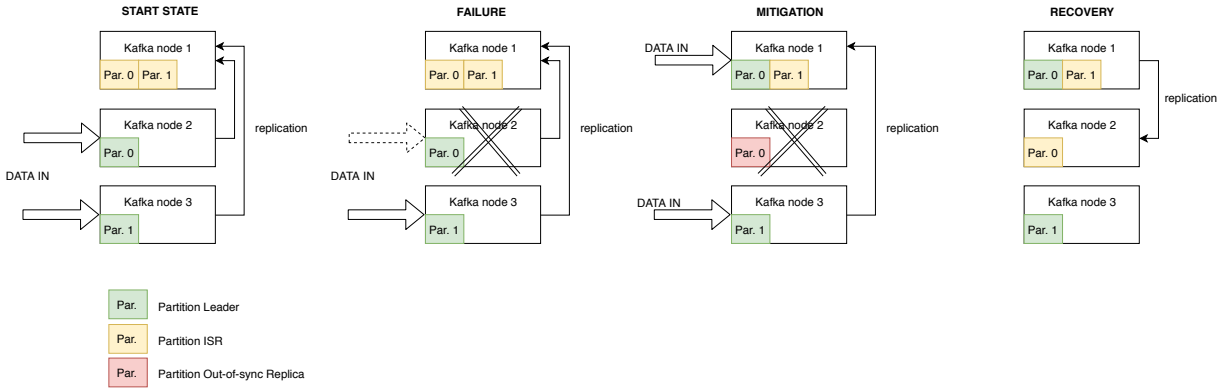

Figure 11.3: Kafka setup

```
    .appName("test").getOrCreate()

df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "slavenode2:9092,slavenode3:9092,slavenode4:9092") \
    .option("subscribe", "delfi6") \
    .start()

hallo = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)", "year(timestamp) as year",
                      "month(timestamp) as month", "dayofmonth(timestamp) as day").writeStream \
    .format("csv") \
    .option("checkpointLocation", "/tmp/") \
    .partitionBy(["year", "month", "day"]) \
    .option("path", "hdfs://masternode1:9000/sink/sat=n3xt") \
    .trigger(processingTime="10 seconds") \
    .start()

hallo.awaitTermination()
```

As an example of ease of code modification, consider Kafka message offload to ElasticSearch:

```
spark = SparkSession.builder \
    .appName("test").getOrCreate()

df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "slavenode2:9092,slavenode3:9092,slavenode4:9092") \
    .option("subscribe", "delfi6") \
    .start()

hallo = df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)", "year(timestamp) as year",
                      "month(timestamp) as month", "dayofmonth(timestamp) as day").writeStream \
    .format("es") \
    .option("checkpointLocation", "/tmp/") \
    .option("es.nodes", "192.168.1.2:9200") \
    .option("es.nodes.discovery", "false") \
    .option("es.nodes.wan.only", "true") \
    .start("n3xt/frames")

hallo.awaitTermination()
```

As can be seen in the experiment above, Spark hides the code complexity from the developer with use of abstraction. This is fitting to Delfi operations as TPC system is modified infrequently and no specialized knowledge, i.e. project is required for system adjustments. Furthermore, the code is not

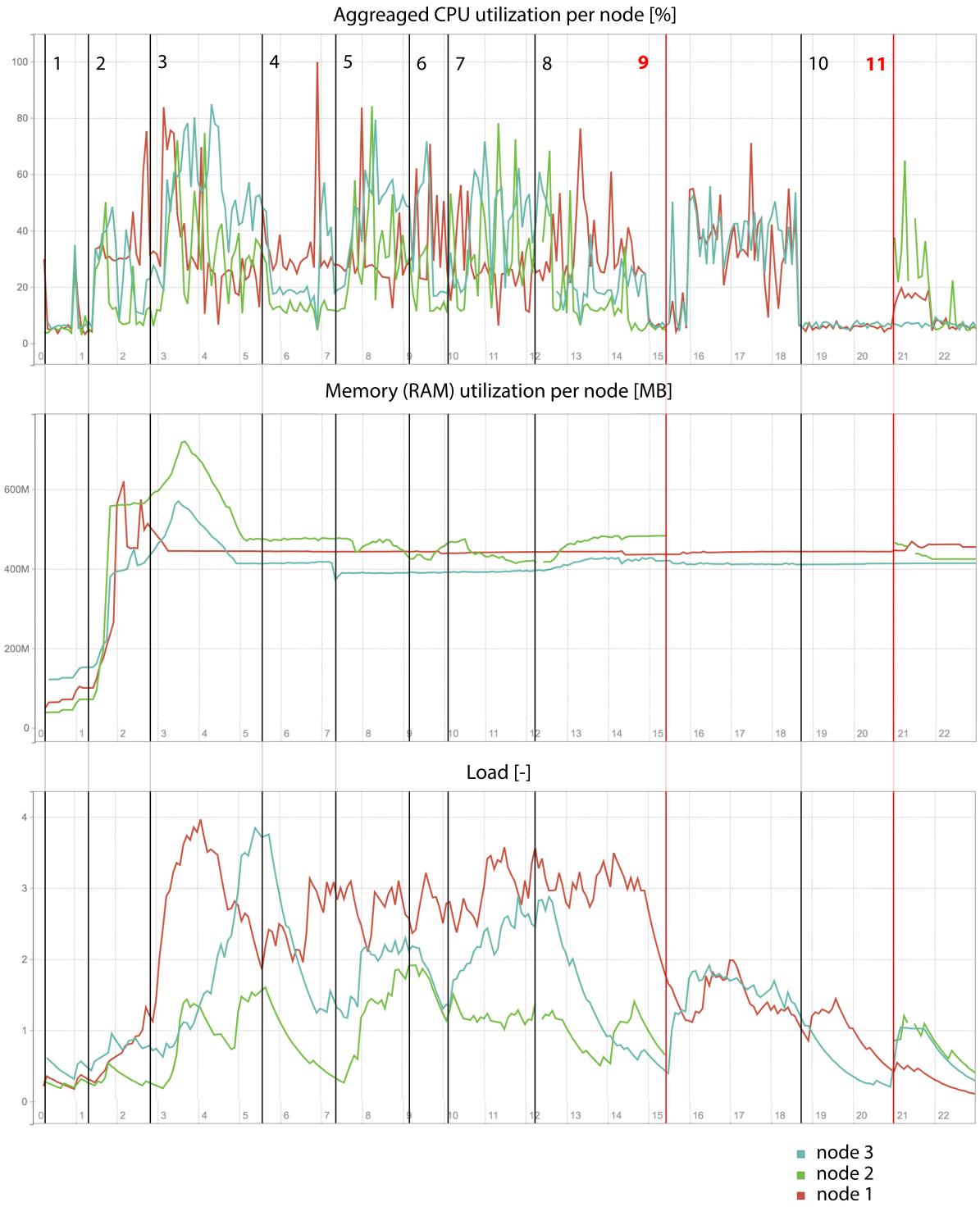

Figure 11.4: Kafka experiment A

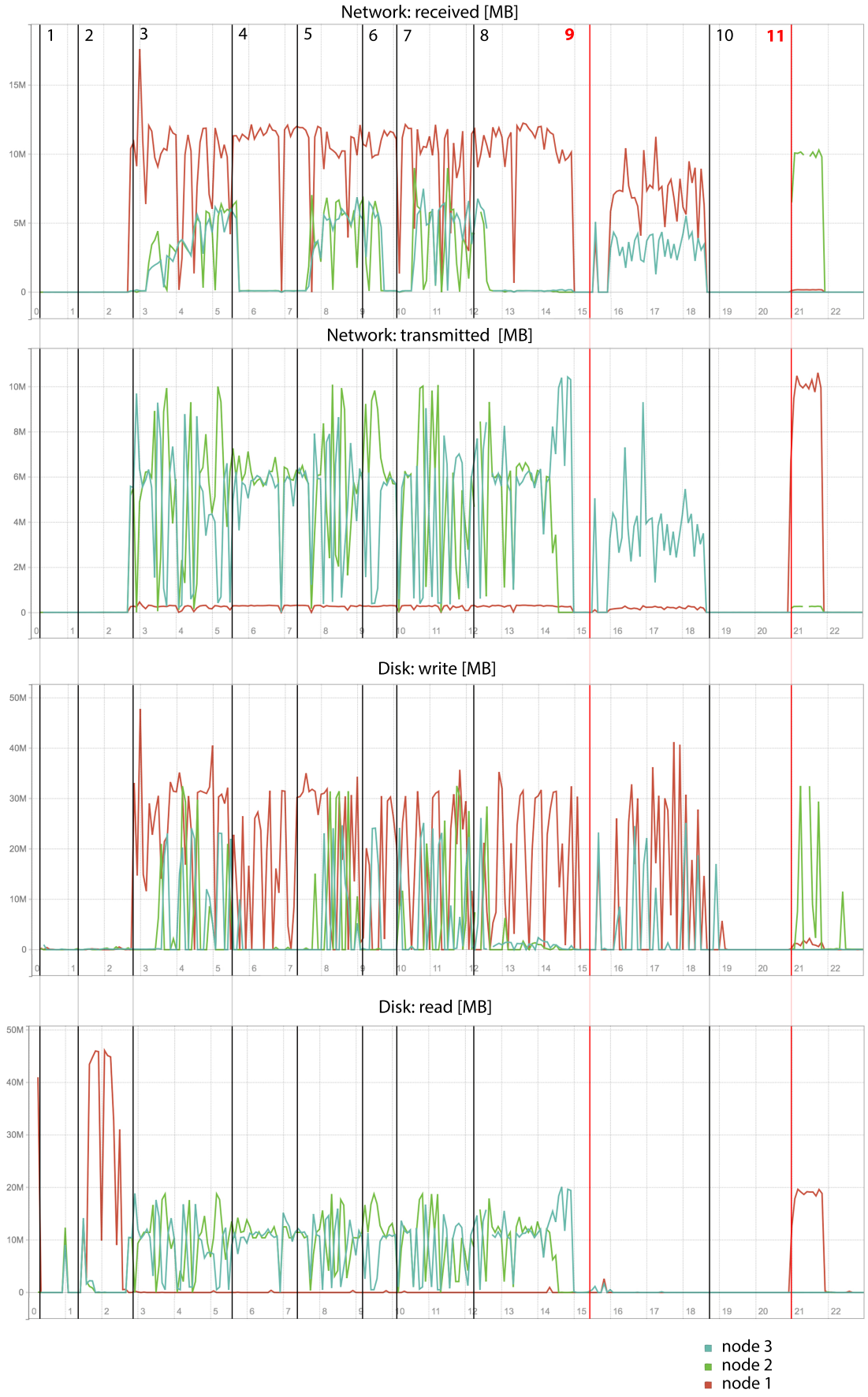

Figure 11.5: Kafka experiment A [cont.]

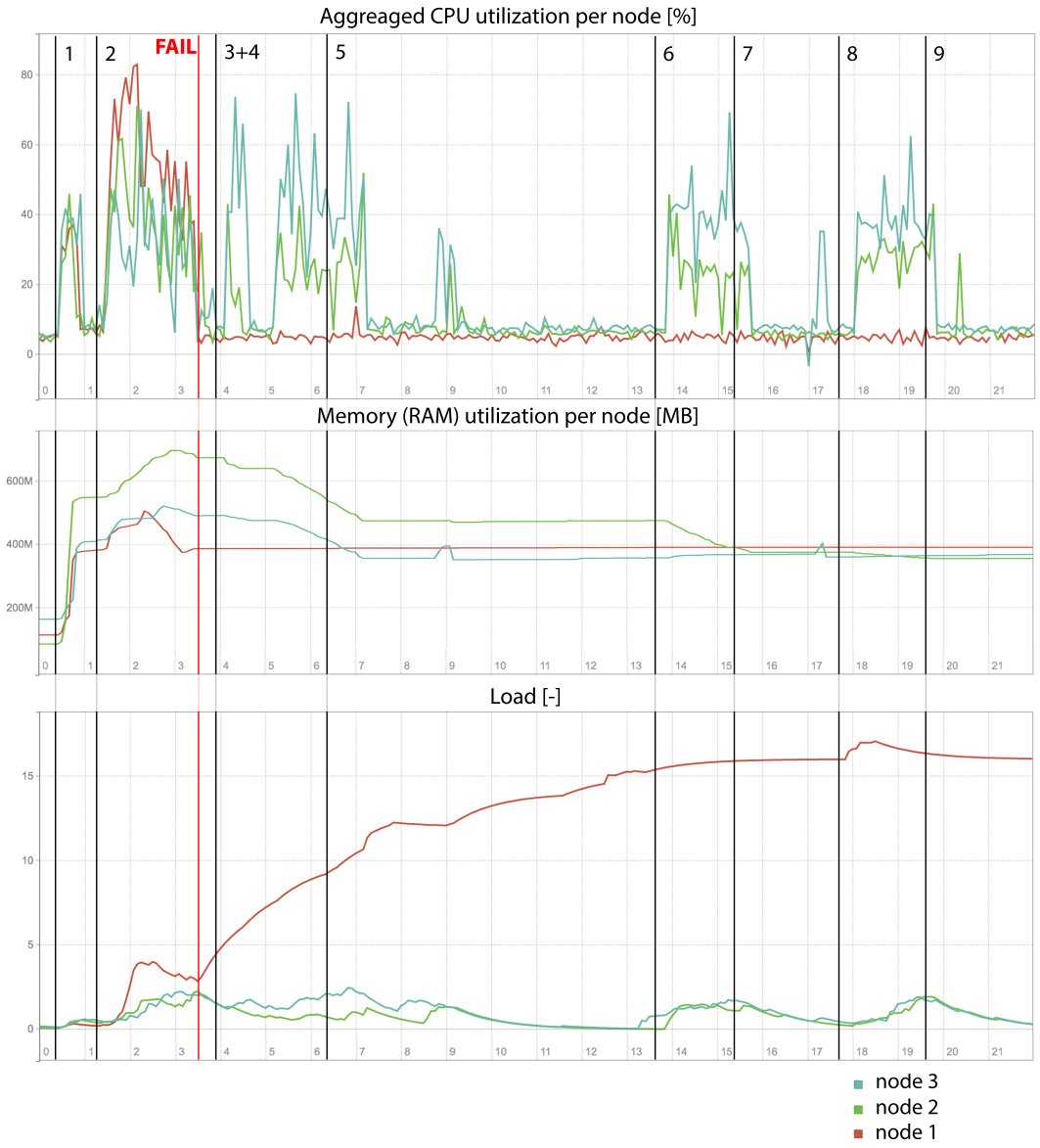

Figure 11.6: Kafka experiment B

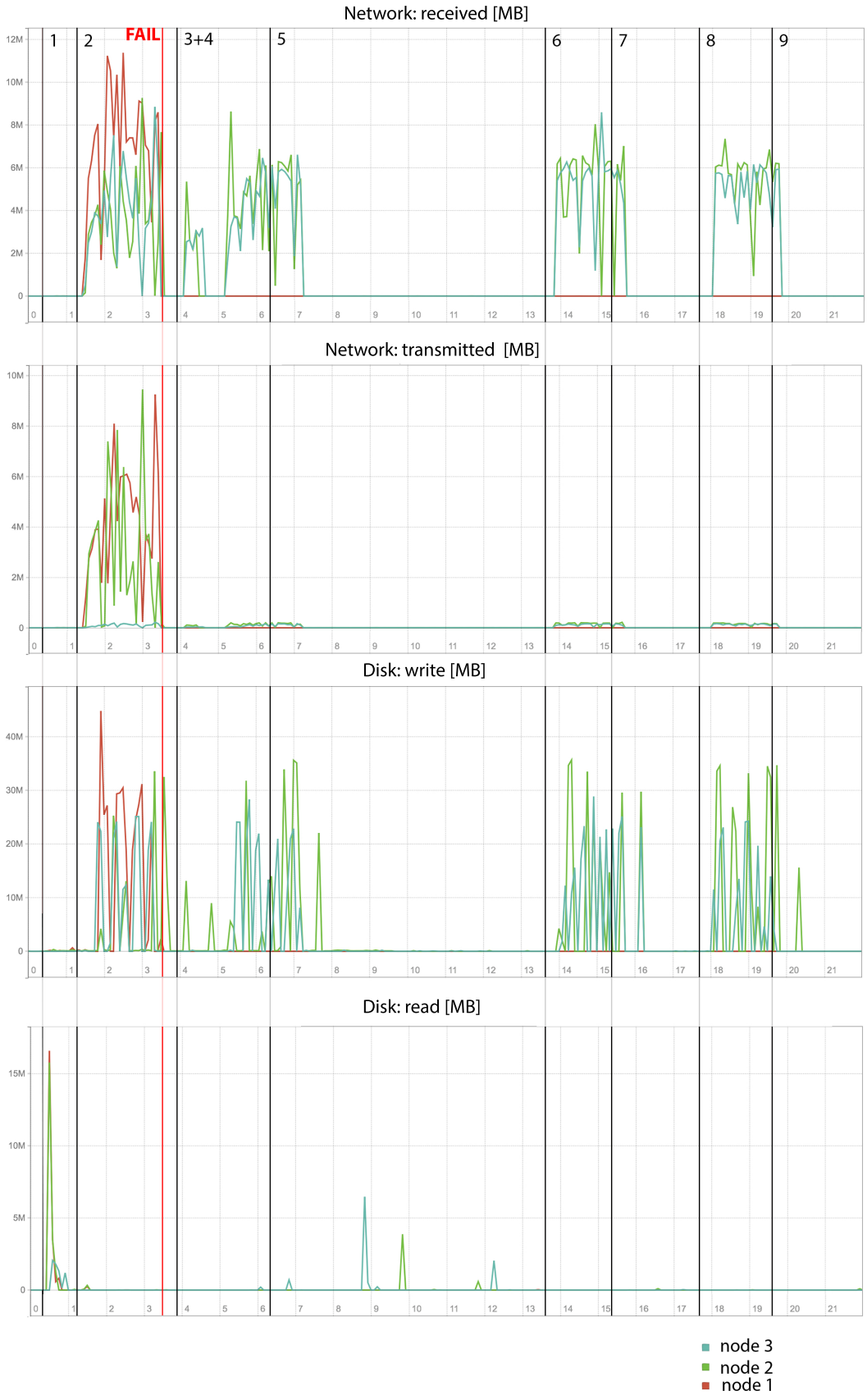

Figure 11.7: Kafka experiment B [cont.]

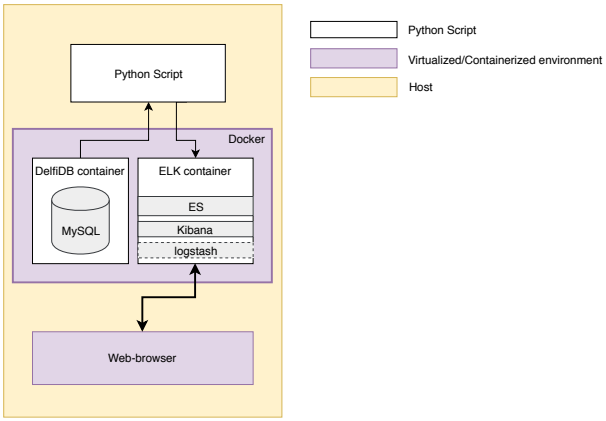

Figure 11.8: ELK latency experiment

affected by the number of data processing nodes, and depends only on the data source and sinks locations.

The second part of the experiment focuses on the performance of the processing. It does not address the research questions directly but serves as a measure for acceptance testing. As discussed in section 3, the frame processing rate (per satellite) observed in the Delfi-n3Xt mission is acceptable for the stakeholders. The experiment therefore focuses on the low-latency streaming data processing, as it has the highest priority of Satellite Operators.

Hypothesis: *Spark system can operate at Delfi-n3Xt peak ingestion rate*

The experiment setup requires the use of Kafka, and hence ZooKeeper, HDFS, and therefore YARN and Spark components that are shown in the figure 11.2. The data source, Kafka producer, is identical to the Kafka experiments and injects 1000 frames as a batch into Kafka topic (two partitions, two replica's) within 600ms. The data sink is ElasticSearch (ES) data store and Kibana UI used as a dashboard.

The cluster performance for the experiment is expressed as a measure of the delay between data submission by Kafka Producer and processed data appearance in the Kibana UI. To determine the processing time, the latency within ELK stack, docker virtual networking and storage has to be determined. The precursor experiment, illustrated in figure 11.8 determined that on average 18 seconds is required to transfer 1000 unprocessed frames from MySQL database to Kibana, which correlates to an average ingestion rate of 55.5 frames/s.

Figures 11.10 and 11.11 show the server utilization during two phases of the experiment: Job deployment and Frame processing. As defined in table 11.6, points 1 to 8 span the automated Spark Job deployment to the cluster. The deployment is managed by the Spark driver (Master) on node 1. The first step of the process is the preparation of job deployment to the cluster, that verifies the code, builds a deployment container and validates the cluster availability illustrated by points 1 to 3. In the experiment, sufficient resources are available and the job can be deployed, leading to the selection of executor (4) followed by establishing of the Spark Streaming Context (5) and Kafka consumer (7) per participating node. At point 8 the querying service is initiated, at this stage, Spark Job is ready to accept and process new data.

It should be noted that the processing job is executed by all nodes, with exception of the driver. However, since Kafka topic consists of two partitions, only two Spark executors can pull the data. This is clearly visible during processing phase 9 to 19 and 19 to 27 showing node 2 inactivity of the CPU 11.10.

As discussed in section 9.3 Spark Streaming process the data in micro-batches. Due to severe hardware limitations of the cluster in comparission to the [137], the performance of Spark is heavily affected allowing each micro-batch to be logged. The per batch results are summarized in table 11.5 spanning both runs. The processing speed is shown in figure 11.9. The average processing speed during the experiment is 9.4 f/s (STD 0.26s), based on the average experiments duration of 104 seconds and 1000 ingested frames. In contrast, the highest observed ingestion rate for the Delfi-n3Xt mission is 8 f/s per second as shown in figure 3.20.

As discussed in section 9.3 , Spark is heavily memory optimized, attempting to utilize all YARN allocated memory. This is supported by low disk transactions and increased memory utilization as

Table 11.5: Stream Processing speed on RPI cluster

Run A			
Point	Eq. Latency[s]	Frames in Kibana	Processing speed [f/s]
10	3	8	2.7
11	20	18	1.1
12	33	143	11.0
13	43	264	26.5
14	53	502	50.2
15	63	624	62.4
16	68	744	148.8
17	88	865	43.3
18	103	1000	66.7

Run B			
Point	Eq. Latency [s]	Frames in Kibana	Processing speed [f/s]
19	1	9	15.0
20	31	134	4.5
21	41	255	25.5
22	51	372	37.2
23	58	495	70.7
24	61	617	205.7
25	68	735	105.0
26	90	858	39.0
27	106	1000	62.5

Table 11.6: Point description for figures 11.10 and 11.11

Point	Description
1	Job submitted
2	Job container prepared
3	YARN: job accepted
4	Job allocated
5	Spark Streaming Context started
6	Spark proxy started
7	Kafka Consumer started
8	Quering started
9	Data added to Kafka topic
10	First data data in Kibana

shown in graphs 11.10 and 11.11 during data ingestion (points 9 to 18 and 21 to 27). While node memory is not fully utilized on all nodes, the available Swap memory is zero on nodes 1 and 2. This means that nodes 1 and 2 are out of RAM requiring OS to use the disk for memory purposes, thus critically reducing the performance. This seemingly unlogical situation is easily explained. First, 550MB is allocated per Spark executor for the processing job, the minimum allocatable memory for given Spark architecture. In this case, the memory is used to load data from Kafka and perform computations. Since the computations are memory extensive the major portion of the allocated memory is empty and unutilized. But, since it is allocated, memory is unusable for OS or other application. The rigid memory allocation is paramount for the robustness, as each processing job is ensured to have access to the allocated resources. Furthermore, this prevents a malfunctioning or buggy job from crashing Spark or other application on the server due to memory or CPU overutilization. Unfortunately, this may come at cost of memory waste as shown in the experiment.

The experiment is continued with additional runs C and D that can be found in Appendix B. While the processing performance and perceived batch sizes are very similar, the load trend of the cluster is drastically different as shown in figure 11.12. Comparing the load figures 11.11 and B to the Kafka ingestion timestamps, it is clear that the cluster load is not affected by the data ingestion, nor actual frame processing. This is a clear indication of a severe resource shortage.

11.6. Experiment Results Discussion

The ingestion rate at worst-case scenario surpassed the expectations. (More tests will be conducted in the next two days.)

The experiments did not prove the hypothesis of a reduced server load by data replication. The instantaneous server load due to query execution is comparable to replication load but much shorter in duration. A query pulling 3000 frames has an average runtime of 35 seconds, while replication of similar dataset would require 2.4 minutes. It should be stressed that once replicated, the queries can be performed on the client side without further impact on the server operations. Therefore it would be ideal for the workloads requiring frequent data transversals such as machine learning model training and analytics.

In case of Delfi missions, the processing of telemetry frames has a low computational impact. The entire Delfi-n3Xt dataset can be decoded and processed in minutes, while replication of 220,000 frames to a single connected client would take approximately 2.9 hours.

The experiments focused on small (182 % smaller than Delfi-n3Xt) datasets, therefore placing data directly in memory, hence providing the best possible performance. Even under these conditions, the performance is lacking, which can be arguably attributed to the selected hardware. However, validation

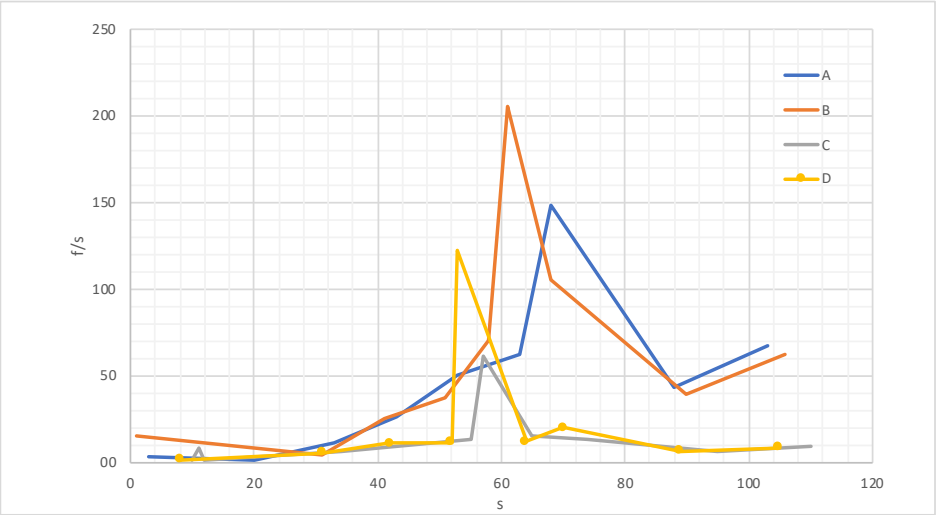

Figure 11.9: Processing Speed and Latency as measured in Kibana

runs on a high-end Windows machine did show a limited increase in the replication performance for CouchDB system. A number of optimizations can be designed, such as data compression to increase of the batch size for each replica shard. However, the load to performance ratio is high and much higher than querying or frame processing. Furthermore, placing clients outside the managed network, as was the case in the experiment, will degrade network performance and introduce uncertainties, i.e. frame drops, routing errors etc.

Based on aforementioned points, it was concluded that bringing data to processing nodes, i.e client machines, is computationally expensive when database replication is used. Different methods of data delivery to the client applications can be considered. Regardless, networking will become the bottleneck and with limited computational requirements processing server side is both faster and more robust.

The experiments showed the incredible robustness of the processing system, allowing data processing under heavy resource shortage with performance similar to the peak data ingestion rate observed during the Delfi-n3Xt mission.

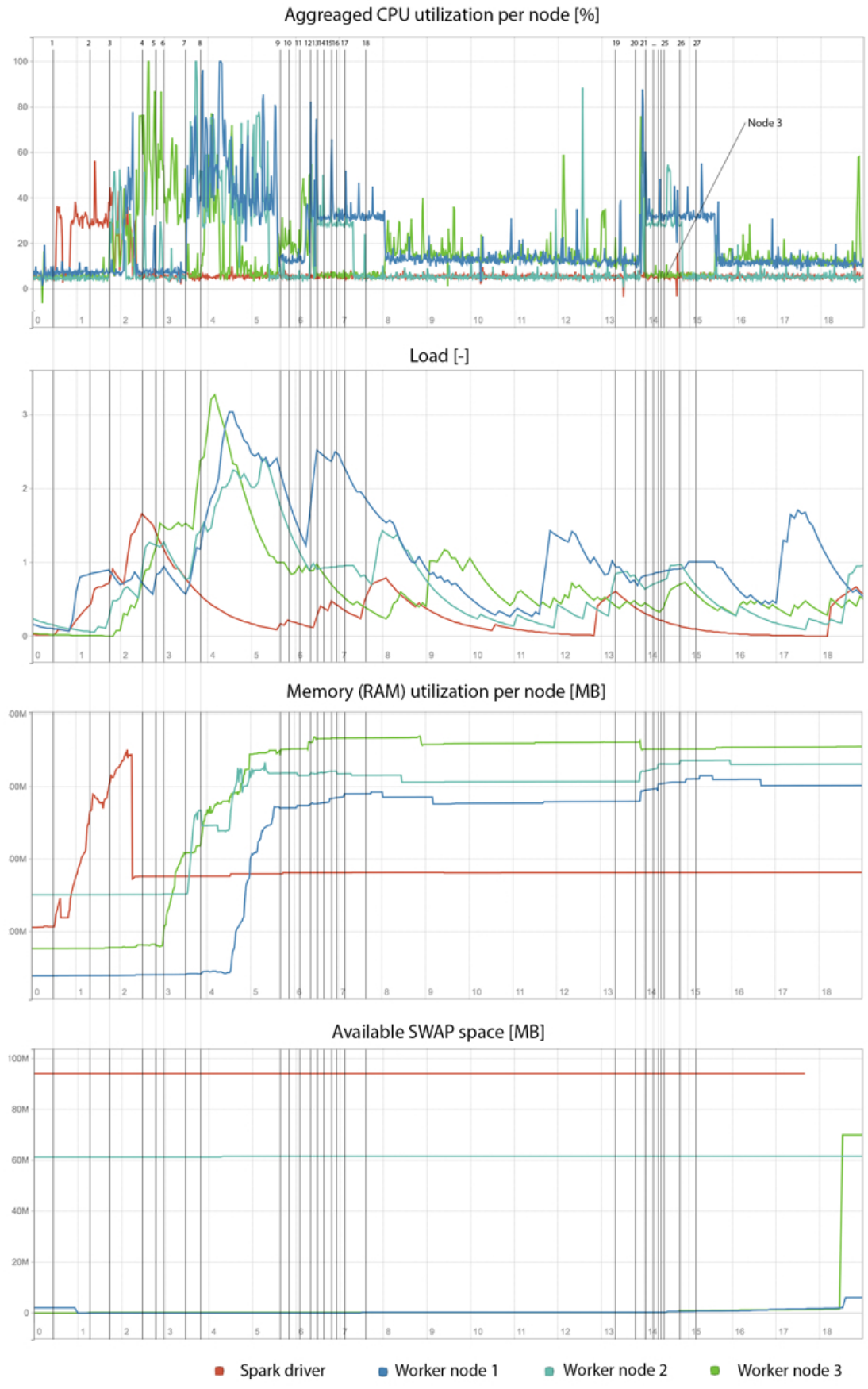

Figure 11.10: Spark Streaming experiment

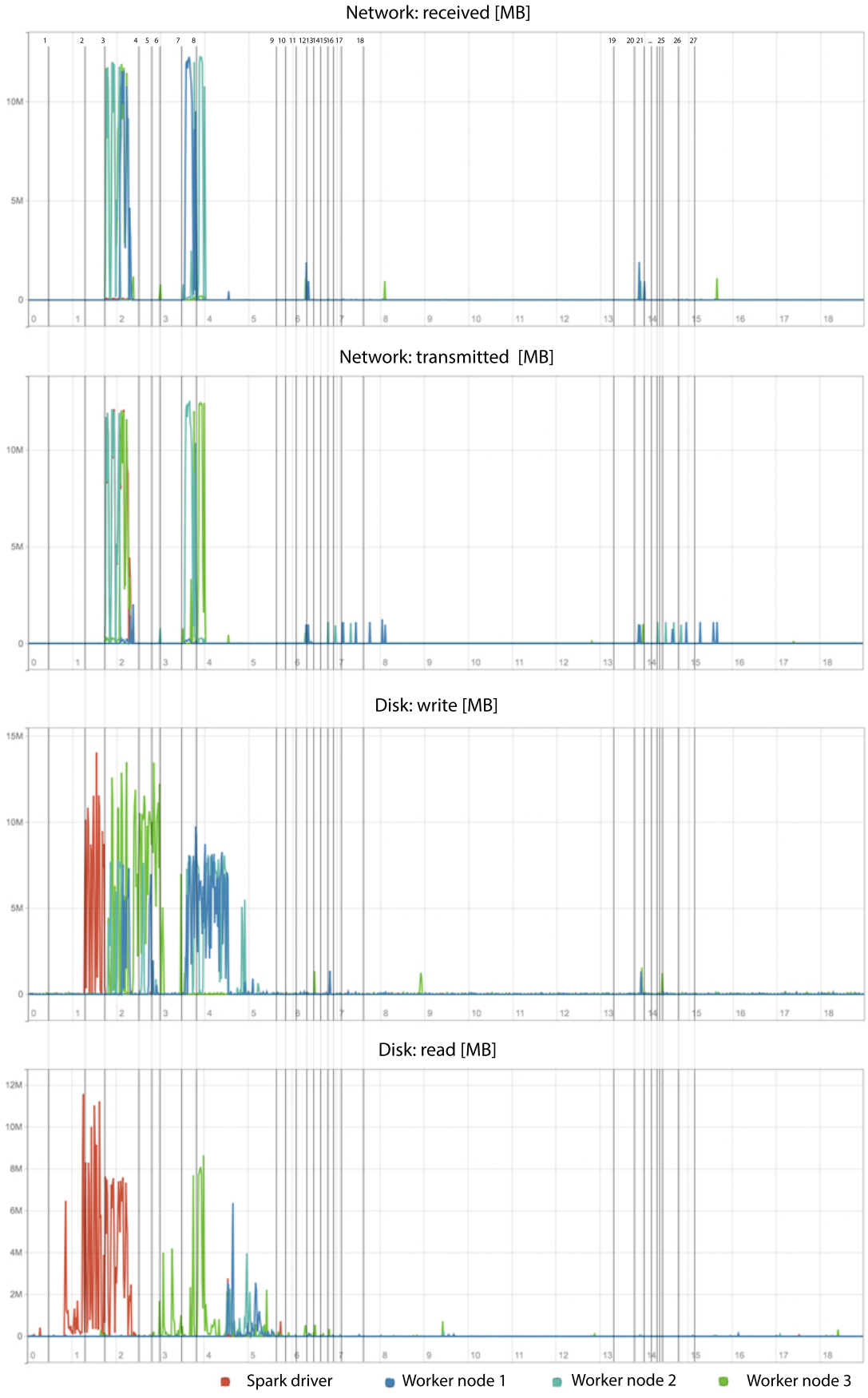

Figure 11.11: Spark Streaming experiment [cont.]

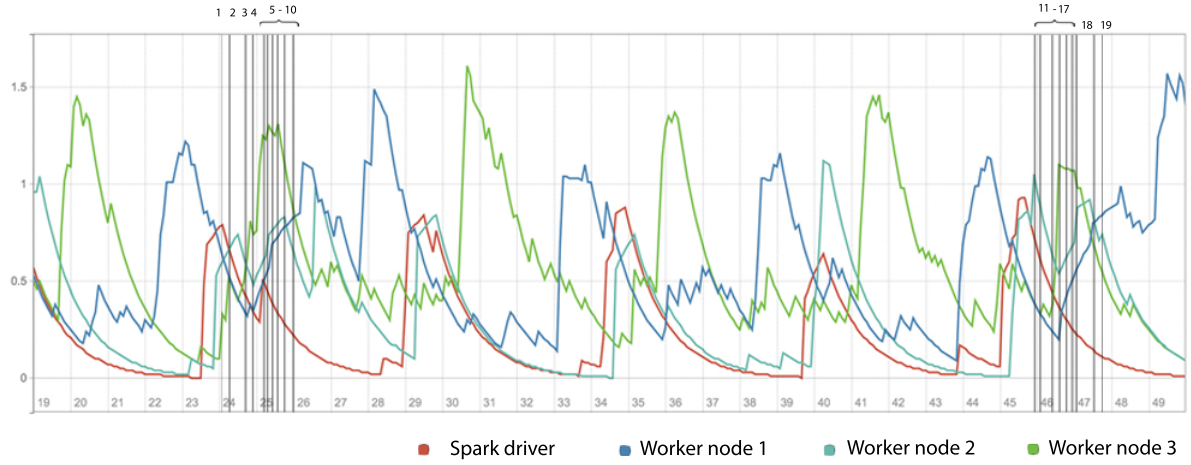

Figure 11.12: Cluster load during runs C and D

12

Conclusion

The telemetry processing system provides the telemetry storage and processing needs for the day-to-date satellite operation and researchers utilizing the stored data for research purposes. The legacy systems, as discussed in chapter 3, showed limitations on both storage and processing functionality which combined with the increased capabilities of the next generations Micro and Pico satellites reduce the reusability and fail the specified requirements. The research of the available COTS applications led to a strong preference for in-house developed systems due to higher level suitability and compatibility, discussed in the project precursor literature study(see appendix C).

At the early stage of the project, two primary areas of interest have been selected for the further research: data storage (chapter 4) and data processing (chapter 5). In order to comply with the operational requirements for the availability, reliability and robustness, the distributed deployment of the storage system was identified as the most feasible solution. This, furthermore, complied with the unbound volume of data, and ingestion data rate requirements (appendix A).
The data processing research lead to the conclusion that in the legacy applications, the processing errors can be accredited by the lack of requirements on the telemetry definition. Primarily due to the assumption of a single telemetry frame definition (section 3.5). Furthermore, due to lack of standardization, processing errors are likely both in the client and server applications. This lead to the concept of unified processing, a language-agnostic processing system with a high-level telemetry frame definition introduced in section 5.2.

To facilitate the distributed deployment of the storage system, two options were considered and discussed in chapter 7: scaling on server-side (multi-server deployment) and scaling to client applications. Due to the correlation between system load and the number of the connected client applications, chapter 8 investigated an approach to source the server-side operations to the swarm of client applications. Unfortunately, early on, the implemented PoC showed a high degree of operational complexity. More importantly, and regardless on the implementation, the networking between server and the client affected the data replication rate which was determined unacceptable for production and failed the established requirements. Aforementioned rendered the client-application feasible primarily for the computationally expensive operations and data back-up purposes. Bound by the CAP theorem, the distributed storage system is eventually consistent and affected by the client-server networking. Therefore, a high degree of data consistency repairs is required even for the backup purposes of the operation.

The failure of client leveraged PoC to comply with requirements led to more in-depth research into the software engineering and further refinement of the requirements, aggregated in chapter 2. The study led to the conclusion that the system requirements for the future mission(s) cannot be defined precisely at this stage of the project and are, therefore, likely to change. This lead to a change in methodology from the search for new requirements to the design of a system that can cope with the requirements change.
Furthermore, to fit with Delfi-mission design methodology, the system should consist of highly de-

coupled components providing high-level abstractions and allowing each component to be designed, implemented and tested independently. Thus, both allowing the future (student) developers to complete implementation within the allocated time and giving the opportunity to select the best fitting tools for the subsystem.

These requirements blueprint a distributed system consisting of highly decoupled and exchangeable components. Two architectures have been presented in chapter 9: Streaming and Lambda. Streaming Architecture failed to comply with data de-duplication requirement that inherently resulted in inconsistencies in the data delivery logic. The proposed Lambda Architecture complies with set requirements and solves the data de-duplication problem by splitting processing into two: batch and stream. Batch layer ensures the consistency of the results, however at the cost of the processing latency. The stream processing provides the near real-time processing, but at the expense of consistency. By balancing both, Lambda Architecture implicitly ensures Consistency of the results, Availability and Partition tolerance (CAP).

For a system to be reliable, thus the opposite of being failure-prone, the system should accommodate high Availability and Testability on short-term, while facilitating the Maintainability for the duration of the mission(s). All of which are provided by Lambda Architecture from the ground up.
The system robustness, i.e. the ability to cope with error, inconsistencies and unpredicted input, is achieved by combining the unified processing with a high level of decomposition, allowing each component to be tested separately in unit-test like manner.

In short, the proposed architecture (chapter 10) ensures the compliance with the existing requirements, and with the components interchangeability accommodates for future requirements that are unknown at the time of writing. With the high level of abstraction and one-way dependency, each component can be designed independently, facilitating software quality.

Furthermore, using the proposed architecture, a wide variety of queries, analytics computations and machine learning algorithm can be deployed in conjunction, scheduled and executed independently. With redundant data ingestion and processing components, Lambda Architecture is robust not only to hardware failures but also to human-induced errors, both in programming bugs as to the majority of the operational mistakes (configuration, networking) as discussed in chapter 11.

In authors personal opinion, the proposed architecture is the best fit for the current and future Delfi operations and is most likely applicable to the other university grade cubesat missions. The use of HDFS provides the ability to scale up to store thousands of petabytes worth of data, and ingest and process data at tens of gigabytes per second rates, comfortably accommodating the majority, if not all, university grade satellites. More importantly, the proof of concept on underpowered, below specifications hardware, showed that the system does not require expensive, purpose-designed equipment for operations and running inexpensive low-end servers would suffice for any form of service.
With a high level of abstraction and use of state of the art processing technologies, the architecture allows sophisticated data mining and analytics without the need for system redesign or custom hardware. If more processing is required, new (virtual) servers can be added and then removed the cluster, without system downtime and with minimal reconfiguration.

The extreme flexibility of Lambda architecture comes at the cost of complexity. Due to distributed deployment, additional technologies, such a Yarn and Zookeeper are utilized for cluster operations. The complete cluster operation requires fine tuning when subsystems are deployed in parallel on the same machines,(Kafka, HDFS + Spark, Delivery layer). A resource waste can be observed in the processing, as each telemetry frame is processed at least twice in Stream and Batch layers. To prevent delivery layer data duplication, the Batch should iterate large if not complete dataset on each execution. By design, the delivery layer data is overwritten, to remove the possible processing bugs and frame duplications.

In conclusion, the proposed architecture offers flexibility in operations, providing robustness to human errors and hardware failures. Furthermore, with use of a dedicated processing framework (Apache Spark), the processing system is abstracted and can be scaled from one to thousands of machines without code modifications. Multiple processing jobs can be running in parallel while being monitored by YARN. This greatly reduces the SysAdmin responsibilities and while providing an intuitive

processing monitoring and debugging tool. Furthermore, due to the simplicity of data processing, the implementation requires less than 50 lines of code and drastically reducing the risk of bugs and increasing the readability.

13

Recommendation

The use of underpowered hardware cluster proved that BigData oriented system can be used for SmallData applications. However, operational test on the actual hardware is required to benchmark the expected real-world performance. More specifically, the effects of partitioning, such as number partitions and replication factor of Kafka has to be investigated. As discussed in [Prop architecture] chapter, the number of Kafka partitions defines the maximum number of parallel Spark executer, hence capping the performance. It is expected that the overpartitioning, as frequently done in NoSQL systems, will reduce the performance, as more data replication will be required.
Secondly, due to the low performance of the test setup, the deployment time of Spark Jobs could not be estimated for the target deployment. This is important since any job updates require job restart and redeployment on the cluster, during which no processing can take place.

The proposed architecture and the proof of concept utilized a generic delivery layer as a tool to benchmark the processing system. Further analysis and study are required to design the data visualization and user interface.

The unified processing is achieved with the use of Kaitai framework, that converts YAML formatted files to the target language executables. The use of custom YAML protocol for processing definition is acceptable, but a more widely utilized protocol would be preferred. For example, Telemetry and Command Exchange protocol (XTCE) [138]. XTCE is XML based, but with a number of XML-YAML converters available, most notably YAXML [138] a YAML- XML binding, allowing YAML structures to be expressed in XML language and vice-versa. Further research is required to investigate the conversion feasibility.

Another aspect of processing that could not be investigated during the project are the real-time updates of Spark Job. It is assumed that encoding auto-update functionality can modify spark Jobs by loading the processing binaries from an external HDFS directory. This requires further investigation to ensure the robustness.

With regards to the methodology, the use of iterative design led to inconsistencies in the development process. More specifically, the abrupt stop of PouchCouch iteration that led to BigData inspired architectures. While acceptable in terms of the delivered result, i.e. more applicable architecture, the discontinuation could have been avoided with a more refined ”Objective” (the first part of the iteration). From a larger perspective, the principle applied to the entire MSc thesis, leading the recommendation to focus more on the planning than execution. More specifically, what is classically defined as the Literature Study made its way into the primary work of the thesis (execution), leading to this bulky document and a more theoretical (guidelines and design) than a practical PTS implementation. In other words, the Literature Study topic should have been more in line with the main thesis project.

A

Requirements

A.1. Introduction

The purpose of this section is to provide set of software system requirements for the minimum viable product of the Telemetry Processing System (TPS). The section is written with goal to provide the background for design decisions and covers the scope and the functional, and non-functional software requirements. At the time of writing no significant system level constraints have been observed.

The scope of the project is to provide a solution for the telemetry data storage, processing and visualisation. Due to time constraints, the scope has been limited to the ingestion, processing and querying components.

A.2. Product Description

A.2.1. Product Perspective

The TPS is a standalone system that provides functionality outlined in Product Functions section and ought to fullfill the requirements described in teh Software Requirements sections. The TPS has interfaces for external client applications (ECA), the definion of requirements of which are out of scope of the document.

A.2.2. Product Functions

The TPS provides five major functions: Data ingestion, Data storage, Data processing, Data querying and Data visualization. The data visualization is coupled with data querying, but is out of scope of the thesis. The five respective subsystems are considered stand-alone and ought to be designed indepently, intefacing only in the following order: Data ingestion to Data storage, Data processing to/from Data Storage, Data querying to Data Storage, Data visulatization to Data querying. The interfaces, detailed definition and the architecture of the aforementioned components are discussed in the chapter 9 and 8.

A.2.3. Product Constraints

The TPS ought to be installed and maintained on-premise of TUD. This means that ready-to-use, highly scalable and efficient FaaS services are unavailable. Furthermore, TBD version of Linux and TBD server version of Windows Server are available.

A.2.4. Dependencies

The TPS sub-systems are components ought to comply with available operating system: Linux TBD version.

A.2.5. Assumptions

~~The TPS systems relies on the participation of Radio Amateur community for satellite data reception and processing.~~ The designed system is designed with an assumption of higher data ingestion rates than observed in the legacy missions. It is therefore assumed that the future missions will generate

more data [GN-AS-1] due to use of the on-board S-Band transeivers [GN-AS-2], and decreased cost of the radio-equipment [GN-AS-3] higher participation rate of the radio amateur community, will lead to higher data ingestion rates than previously observerd. Furthermore, it is assumed that the sattelite telemetry frames are independent, and can be processed arbitrarily out of sequence [GN-AS-4].

A.3. External Interface Requirements

A.3.1. User Interface

GN-UI-GEN-01 The user interface should provide an interface specific to the user type [RA, Operator, Admin, Data scientist]

GN-UI-GEN-02 The user interface should visualize datasets in TBD format.

GN-UI-GEN-03 The user interface should provide functionality to filter datasets based on the spacecraft and spacecraft's parameters.

GN-UI-GEN-04 The user interface should indicate the dates and time of the next Satellite pass for a given location.
Rationale: Spacecraft operators require information on the next spacecraft downlink and uplink opportunity.

GN-UI-GEN-05 The user interface should require user to authenticate using TBD method.

GN-UI-GEN-07 The user interface should allow user to change email address and the password requiring TBD minimum complexity.

GN-UI-GEN-08 The user interface should only allow UTF-8 (ISO-8859-1 standard) characterset for the input field.

GN-UI-RA-01 The Radio Amateur user type should be able to access the statistics of the submitted data: total frame count, number of unique frames (received by RA only) and ranking based on the total frame count.

GN-UI-RA-02 The user interface should allow Radio Amateur type users to review the locally received telemetry frame.

GN-UI-RA-03 The user interface should require Radio Amateur to specify the location as latitude and longitude in numeric format with "." as decimal separator.

GN-UI-RA-04 The user interface should allow Radio Amateur to specify radio call sign, only allowing a-z, A-Z and 0-9 UTF-8 based characterset.

GN-UI-OPS-01 The user interface should allow Spacecraft Operators to review the data stream from the satellite in GN-UI-GEN-2 format, with maximum allowed TBD seconds latency.

GN-UI-OPS-02 The user interface should allow Spacecraft Operators to retrieve the historic satellite data on demand, with ad-hoc defined ranges.

GN-UI-OPS-03 The User interface should visually indicate the spacecraft parameters beyond accepted ranged for Spacecraft Operator user types.

GN-UI-OPS-04 The user interface should allow Spacecraft Operators to review the telecommands sent to the spacecraft.

GN-UI-OPS-05 The application should offer TBD parts of the user interface to Spacecraft Operators with system downtime.

GN-UI-OPS-06 The user interface should provide Spacecraft Operators with information required to determine the on-board clock drift and provide functionality to store the clock offset.

GN-UI-OPS-07 The user interface should provide a visual aid for Spacecraft Operators to determine the spacecraft position in orbit, with use of TLE.

GN-UI-ADMIN-01 The administrator user should have access to the user interface allowing Create, View, Update and Delete operations for the user accounts.

GN-UI-ADMIN-02 The administrator user interface should provide password reset functionality for all user types.

A.3.2. Hardware Interface
TBD hardware interface.

A.3.3. Software Interface
TBD external interface with the client application.

A.3.4. Communication Interface
~~**GN-NET-01** Each system node must have at least one public network adapter.~~
~~Rationale: Each node of distributed system should be accessible from each other node.~~

~~**GN-NET-01-rev1** Each load balanced node must have at least one private network adapter.~~
~~Rationale: Load balanced nodes are addressed via the load balancer, and should not be publicly accessible~~

~~**GN-NET-02** Each load balanced node must have at least one least one public network adapter and at least one private network adapter.~~
~~Rationale: Load balanced nodes are addressed via the load balancer, and should not be publicly accessible~~

GN-NET-01-rev2 The system nodes required for the external client application functionality should have at least one publicly accessible network interface.

GN-NET-02-rev1 The system nodes not required for the external client application functionality should not be accessible via the public network.

GN-NET-03 The system nodes should support TCP/IP.

GN-NET-04 10 Gigabit connection speeds is prefered for the intra-cluster node connectivity.

A.4. System Features
A.4.1. Data Ingestion
GN-ING-01 The system shall expose an REST-based API for client-server communication.

GN-ING-02 The subsystem providing API shall expose a TBD authentication interface.

GN-ING-03 The API shall allow ingestion of binary-encoded data, regardless the encoded format.

GN-ING-04 The encoding of a specific binary-encoding shall not drive API design.

GN-ING-05 The API shall facilitate concurrent data ingestion (from multiple sources).

GN-ING-06 The API shall require client application authentication prior to data ingestion.

GN-ING-07 The API shall be reusable and independent of the server or client implementation, data storage, processing or data delivery subsystems.

GN-ING-08 The API shall utilise a stateless protocol.

GN-ING-09 The API shall be independent of the mission and mission parameters.

~~**GN-ING-10** The API system shall provide interface for data filtering.~~
Rationale: Filters increase chance of human-errors, discarded frames are unrecoverable.

~~**GN-ING-11** The ingestion system shall be scalable Rationale: Increases data inflow.~~
Rationale: non-functional requirement, and superseeded by GN-PERF-6

GN-ING-12 The data ingestion system shall facilitate growth in data volumes.

~~**GN-ING-13** The API system shall provide interface for data filtering.~~
Rationale: duplication of GN-ING-10.

GN-ING-14 The API system shall provide ability to block communication from blacklisted hosts.

GN-ING-15 The ingestion system shall provide ability provide ability to buffer data preventing processing system overload.

GN-ING-16 The ingestion system shall store ingested data for TBD period of time.

A.4.2. Data Processing

GN-PR-01 The processing system shall be independent of ingestion, storage and data delivery subsystems.

~~GN-PR-02~~ ~~The processing system shall be scalable, to facilitate WOD datalink~~
Rationale: Superseded by GN-PERF-6

GN-PR-03 The system shall provide an interface to LSB sort the incoming stream.

GN-PR-04 The system shall provide an interface to scan for the AX.25 flags.

GN-PR-05 The system shall provide an interface to perform frame validity checks (CRC).

GN-PR-06 The system shall provide processing capabilities to perform frame parameter parsing.

GN-PR-07 The system shall provide interface to perform frame decoding.

GN-PR-08 The system shall provide an interface for processing script modifications.

GN-PR-09 The system shall provide consistent results

GN-PR-10 The system shall provide ability to reprocess all historic data.

~~GN-PR-11~~ ~~The system shall provide data collision prevention and data conflict mitigation strategies.~~
Rationale: vague, and replaced by GN-PR-19

~~GN-PR-12~~ ~~he system shall utilise an unified parsing subsystem both on the client as server application(s).~~

GN-PR-12-rev1 The system shall utilise an unified telemetry frame parsing and decoding subsystem.

GN-PR-13 The system shall utilise identical frame definitions for client as server data processing.

GN-PR-14 The parsing subsystem shall support flexible telemetry frame definitions.

GN-PR-15 The parsing subsystem is able to identify telemetry types (satellite, frame type and version).

GN-PR-16 The parsing system shall facilitate multi-satellite missions.

GN-PR-17 The parsing system shall facilitate satellite specific processing techniques.

GN-PR-18 The parsing system shall be extandable to support non-AX.25 based protocols.

GN-PR-19 The parsing system shall aggregate duplicate ingested data, by keeping a statistical count of received frame as well the originating clients.

GN-PR-20 The processing system should support processing of the partially correct DF.

GN-PR-21 The processing system shall support custom processing scripts.

~~GN-PR-22~~ ~~The data should be processed in binary matter.~~
~~Rationale: binary processing is efficient and much faster than string processing.~~

GN-PR-22-rev1 The data should be processed in TBD format.
Rationale: The binary telemetry data should not be stored in String format.

GN-PR-23 Processing system should be able to decode DF with directly in the frame embedded schema.

GN-PR-24 Processing system should be able to decode DF with externally (e.g. configuration file) defined schema.

GN-PR-25 Processing system should allow (bulk) data post-processing.

GN-PR-26 Processing system should facilitate post processing for error-correction, by combining historic data.

GN-PR-RAW-01 Single raw frame to satellite parameters

GN-PR-RAW-02 Determination of satellite location based on TLE

GN-PR-RAW-03 Complete dataset scan, retrieval of list without duplicates, processing one by one to satellite parameters

GN-PR-INT-01 Retrieve the TLE from the online store (every day, twice a day), store TLE in original text format

GN-PR-STAT-01 Complete dataset scan, determine amount of duplicate data frames

GN-PR-STAT-02 Complete dataset scan, determine amount of frames per RA, dataloss per RA and other per RA statistics

GN-PR-CL-01 Satellite parameters random-read queries

GN-PR-CL-02 Out of boundaries values queries

GN-PR-CL-03 Complete dataset scan, show the past 24h data reception statistics

A.4.3. Data Storage

GN-DS-01 The storage system shall store all incoming data redundantly.

GN-DS-02 The storage system shall facilitate for the data backup.

GN-DS-03 The storage system shall store data in a structured matter.

GN-DS-04 The storage system shall store data in an aggregated format, accessible for the data science applications.

GN-DS-05 The storage system shall not be directly accessible via external networks.

GN-DS-06 The storage system shall facilitate unbound data growth.

GN-DS-07 The storage system shall have a storage capabilities exceeding 1 TB.

GN-DS-08 The storage medium shall be provided and managed by TUD.

GN-DS-09 The storage system shall provided dedicated storage for satellite data, separated from user data.

GN-DS-10 The data storage format shall be exchangeable with other systems. (Use of Apache AVRO suggested)

GN-DS-11 The storage system shall facilitate concurrent data querying.

GN-DS-12 The storage system shall allows structured and unstructured data storage.

GN-DS-13 The storage system shall facilitate querying of the unstructured data.

GN-DS-14 The storage system shall allow payload data separation for direct stakeholder access.

GN-DS-15 The storage system should be able to store and mark partially correct data.

GN-DS-16 The structure (schema) of telemetry frames is adjustable without system downtime. Prefably on-write.

A.4.4. Data Querying

GN-DQ-01 The querying system shall provide random data query capabilities.

GN-DQ-02 The querying system shall execute recurrent queries capabilities with a higher performance.

GN-DQ-03 The querying system shall provide an interface for data science querying spanning the entire data set.

GN-DQ-04 The querying system shall expose a vendor independent querying language.

GN-DQ-05 The querying system shall utilise actively developed, supported and non-deprecated technologies.

A.4.5. Data Delivery

GN-DD-01 The delivery system shall provide a public Web interface

GN-DD-02 The delivery system shall provide a TBD privatatly accessible interface for data querying.

GN-DD-03 The processed data shall be submit to Spacecraft operators with latency < 1 sec.

GN-UI-GEN-06 The web-based portion of the interface should be delivered using HTTPS protocol with a cipher key size larger than 128 bits.

GN-UI-GEN-06 addendum 1 The web-based portion of the interface should be delivered using HTTPS protocol with a cipher key size larger than 128 bits. Anonymous cipher suits as well SSL 2.0, SSL 3.0, TLS 1.0 and TLS 1.1 protocols should be blocked.

A.5. Non-functional Requirements

A.5.1. Performance Requirements

GN-PERF-01 The system shall have response time of less than 500ms within server timezone.

GN-PERF-02 The system shall provide a minimum data ingestion throughput of 10 telemetry frames per second per satellite.
Rationale: During Delfi-n3Xt operations maximum througput of 8 frames/second was observed.

GN-PERF-03 System should be able to reprocess the complete historic dataset in 6h period.
Rationale: Reprocessing should be agile, allowing frequent data corrections.

GN-PERF-04 The maximum allowed latency between data ingestion by the client applicaiton and delivery to the Spacecraft Operations within one second.

GN-PERF-05 ~~System shall facilitate both scale-up (veritcal) and scale-out to comply with GN-PERF requirements, both server- and network-wise.~~ Rationale removal: Implicitely defined by GN-PERF-3, GN-PERF-6 and GN-PERF-7.

GN-PERF-06 System shall facilitate continuous growth of the stored data.

GN-PERF-07 System shall not place limiations on the data storage volume.

A.5.2. Safety Requirements

GN-SAFE-01 System shall prevent data loss. (Preferably by providing data redundancy)

GN-SAFE-02 System shall facilitate recovery from incorrectly decoded data.

GN-SAFE-03 System shall facilitate recovery from a server failure.

A.5.3. Security Requirements

GN-QA-SEC-01 The network security, such as connection availability, DOS, DDOS, flooding, network protocol and hardware vulnerabilities will be addressed by the service provider.

GN-QA-SEC-02 The server software updates will be handled by the system administrator.

GN-QA-SEC-03 The private user data retention period will be set in accordance to the educational GDRP law.

GN-QA-SEC-04 The private user data, as name, user location and username shall not be made available, unless specifically agreed on, to the Spacecraft Operators and Radio Amateurs.

GN-QA-SEC-05 The system should allow private user data, such as user uuid, to be separated from the telemetry stored information.
Rationale: each received frame contains uuid of the client, client should be able to request this information to be removed without deleting the telemetry data itself

GN-QA-SEC-06 System shall facilitate software updates.
Rationale: System update and upgrades should not force to redesign system components.

GN-QA-SEC-07 Data sent between server and client application is encrypted.
Rationale: User password shall not be sent unencrypted.

GN-QA-SEC-08 The system should throttle API for data ingestion and retrieval.
Rationale: Limit system load and prevent DOS attack.

GN-QA-SEC-09 The system should queue API request and remove old requests.
Rationale: Required to comply with GN-PERF requirements.

A.5.4. Quality Attributes

GN-QA-REL-01 The maximum allowed mean time to repair (MTTR) of system hardware is TBD hours.
Rationale: System downtime due to hardware failure requires manual intervention.

GN-QA-REL-02 Each subsystem should tolerate single node failure.

GN-QA-REL-03 A subsystem downtime should not downtime of the complete system.

GN-QA-REL-04 A subsystem should be testable. With preference for the systems allowing for independent testing.

GN-QA-AVAIL-01 The system shall provide data storage capabilities, even when main storage application is offline.

~~**GN-QA-AVAIL-02** The system shall reamain available under hardware failure~~

GN-QA-AVAIL-02-rev2 The system shall provide storage functionality for the incoming data as well TBD set of functions, provided by the client application.

GN-QA-AVAIL-03 The system shall ensure uptime of at least 99.9 %.

GN-QA-MAIN-01 The system should be maintainable with in-house available knowledge.

GN-QA-MAIN-02 The system should utilised software components with long term support policy.

GN-QA-MAIN-03 The system should be maintainable and supported at least for the expected duration of the mission.

GN-QA-MAIN-04 System components should be exchangable.

GN-QA-MAIN-05 System should be modular.
Rationale: Data science technology is not yet established and new systems becoming avialable every day.

A.5.5. Business Rules

GN-BR-DATA-01 All system components shall utilize UTC time for timestamps and syncronise clock with TBD interval.
Rationale: Descrepencies between local client and server time was observed in Delfi-n3Xt datasets.

GN-BR-DATA-02 All system componenets shall utilize Unix format for the timestamps.

A.6. Other Requirements

A.6.1. Documentation Requirements

GN-AUX-DOC-01 The utlized software components should be documented, either by vendor or in-house.

GN-AUX-DOC-02 The system configuration of publically accessible components should be documented and reviewed by external commitee.

A.6.2. Licensing Requirements

GN-AUX-LIC-01 System shall utilise only Open-Standard software components.

GN-AUX-LIC-02 System components shall not limited system upgradability due to licensing agreements.
Rationale: System shall not use components that prevent use of any other systems.

A.6.3. Legal, Copyright, and Other Notices

GN-AUX-LEG-01 System UI shall provide disclaimers, copyright and cookie/tracking statements and require user acceptance.

A.7. Requirements Validation

PASS	Requirement is validated or feasible in given architecture
FAIL	Requirements is not feasible
OOS	Out of scope
-	Unknown or cannot be assessed

Requirement:	Client Leveraged: Conceptual	Client Leveraged: PoC	Streaming Architecture: Conceptual	Lambda Architecture: Conceptual	Proposed Architecture
GN-UI-GEN-01	PASS	OOS	PASS	PASS	OOS
GN-UI-GEN-02	PASS	OOS	PASS	PASS	OOS
GN-UI-GEN-03	PASS	OOS	PASS	PASS	OOS
GN-UI-GEN-04	PASS	OOS	PASS	PASS	OOS
GN-UI-GEN-05	PASS	OOS	PASS	PASS	OOS
GN-UI-GEN-07	PASS	OOS	PASS	PASS	OOS
GN-UI-GEN-08	PASS	OOS	PASS	PASS	OOS
GN-UI-RA-01	PASS	OOS	PASS	PASS	OOS
GN-UI-RA-02	PASS	OOS	PASS	PASS	OOS
GN-UI-RA-03	PASS	OOS	PASS	PASS	OOS
GN-UI-RA-04	PASS	OOS	PASS	PASS	OOS
GN-UI-OPS-01	PASS	OOS	PASS	PASS	OOS
GN-UI-OPS-02	PASS	OOS	PASS	PASS	OOS
GN-UI-OPS-03	PASS	OOS	PASS	PASS	OOS
GN-UI-OPS-04	PASS	OOS	PASS	PASS	OOS
GN-UI-OPS-05	PASS	OOS	PASS	PASS	OOS
GN-UI-OPS-06	PASS	OOS	PASS	PASS	OOS
GN-UI-OPS-07	PASS	OOS	PASS	PASS	OOS
GN-UI-ADMIN-01	PASS	OOS	PASS	PASS	OOS
GN-UI-ADMIN-02	PASS	OOS	PASS	PASS	OOS
GN-NET-01	PASS	PASS	PASS	PASS	PASS
GN-NET-02	PASS	PASS	PASS	PASS	PASS
GN-NET-03	PASS	PASS	PASS	PASS	PASS
GN-NET-04	FAIL	FAIL	PASS	PASS	PASS
GN-ING-01	PASS	PASS	PASS	PASS	OOS
GN-ING-02	PASS	PASS	PASS	PASS	OOS
GN-ING-03	PASS	PASS	PASS	PASS	PASS
GN-ING-04	PASS	PASS	PASS	PASS	PASS
GN-ING-05	PASS	PASS	PASS	PASS	PASS
GN-ING-06	PASS	OOS	PASS	PASS	OOS
GN-ING-07	PASS	PASS	PASS	PASS	PASS
GN-ING-08	PASS	PASS	PASS	PASS	PASS
GN-ING-09	PASS	PASS	PASS	PASS	PASS
GN-ING-10	PASS	OOS	N/A	N/A	N/A
GN-ING-11	N/A	N/A	N/A	N/A	N/A
GN-ING-12	PASS	PASS	PASS	PASS	PASS
GN-ING-13	PASS	PASS	PASS	PASS	PASS
GN-ING-14	PASS	OOS	PASS	PASS	OOS
GN-ING-15	-	-	PASS	PASS	PASS
GN-ING-16	-	-	PASS	PASS	PASS

Requirement:	Client Leveraged: Conceptual	Client Leveraged: PoC	Streaming Architecture: Conceptual	Lambda Architecture: Conceptual	Proposed Architecture
GN-PR-01	PASS	FAIL	FAIL	PASS	PASS
GN-PR-02	N/A	N/A	N/A	N/A	N/A
GN-PR-03	PASS	PASS	PASS	PASS	PASS
GN-PR-04	PASS	PASS	PASS	PASS	PASS
GN-PR-05	PASS	OOS	PASS	PASS	OOS
GN-PR-06	PASS	PASS	PASS	PASS	PASS
GN-PR-07	PASS	PASS	PASS	PASS	PASS
GN-PR-08	PASS	PASS	PASS	PASS	PASS
GN-PR-09	PASS	PASS	PASS	PASS	PASS
GN-PR-10	PASS	PASS	PASS	PASS	PASS
GN-PR-11	N/A	N/A	N/A	N/A	N/A
GN-PR-12	PASS	PASS	PASS	PASS	PASS
GN-PR-13	PASS	PASS	PASS	PASS	PASS
GN-PR-14	PASS	PASS	PASS	PASS	PASS
GN-PR-15	PASS	PASS	PASS	PASS	PASS
GN-PR-16	PASS	PASS	PASS	PASS	PASS
GN-PR-17	PASS	PASS	PASS	PASS	PASS
GN-PR-18	PASS	PASS	PASS	PASS	PASS
GN-PR-19	PASS	PASS	FAIL	PASS	PASS
GN-PR-20	PASS	PASS	PASS	PASS	PASS
GN-PR-21	PASS	PASS	PASS	PASS	PASS
GN-PR-22	PASS	PASS	PASS	PASS	PASS
GN-PR-23	N/A	OOS	PASS	PASS	PASS
GN-PR-24	PASS	PASS	PASS	PASS	PASS
GN-PR-25	PASS	PASS	PASS	PASS	PASS
GN-PR-26	PASS	PASS	FAIL	PASS	PASS
GN-PR-RAW-01	PASS	PASS	PASS	PASS	PASS
GN-PR-RAW-02	PASS	OOS	PASS	PASS	PASS
GN-PR-RAW-03	N/A	OOS	FAIL	PASS	PASS
GN-PR-INT-01	N/A	N/A	PASS	PASS	PASS
GN-PR-STAT-01	PASS	OOS	PASS	PASS	PASS
GN-PR-STAT-02	PASS	OOS	PASS	PASS	OOS
GN-PR-CL-01	PASS	PASS	PASS	PASS	PASS
GN-PR-CL-02	PASS	PASS	PASS	PASS	PASS
GN-PR-CL-03	PASS	PASS	PASS	PASS	PASS

Requirement:	Client Leveraged: Conceptual	Client Leveraged: PoC	Streaming Architecture: Conceptual	Lambda Architecture: Conceptual	Proposed Architecture
GN-DS-01	PASS	PASS	PASS	PASS	PASS
GN-DS-02	PASS	PASS	PASS	PASS	PASS
GN-DS-03	PASS	PASS	FAIL	PASS	PASS
GN-DS-04	PASS	PASS	PASS	PASS	PASS
GN-DS-05	N/A	N/A	PASS	PASS	PASS
GN-DS-06	FAIL	FAIL	PASS	PASS	PASS
GN-DS-07	FAIL	FAIL	PASS	PASS	PASS
GN-DS-08	FAIL	FAIL	PASS	PASS	PASS
GN-DS-09	PASS	PASS	PASS	PASS	PASS
GN-DS-10	PASS	FAIL	PASS	PASS	PASS
GN-DS-11	PASS	PASS	PASS	PASS	PASS
GN-DS-12	PASS	PASS	PASS	PASS	PASS
GN-DS-13	PASS	PASS	PASS	PASS	PASS
GN-DS-14	PASS	PASS	PASS	PASS	PASS
GN-DS-15	PASS	PASS	PASS	PASS	PASS
GN-DS-16	PASS	PASS	PASS	PASS	PASS
GN-DQ-01	PASS	PASS	PASS	PASS	PASS
GN-DQ-02	PASS	PASS	PASS	PASS	OOS
GN-DQ-03	PASS	PASS	PASS	PASS	PASS
GN-DQ-04	PASS	FAIL	PASS	PASS	PASS
GN-DQ-05	PASS	PASS	PASS	PASS	PASS
GN-DD-01	PASS	OOS	PASS	PASS	OOS
GN-DD-02	PASS	OOS	PASS	PASS	OOS
GN-DD-03	FAIL	FAIL	PASS	PASS	OOS
GN-UI-GEN-06	PASS	OOS	PASS	PASS	OOS

Requirement:	Client Leveraged: Conceptual	Client Leveraged: PoC	Streaming Architecture: Conceptual	Lambda Architecture: Conceptual	Proposed Architecture
GN-PERF-01	Unknown	Unknown	PASS	PASS	OOS
GN-PERF-02	PASS	PASS	PASS	PASS	PASS
GN-PERF-03	Unknown	Unknown	PASS	PASS	PASS
GN-PERF-04	PASS	FAIL	PASS	PASS	OOS
GN-PERF-05	N/A	N/A	N/A	N/A	N/A
GN-PERF-06	PASS	FAIL	PASS	PASS	PASS
GN-PERF-07	FAIL	FAIL	PASS/FAIL	PASS	PASS
GN-SAFE-01	PASS	PASS	PASS	PASS	PASS
GN-SAFE-02	PASS	PASS	PASS	PASS	PASS
GN-SAFE-03	PASS	PASS	PASS	PASS	PASS
GN-QA-SEC-01	PASS	FAIL	PASS	PASS	PASS
GN-QA-SEC-02	PASS	OOS	PASS	PASS	PASS
GN-QA-SEC-03	N/A	OOS	PASS	PASS	OOS
GN-QA-SEC-04	PASS	FAIL	PASS	PASS	OOS
GN-QA-SEC-05	PASS	FAIL	PASS	PASS	OOS
GN-QA-SEC-06	PASS	PASS	PASS	PASS	PASS
GN-QA-SEC-07	PASS	FAIL	PASS	PASS	OOS
GN-QA-SEC-08	PASS	FAIL/OOS	PASS	PASS	OOS
GN-QA-SEC-09	PASS	FAIL	PASS	PASS	PASS
GN-QA-REL-01	-	-	-	-	-
GN-QA-REL-02	-	-	-	-	PASS
GN-QA-REL-03	-	-	-	PASS	PASS
GN-QA-REL-04	-	-	FAIL	PASS	PASS
GN-QA-AVAIL-01	PASS	PASS	PASS	PASS	OOS
GN-QA-AVAIL-02	PASS	PASS	PASS	PASS	PASS
GN-QA-AVAIL-03	PASS	Unknown	PASS	PASS	OOS
GN-QA-MAIN-01	PASS	FAIL	PASS	PASS	PASS
GN-QA-MAIN-02	PASS	PASS	PASS	PASS	PASS
GN-QA-MAIN-03	PASS	Unkown	PASS	PASS	PASS
GN-QA-MAIN-04	PASS	FAIL	PASS/FAIL	PASS	PASS
GN-QA-MAIN-05	PASS	FAIL	PASS	PASS	PASS
GN-BR-DATA-01	PASS	OOS	PASS	PASS	PASS
GN-BR-DATA-02	PASS	PASS	PASS	PASS	PASS
GN-AUX-DOC-01	PASS	FAIL	PASS	PASS	PASS
GN-AUX-DOC-02	PASS	OOS	PASS	PASS	OOS
GN-AUX-LIC-01	PASS	PASS	PASS	PASS	PASS
GN-AUX-LIC-02	PASS	FAIL	PASS	PASS	PASS
GN-AUX-LEG-01	PASS	OOS	PASS	PASS	OOS

B

Spark Streaming Experiment Addendum

Table B.1: Point description for figures B.1 and B.2

Point	Description
1	Data added to Kafka topic
2-10	Run C
11	Data added to Kafka topic
12-19	Run D

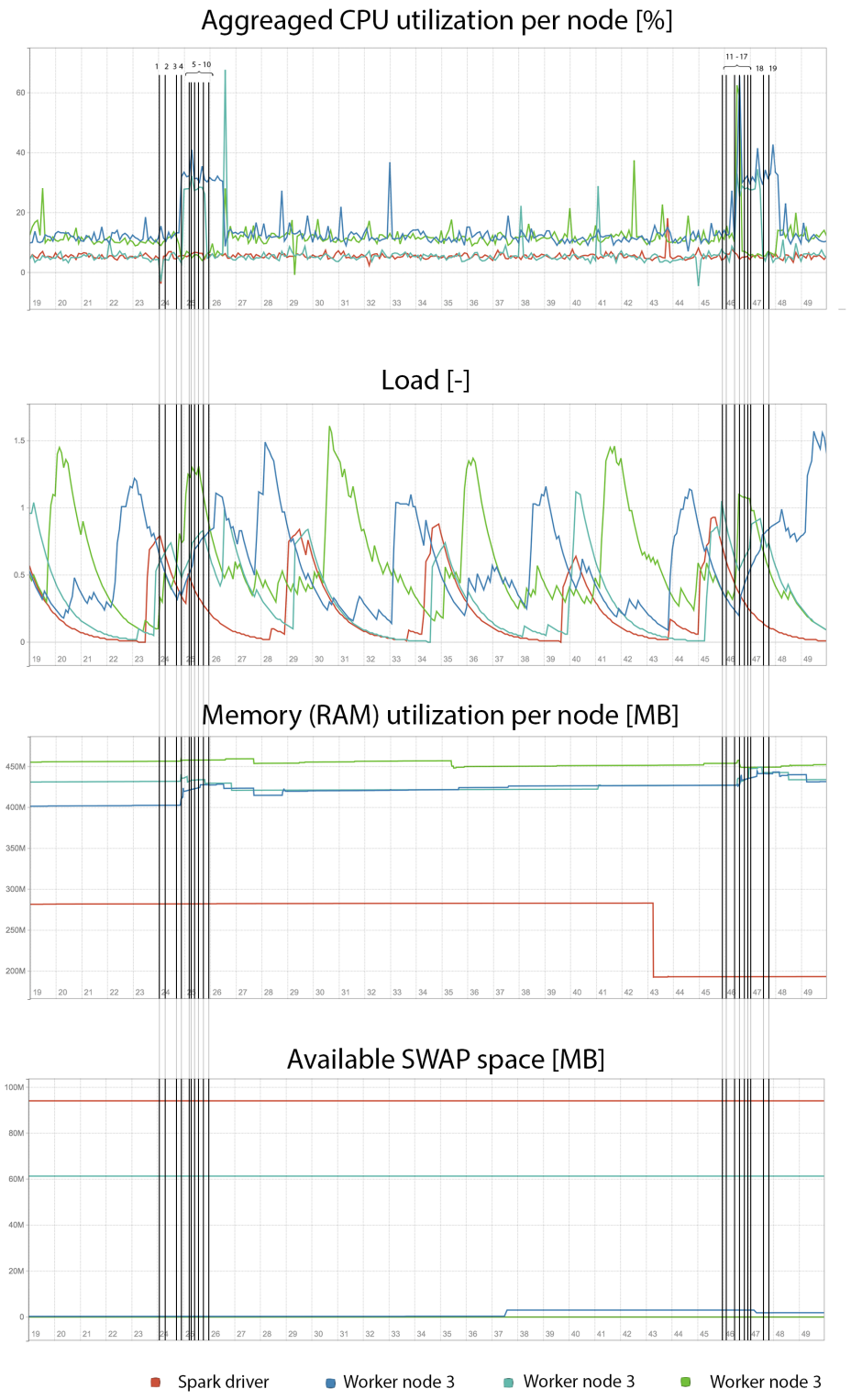

Figure B.1: Spark Streaming experiment C and D

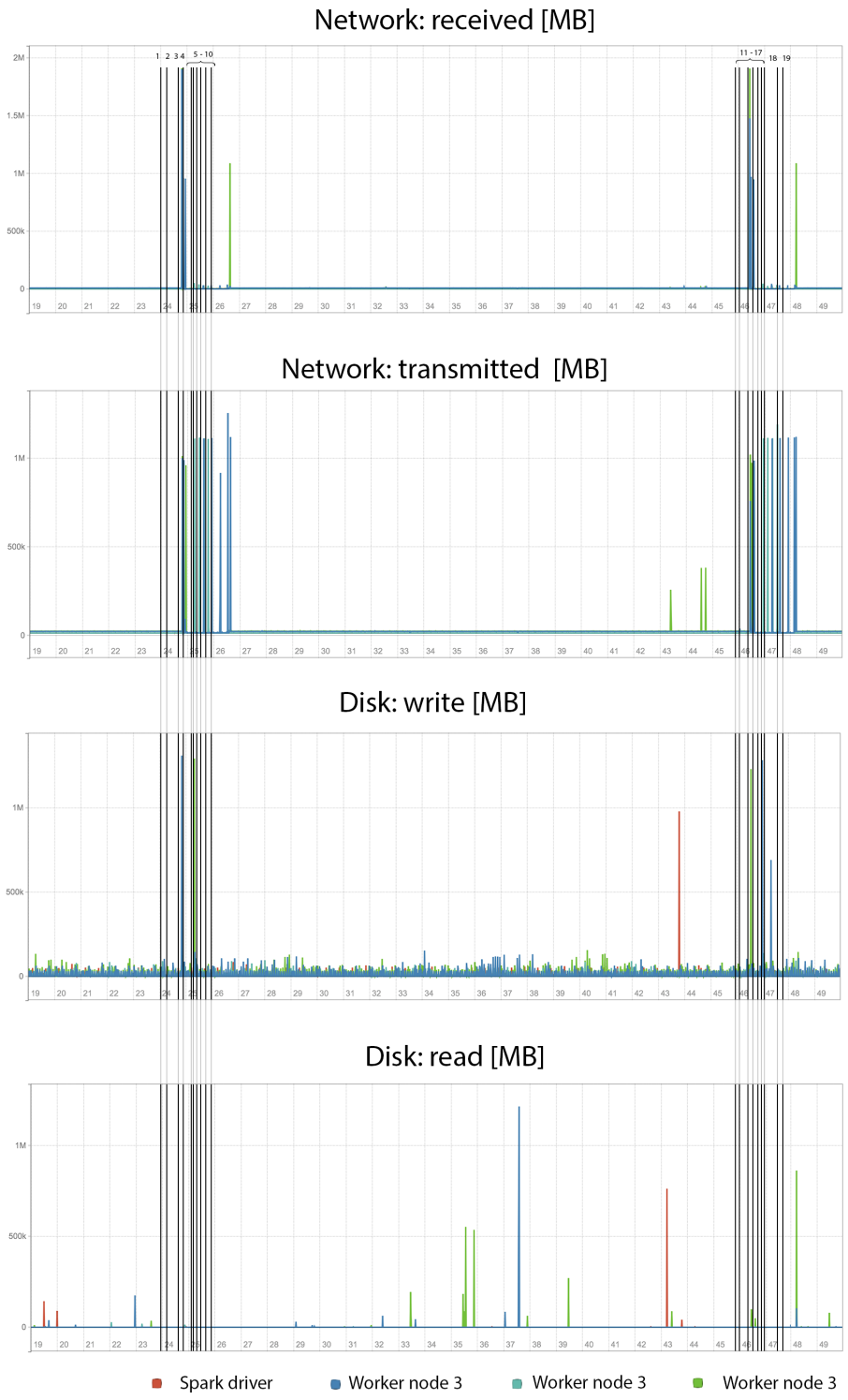

Figure B.2: Spark Streaming experiment C and D [cont.]

Table B.2: Stream Processing speed on RPI cluster: gray

Run C			
Point	Eq. Latency[s]	Frames in Kibana	Processing speed [f/s]
2	10	9	0.9
2	11	17	8.0
2	12	18	1.0
3	34	143	5.7
4	46	264	10.1
5	55	379	12.8
6	57	502	61.5
7	65	624	15.3
8	74	744	13.3
9	95	865	5.8
10	110	1000	9.0

Run D			
Point	Eq. Latency [s]	Frames in Kibana	Processing speed [f/s]
12	8	9	1.1
13	31	136	5.5
14	42	256	10.9
15	52	371	11.5
15	53	493	122.0
16	64	624	11.9
17	70	744	20.0
18	89	865	6.4
19	105	1000	8.4

C

Literature Study

Delft University of Technology

Selection of telemetry server software for the DelFFi mission

Anatoly Ilin, 1538144

September 18, 2014

Supervisor: Jasper Bouwmeester

Contents

List of abbreviations

AHP	Analytic Hierarchy Process
API	Application Programming Interface
AS	Authentication Server
CLI	Common Language Infrastructure
COSMIAC	Configurable Space Microsystems Innovations & Applications Center
COTS	Commercial Off-The-Shelf
CRC	Cyclic Redundancy Check
DCGS	Radio Amateurs and Delfi Central Ground Station
DEM	Data Event Messages
DUDe	Delft Universal Data extractor
DX	Frame
EEM	Error Event Messages
EGSE	Electrical Ground-Support Equipment
EPFL	Swiss Space Center
FGN	Federated Ground Station Network
GENSO	Global Educational Network for Satellite Operations
GMS	Ground Station Management Service
GROWS	Ground Station Remote Operation Web Service
GS	Ground Station
GSML	Ground Station Markup Language
GSS	Ground Station Server
GUI	Graphical User Interface
MCC	Mission Control Client
MCS	Mission Control System
MSGN	Mercury Ground Station Network
P2P	Peer-to-Peer
PTO	Packet Type Objects
RA	Radio Amateur
RAW	unprocessed data
S/C	Spacecraft
SCOE	Special Checkout Equipment
SQL	Structured Query Language
SSH	Secure Shell
SSID	Secondary Station Identifier
TBD	To Be Defined
TM/TC	Telemetry and Telecommand hardware interface
UNISEC	University Space Engineering Consortium
UTC	Coordinated Universal Time
VKI	The von Karman Institute
WOD	Whole Orbit Data
YODA	Your Own Data Analysis

Introduction

DelFFi, the successor of the Delfi-n3Xt cubesat mission, is a twin satellite mission that makes part of the 50 cubesat network QB50. The payload on board of both satellites, Delta and Phi, will perform in-situ measurements of the lower thermosphere, the least explored layer of the atmosphere. In addition to the QB50 mission objective, DelFFi will demonstrate the autonomous formation flying of both satellites using various guidance, navigation and control technologies [26].

During the Delfi-n3Xt mission, the data sent from the satellite, the telemetry, is collected by a number of radio-amateurs and is delivered to the university through the DUDe client software to the telemetry server. Whereupon the telemetry server processes and distributes data to the stakeholders. In contrast to Delfi-n3Xt , DelFFi mission opposes requirements, not only on the data storage and processing but as well on the functionality. So it is required to deliver the whole orbit data (WOD) to the QB50 committee within 24 hours [QB50-SYS-1.7.9]. In order to accommodate these new requirements, this document will answer the question whether any updates should be performed on architecture of Delfi-n3Xt server implementation to make it compatible for the DelFFi - mission or whether a complete new or off-the-shelf software should be used.

From analysis it has been determined that in-house telemetry server software is more desirable, when compared to considered software products. It adds to certain extend risk to the system, but provides unmatched design flexibility and due to presence of the source code, contributes to the refinement of the product on the long run. By doing so it increases the overall mission capability and arguably the mission outcome as well.

In order to fully assess the need for telemetry adjustment the set of requirements for both missions will be defined in Chapter 1. Once the requirements are analyzed, Chapter 2 will present the high level functionality of the Delfi-n3Xt telemetry server. This will be used as an introduction into the telemetry server functionality, users and data flows. Chapter 3 will present the telemetry systems developed by 3rd party organizations. Chapter 4 will cover a more in-depth design of the Delfi-n3Xt server, which will be followed by the trade-off in chapter 5.

1 Requirements

In order to justify the selection of the available software, a set of trade-off criteria is required. Those criteria define how well software is suited to the mission needs. The software selection is a well studied approach and a number of critical selection criteria are well defined in the literature [20]. Therefore the challenge lies not in the selection of the criteria, but rather in understanding and specifying how well given software solution comply with them. The first step in this process is the definition of the 'need' and the actual problem definition. In order to answer this question the Systems Engineering Approach is followed. By analyzing requirements, the problem domain and the actual software capability should be able to be assessed easily.

Sections below present the requirements used for design and implementation of the Delfi-n3Xt telemetry server and requirements set for the DelFFi mission with relevance to the telemetry server.

1.1 Delfi-n3Xt Requirements

Delfi-n3Xt is a single cubesat mission designed and operated by the Delft University of Technology since XXXX. From the technical documentation such as design documentation a number of requirements has been derived:

Delfi-n3Xt Mission Requirements

- FUNCTIONAL MIS-F.01 The mission shall facilitate payloads from external partners
- FUNCTIONAL MIS-F.02 The mission shall facilitate bus advancements w.r.t. Delfi-C3
- FUNCTIONAL MIS-F.03 The mission shall facilitate additional experiments under strict conditions, as specified in [SLR 0263]
- GENERAL MIS-G.02 The mission shall facilitate educational objectives

Telemetry Server Requirements

- FUNCTIONAL GS.2.3-F.01 The telemetry server shall provide an end-to-end data flow; receive data from the DUDe telemetry client, process this data, and distribute this data to the users.
- FUNCTIONAL GS.2.3-F.02 The telemetry server shall have redundant data storage at different location.
- PERFORMANCE GS.2.3-P.01 The telemetry server shall have data storage of at least 1150 Gbytes.
- PERFORMANCE GS.2.3-P.02 The telemetry server shall support at least 50 simultaneous connections from telemetry software clients.
- PERFORMANCE GS.2.3-P.03 The telemetry server shall be connected to the internet with a bandwidth of at least 4 Mbits/s.

1.2 (Preliminary) DelFFi Requirements

At the moment of writing of this document, the DelFFi, successor of the Delfi-n3Xt, is in the design phase. Therefore, the set of the requirements as mentioned in this document may deviate from final version.

DelFFi Mission Requirements

- MIS-F-03 DelFFi: The mission shall facilitate bus advancements w.r.t. Delfi-n3xt.
- MIS-F-04 DelFFi: The mission shall facilitate two identical satellites.
- MIS-C-DelFFi: A lifetime of 0.5 year of the satellites shall be taken into account in engineering decisions (such as system dimensioning, solar panel, etc.).
- MIS-C-DelFFi: For project scheduling, the delivery of the satellites shall be assumed to be no later than Feb 2015
- MIS-DelFFi: The COMMS of both satellites shall be capable of data transmission between the two spacecraft via two different routes: direct ISL and ground-in-the-loop

QB50 requirements:

- QB50-SYS-1.7.9: The CubeSat provider shall transfer the whole orbit data and science data to the QB50 storage server within 24 hours following reception on the ground.
- QB50-SYS-1.5.13: The CubeSat shall use the AX.25 Protocol (UI Frames). The data type during downlink shall be specified in the Secondary Station Identifier (SSID) in the destination address field of the AX.25 frame. Science data shall be indicated using 0b1111 and Whole Orbit Data with 0b1110.
- QB50-SYS-1.5.14: User-friendly and documented software consisting of a) CubeSat data Frames Decoder b) CubeSat data Packet Decoder and c) Cube- Sat data Viewer that complies with radio amateur regulations shall be made available to VKI 6 months before the nominal launch date.
- QB50-SYS-1.7.9: The WOD shall be send by the teams to the QB50 server using provided format
- QB50-SYS-1.5.13: The CubeSat shall use the AX.25 Protocol (UI Frames). The data type during downlink shall be specified in the Secondary Station Identifier (SSID) in the destination address field of the AX.25 frame. Science data shall be indicated using 0b1111 and Whole Orbit Data with 0b1110.

Telemetry Server Requirements:

- OPERATIONS: Store the commands sent to the satellite
- TIMEKEEPING: The ground segment should be able to determine the reception time in UTC with 0.2s accuracy
- TIMEKEEPING: The telemetry processing software should be able to determine the Epoch and drift of the onboard elapsed time counter
- ORBIT DETERMINATION: The telemetry processing software should be able to retrieve two-line-elements automatically when there is an update and append each frame with the relevant set.
- ORBIT DETERMINATION: The telemetry processing software should be able to calculate the orbital position in a TBD format from the most accurate time stamp and the two-line-elements.
- WEBSITE: Always display the latest frame, if possible by pushing the data from the server
- WEBSITE: Display live data, optionally tables/figures/graphs
- WEBSITE: Website authorization. If possible by using existing user-pool, without asking users to create a new account.
- OPERATIONS: For real time interface, indicate when a value exceeds the given threshold

1.3 Mission specific deviation between DelFFI and Delfi-n3Xt missions

Based on the requirements for both missions it can be concluded that the mismatch in requirements between both missions occur in the following topics:

- Number of the spacecraft, which results in number of discrepancies such as user interface, storage etc
- Telemetry frames format (flexible frame architecture), which require different data decoding and storage techniques
- Data transfer from Del to Phi satellite through the ground station network, which may oppose requirements on the telemetry server
- Whole Orbit Data (WOD) requirement, which require an API for the QB50 server

Because the set of requirements for the DelFFi mission is incomplete this approach cannot provide sufficient insight on the system. A deeper analysis is required to be able to perform a meaningful software selection.

2 High level functional analysis of the Delfi-n3Xt telemetry server

In the chapter 1, it has been determined that the telemetry software selection can not be solely based on the documented requirements. In order to overcome this issue, the Delfi-n3Xt telemetry server will be analysed from functional point revealing the mission critical components and overall functionality. The main domain of the interest is the data flow from the satellite to the end user (operations crew and QB50) as the data processing techniques. Analysis is done on the basis of available papers, technical notes and the source code of the Delfi-n3Xt and Delfi-C3 missions.

The functional breakdown of Delfi-n3Xt mission with the relevance to the telemetry server can be found in figure 2.1 [8]. It can be seen that the data operations relay on three tiers: Data Collection, Data Handling and Data Distribution. Data Collection is build up from two elements: Data Acquisition and Data Storage. Data Handling function is performed by 3 sub-functions: Frame Filter(ing), Frame Merger(Merging) and Frame Processor(Processing). Data Distribution function is performed by providing a web interface for different stakeholders.

The Data Acquisition is performed by the radio amateurs using 3rd party soft- and hardware, who transfer the Data Frames to the server by means of the DUDe-client. Data Storage is a separate subsystem of the server that is managed by MySQL database system [25]. The data transfered by the DUDe client is injected directly into the database.

The incoming frames, inserted into the database by DUDe, are checked by the Frame Filter, to ensure that frames indeed originate from Delfi-n3Xt cubesat and no transmission errors has occurred (e.g. bitflips or effect of the noise on the signal). It should be noted that the solely purpose of the Frame Filter is to discard faulty data and no error corrections are performed. Data sent from Delfi-n3Xt satellite is encoded into a AX.25 frame format. As the amount of data exceeds single AX.25 frame specifications [4], telemetry data had to be split and transfered into two separate frames. It is therefore required to merge data of the both frames on the server side of the mission. In the next step, Frame Merger, the payload frame is extracted from the data frames by the Frame Merger and combined with the frame payload from the other matching frame. The reason behind this design approach, is the fact that the telemetry server software was finalized before the decision of the frame splitting was made. By encoding frames differently and by adding the Frame Merger it was no longer required to rewrite the existing telemetry software. Frame Merger is followed by the Frame Processor, that decodes the combined frame-payload to the human readable format and injects it into the server database.

The Data Distribution part assures the data flow from the server to the end user. This function is closely tied with Server Administration, which handles the authorization and contains user information. In order to provide access to the remote users, data is distributed using a web interface. System identifies different users types and provide access rights accordingly [8], this is summarized in the table 2.1. Administrator was not considered a user type by the developer and is been added to the table below for the sake of completeness.

The physical Ground Segment architecture of the Delfi can be found in the figure 2.2, it differs from functional overview as it only presents the physical connections between different subsegments. The system architecture is based on a central processing unit, the telemetry server, that relies on the clients (Radio Amateurs and Delfi Central Ground Station (DCGS)) for the data acquisition and a web-interface for the data distribution.

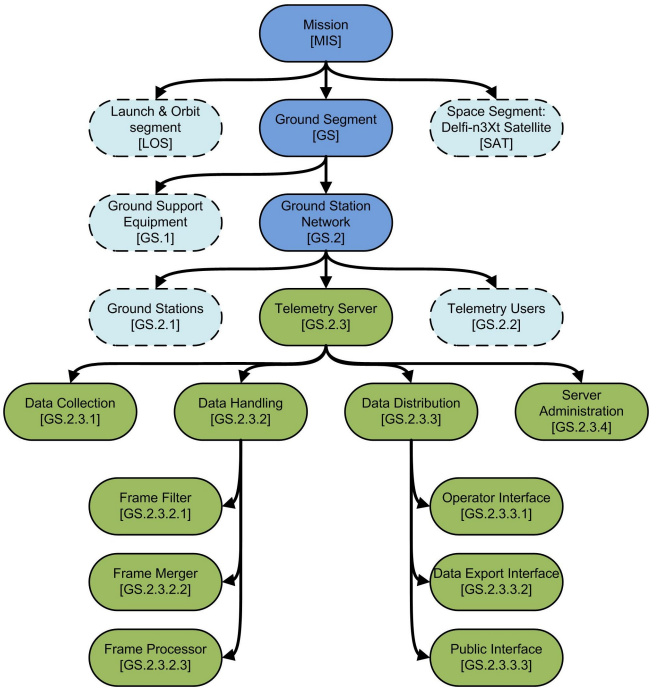

Figure 2.1: Functional Mission Breakdown with relevance to the telemetry server

User type	User description	User interface
Operator	Users directly involved with spacecraft operations	Real time overview of the incoming data (*Operator Interface*)
Telemetry analyst	Users responsible for historical data interpretation	Data archive and data mining/ exploration (*Data Export Interface*)
Radio Amateur	Stakeholders that receive and transmit satellite data to the server	Overview of the data frames they sent to the server personally (*Not specified by the documentation*)
General Public	Stakeholders not involved in the project	General overview of the satellite status (*Public Interface*)
Administrator	User responsible for server operations and maintenance	Should have access to all user interfaces (*All Interfaces*)

Table 2.1: Users types and access privileges [8]

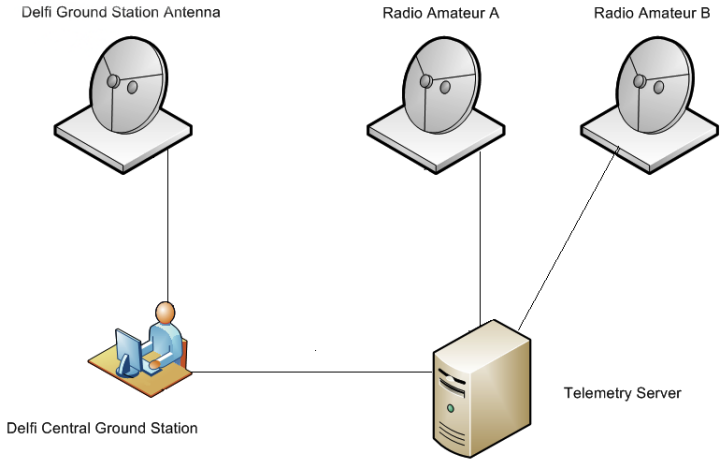

Figure 2.2: Physical overview of the Delfi Ground Segment

3 Third party applications

The in-house software development is a costly affair in terms of man-hours that introduces an extra risk factor to the mission. The usage of a validated third-party software may therefore be more beneficial for the overall outcome of the mission. In the subsections below a number of present-day and historical Ground Segments will be presented.

3.1 Swiss Space Center (EPFL)

The EPFL ground segment is used for SwissCube missions for 4 years. During this period it provided the software interface for the two-way communication with the EPFL satellites. Swiss Space Center (EPFL) agreed on providing a software for the ground station network to all interested QB50 participants. In this section the functional (system logic) analysis will be performed along with high level system architecture overview.

The logical overview of the Swiss Space Center can be found in the figure3.1.

The system is based on three elements:

- Ground Station (incl. Antenna's, Complementary Hardware and Control Software)
- TM/TC Front End
- The Mission Control System (MCS)

Each Ground Station that participates in the EPFL network has a standardized interface on the software side, which allows an easy expansion of the ground segment and assures compatibility within the EPFL software. The ground station interface software is part of the software package provided by EPFL. The second element is the TM/TC frond end, TM/TC stands for TeleMetry /TeleCommand and forms an interface between incoming data form the GS, which is frame based to packet-based data required by the Mission Control System[18]. Within this configuration, "TM hardware performs all baseband functions from interfacing to frame synchronization, de-scrambling, Convolutional and/or Reed-Solomon decoding / correction, time/quality tagging and frame level transfer via the LAN interface" [33]. In other words, TM/TC is responsible for AX.25 frame handling (storage of the raw telemetry frames and decoding of the telecommands to AX.25 format) and repacking it to the packet structure accepted by the Mission Control System. The last element is the Mission Control System (MCS). It forms the central part of the EPFL ground segment architecture and is responsible for the monitoring of the space and the ground segment for the mission, along with the (post-)processing and the storage of the spacecraft data. Additionally, MCS provides a telecommand interface, for the telecommand encoding, sequencing and sending to the S/C (through TM/TC for encoding and GS for actual broadcast).

From the functional perspective, Ground Segment architecture of EPFL is in many ways comparable to that of Delfi-n3Xt mission:

1. Defragmented Ground Stations with a standardized software interface
2. Separation of the RAW telemetry database from the Processed data database
3. Single Server for the (post-)Processing and Data Distribution

The main difference between EPFL and Delfi implementation is the presence of the TM/TC Front End device for Data Processing. Instead of frame decoding and processing on the server side, the tasks are outsourced to a dedicated hardware. The physical system architecture is however very different, note that "star"-type configuration around EGSE Router in the figure 3.2.

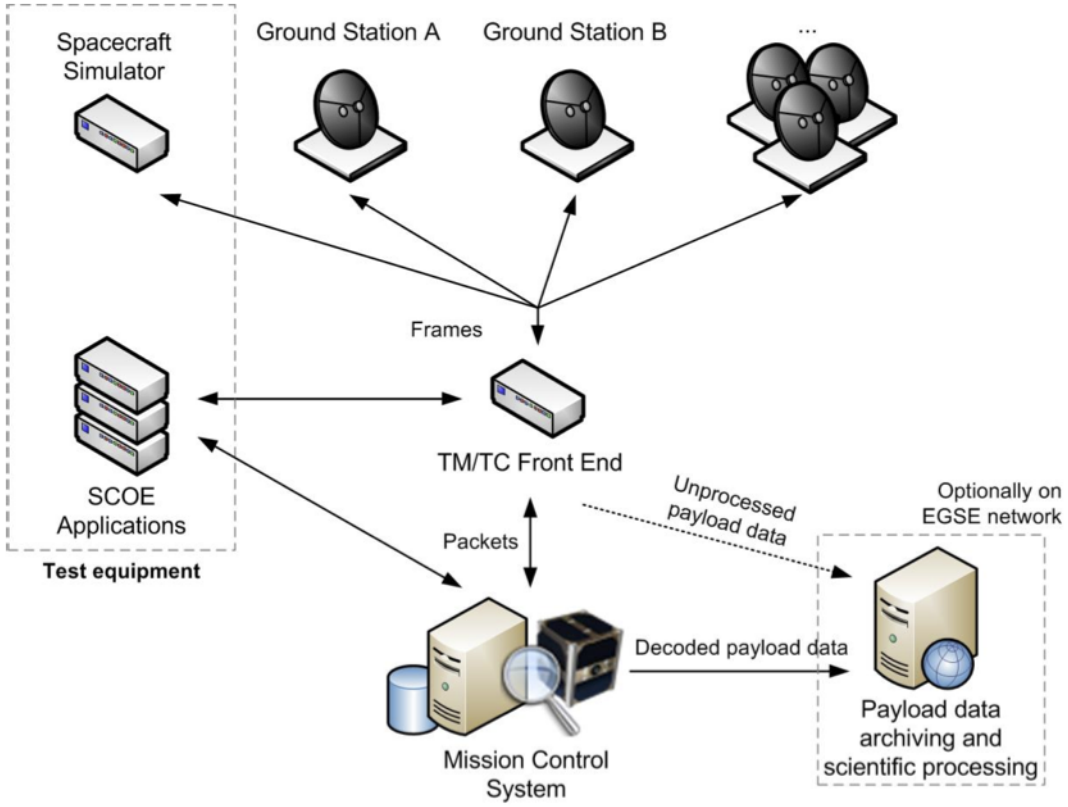

Figure 3.1: Logical view of EPFL ground segment [18]

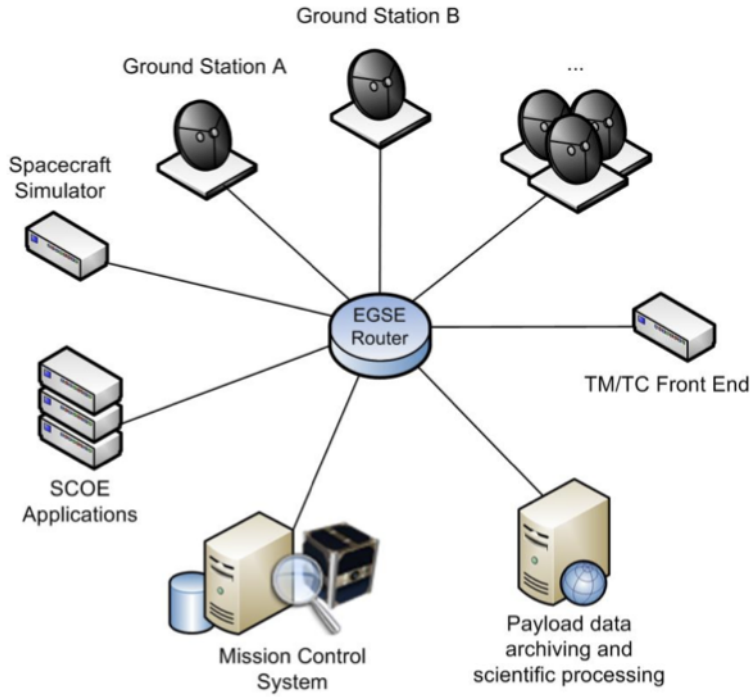

Figure 3.2: Physical view of EPFL ground segment [18]

EGSE stands for Electrical Ground-Support Equipment and is essentially a COM-based protocol (routing) developed in late '90 by ESA/ESTEC to define interfaces within an open communications architecture. In addition, "the use of web based standards (later upgraded to TCP/IP [18]) and technologies such as XML, SOAP and WSDL provides a significant increase in interoperability of the developed EGSE modules" [15]. In its core EGSE is a "central checkout system" , software system designed to provide an interface to different instruments during pre-launch testing of a spacecraft. Essentially, EGSE router acts as a broker and the whole communication system is a derivative of a messaging protocol. In EPFL implementation, the data sent across the network is ether data-frames from GS or EGSE-data packages from different network devices (e.g. MCS to TM/TC front end). This set-up has an advantage compared to Delfi-n3Xt since the devices (extra servers, etc) can be added to the network without any need for adjustment to the system software (that handles routing). For example, an extra payload processing unit (as seen in the figure 3.2).

Architecture Analysis: EGSE Router

Since the emergence of the "cloud" architectures, middelware messaging is no longer desirable and the star-architecture is no longer a state-of-the-art setup. Instead of the complex implicit routing roles, many of the modern messaging protocols rely on the two-way publish-subscribe techniques and flexible ("direct") routing. The main advantage of these new systems is reliability, as no single point of failure is present.

It does not mean that the software is obsolete, EGSE server is widely used in Space industry for a number of tasks. It is even considered to be the industry standard within ESA and many of the new missions that require remote access or data/telemetry storage are based on the EGSE Router [6]. Even the older systems are been migrated to EGSE, successfully but not as smoothly as intended [19].

The presentation of the European Space Agency's Herschel Space Observatory [19] describes the transition and experiences with EGSE System. Because the EGSE server is the single most critical element of the EPFL software, it is important to investigate the encountered issues. Most problems are related to the migration issues such insufficient documentation of the software used by Herschel, what resulted in the EGSE implementation team not be aware of required software-pipelines. Secondly, the EGSE Components are based on the Microsoft .NET Framework [17] which makes components not directly compatible with any other software and programming languages. It is clear that Herschel solved some of this issues by implementing wrappers, programs that execute a different program (or routine) when executed. Essentially, a .NET compatible software executes a non-compatible software and acts as an interface between both. It is considered to be a good practice to create a standardized system and utilize a single programming language, but it may create a number of boundaries for the future development. This is especially important in case of EPFL software (read further), as the only way to increase system capability is to create custom plugins. The main EGSE routing system issues, relevant to DelFFi mission, identified by Herschel is the limited documentation, very limited message logging and lack of out-of-the-box configuration. It is unclear whether EPFL pre-configure the software for the QB50 participants, therefore limited documentation and hard to probe network (limited messaging logging) may become a challenge to keep system stable.

EGSE Routing Because the EPFL software is available to all QB50 participants, it is important to investigate whether it can be used as Delfi alternative. Due to the high system complexity, the routing of the messages will be investigated first.

Figure 3.3 shows the connection life-cycle of EGSE Router and the users. Before a client can transmit messages, he is required to be connected to the server. First, client creates a connection with the server and requests a register command. The request-message sent to the server (see 3.1), contains a client ID and Name. Both of them must be unique. This means that system requires a client configuration (at least on name and ID- level), before the client can create a connection. Furthermore, router cannot assign a temporary ID/Name to let client discover the available ID/Names and adjust temporary to new fixed once. The network is comparable to Static IP and MAC address: every client requires an unique IP and MAC address and can only

connect to the router if both prerequisites are met. This is a clear disadvantage of this system. For a Delfi-example, a new Radio Amateur cannot transmit data after downloading and installing the software, it requires an extra layer for ID creation. Clearly this makes configuration more complex. Documentation indicates a possible existence of Radio Amateur interface for QB50 network. This may resolve this issue.

Every client on the network is required to register before messages can be sent. Upon the register request, router reacts to the register command with an event, that confirms the registration. Two types of events are supported: Data Event Messages (DEM) and Error Event Messages (EEM). A DEM passes the result of a successful command execution or the actual data when it is sent from another Client. EEM are sent when a Command fails execution (see 3.1). For Sending the data, client transport the data to the Router who will transfer the data to appropriate user. The destination of the message is determined by ID of the client which is embed in the Destination field. Again, it should be noted that ID should be known before the message is sent. Additionally, system does not (yet) support Name lookup. Therefore it is not possible to determine ID from the Client Name. It is however possible to determine the Client Name from clients ID. This pointless, as messages require ID's and not Names. Another interesting detail missing in the documentation is the source field. Because message passes the Router, in is unknown whether the source of the message at the end of the line is the router or the actual sender.

The Receiving of the data follows same guidelines. In order to disconnect from the network, client send a disconnect request to the Router and receives a confirmation request. It can be seen that by default, clients are not informed about a new clients connecting or leaving the network. For example, if a ground station disconnects the network, MCS is not informed unless it is implicitly programmed to do so. For example, by having "keep alive" connection with all GS or sending regular "ping"-like messages.

The Routing of EGSE network is performed on the basis of Client ID. Specific Client ID ranges has been predefined. For example 0xF000 is hardcoded ID of the router and range 0xF001- 0xFFFE is pre-allocated and is not usable for the clients.Furthermore, it is possible to broadcast a message to whole network by sending the data to ID 0xFFFF. This way it is possible to inform connected clients about a new client connecting to the network, but EGSE Router is not programmed to do so. Theoretically, it is possible to inform clients about a disconnecting of a client as well, but this is a suboptimal solution as a client can be timed-out or can lose connection due to network issues.

EGSE Message As stated earlier in the document, EGSE is a messaging protocol and data sent across the network is contained within a message. Two types of messages can be identified: a Command Message and Event Message. A Command Message is a message that has Router as recipient, as name suggests, it contains an instruction for Router to executer (e.g. disconnect). Event Messages are sent as a result of certain event, such as a message from a different client of upon command execution by the Router.

The message layout can be seen in the figure 3.4. On the first sight the protocol does not impose any constrains on the size of the message body (the actual sent Data), but upon a closer look, the size of the message length is limited to 4 octets (4 x 8 bit). Message Length is expressed in octets, so maximum length of the message data is constrained to 65542 octets or 64 kB. For example the maximum allowable length of a full AX.25 frame is 256 octets. It can be seen that 256 AX.25 frames can be fit in a single EGSE Message body.

The Message Type is required to pass a command to the Router. For example, a Type 1 message from client to the server will Unregister client from the network. The Result Code provide information on the success of the command sent to the Router. The default value, 0, is defined as a successful command execution. In case of a failure, client can receive one of 15 possible error codes. The Destination and Source ID identify the receiver and the sender of the message. The Token is an identifier of the sent command, the command event will contain identical token for the traceability purpose. It should be noted that it is the responsibility of the client to generate unique tokens. It is not a system requirement, but if the last is neglected the traceability will be lost. It is furthermore unclear how a new-coming client can determine the next available token. The next field contain UNIX time stamp when the message has been sent.

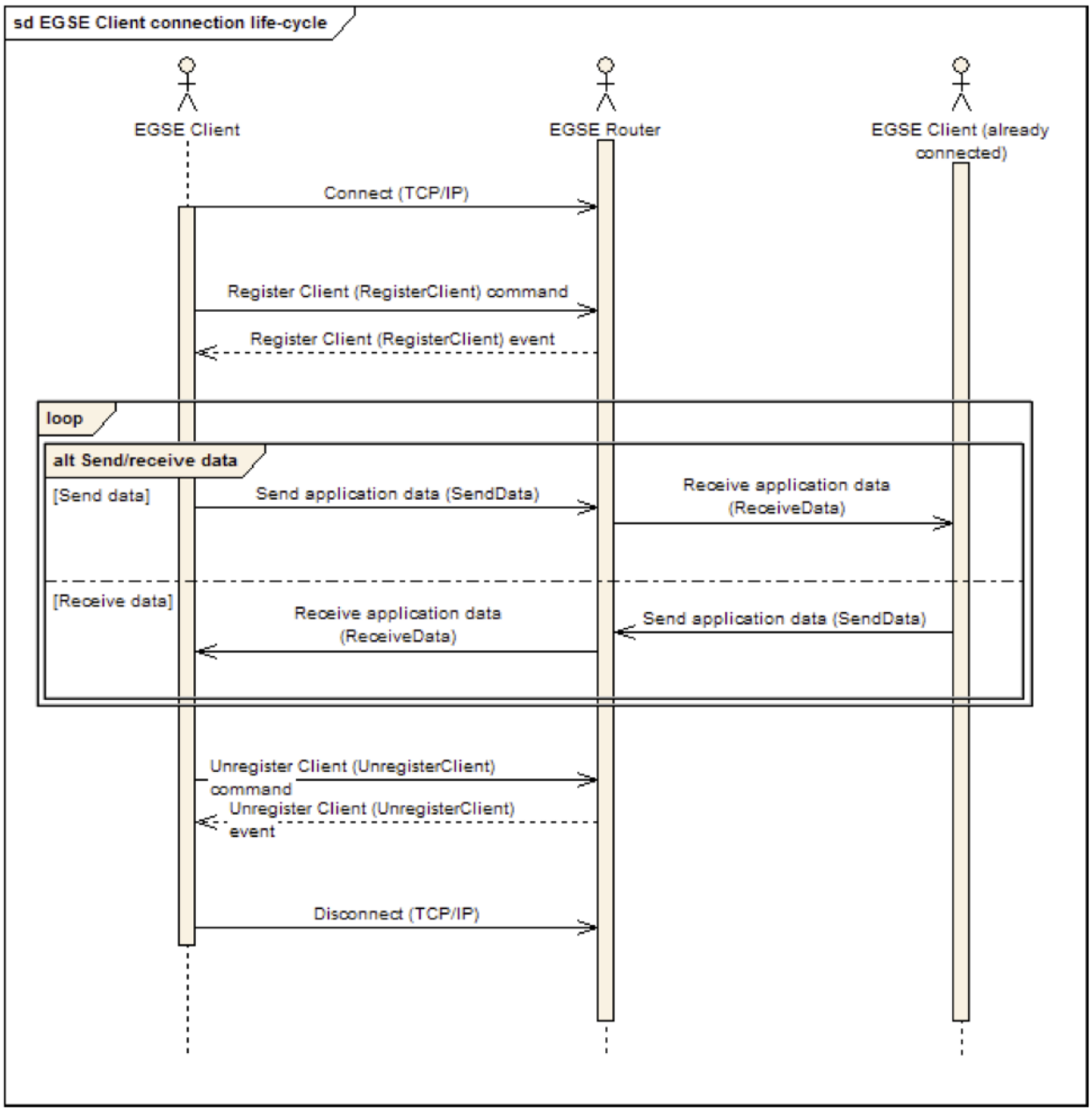

Figure 3.3: EGSE Routing [18]

Part		Field	Size
Header		Message Length	4 octets
		Message Type	1 octets
		Result Code	4 octets
		Destination ID	2 octets
		Source ID	2 octets
		Token	4 octets
		Time	8 octets
	Data Header	Data Type	1 octet
		Spare	1 octet
		Spacecraft ID	2 octets
Data		Message Type specific data	any

Figure 3.4: EGSE Message [18]

Router does not modify this field when transferring message to the next client. Theoretically, by combining Time and Token field, the unique token issue can be solved. Spare is a 1 octet long field unused, but allocated for future application. Spacecraft ID contains the unique identifier of the spacecraft. Data type defines the type of data embedded in the message. EGSE Server of EPFL allow 8 pre-configured data types (see 3.1), essentially different types of reports and requests.

Data types In order to identify the data sent across the network, data types has to be defined. As the name suggest, data type identifies type of data contained in the message. But it goes further than that. In order to explain this, analogy of an email will be used. Consider a client X who would like to know the shoe size of Client Y by sending Y an email. In order to make email clearer he includes a title: "Shoe size". The Client Y replies to his mail and automatically adds "RE" to the title ("RE: Shoe size"). EGSE Router uses same technique, but instead of a refined title, it uses 1 byte HEX. The question to different client has a Request type, answer a Report type. EGSE Router as designed by ESA/ESTEC is preconfigured with 4 different types of Request and hence 4 types of Reports. The data type with value below 63 are predefined by ESA and cannot be modified, types 64 to 127 are considered to be mission specific and are freely available.

Table 3.5 summarizes the pre-configured data types. It can be seen that first three types are implemented to perform Special Checkout Equipment (SCOE) actions. Any network component that can perform and execute a command is considered a SCOE: for example a ground station, TM/TC equipment etc. Type 1 is used to send a command, which is verified upon execution with type 2 report. Type 3, an observation report, is sent without a command request. Type 4, the Send Telecommand Packet Request, is a request containing telecommand for the satellite sent by Mission Control System to the TM/TC front end. This type of Packet Request has a number of settings, such as usage of Virtual Channels etc. The Verification Report, type 5, contains the telecommand is been transmitted by the GS. It contains a traceability error-coding, so it is possible to determine where the telecommand was rejected in case of an error.

Type 6 and Type 7, is Telemetry Packet and Frame Reports are used to transfer telemetry packets and frames. Packets are defined in the EPFL software as reconstructed frames, essentially it is data decoded from the frame. Furthermore it is unclear whether it is user defined. The

#	Name
1	SCOE Command Request
2	SCOE Command Verification Report
3	SCOE Observation Report
4	Send Telecommand Packet Request
5	Send Telecommand Packet Verification Report
6	Telemetry Packet Report
7	Telemetry Frame Report
8	Time Correlation Report

Figure 3.5: EPFL EGSE available datatypes

message-body does not contain any extra overhead like other data types and only the actual telemetry frame is transmitted. Type 6 messages are mainly used for the packets transfer from TM/TC to MCS, meaning that at MCS does not directly receive any information on GS the frame originated from. In order to solve this issue type 8 has been introduced, it contains a Time Correlation Report, essentially a message that correlates a frame to the received GS by satellite timestamp embed in the frame. Type 8 report is generated by GS to MCS without Router modifying the message. This way MCS can determine the GS a given frame originated from. The data field of type 8 message contains Onboard Time Packet, satellite timestamp of data transmission, UTC Time of data acquisition by the GS and an extra field containing internal GS delays.

Mission Control System

EPFL Mission Control System (MCS) has three functions: processing, storage and distribution of the data. The processing and distribution parts can be extended using custom modules. By doing so an additional database will be created to contain data generated by the custom modules. The MCS is build using .NET Framework, hence all modules should be conform with CLI specification. CLI stands for Common Language Infrastructure and is a platform that includes generic functions for exception handling, garbage collection etc. CLI is build into .NET Framework, this way multiple programming languages are supported. List of supported languages is limited and consist out of number of exotic languages like Boo, A# along with CLI derivative of popular languages like Java, Python and Ruby.

Radio Amateurs Radio Amateurs are encouraged to participate in the project, therefore every QB50 member is required to provide CubeSat Frames Decoder, Packet Decoder and data Viewer to VKI who will distributed it inside radio amateur community.

3.2 GENSO Network

GENSO stands for Global Educational Network for Satellite Operations and is an international initiative to form a worldwide network of the ground stations for spacecraft operations. In order to assure flawless functionality each ground station require a standardized communication software, which is able to transmit (uplink) and receive the data. The initial work dates back to october 2006, with first results presented during kickoff meeting on november 2007.

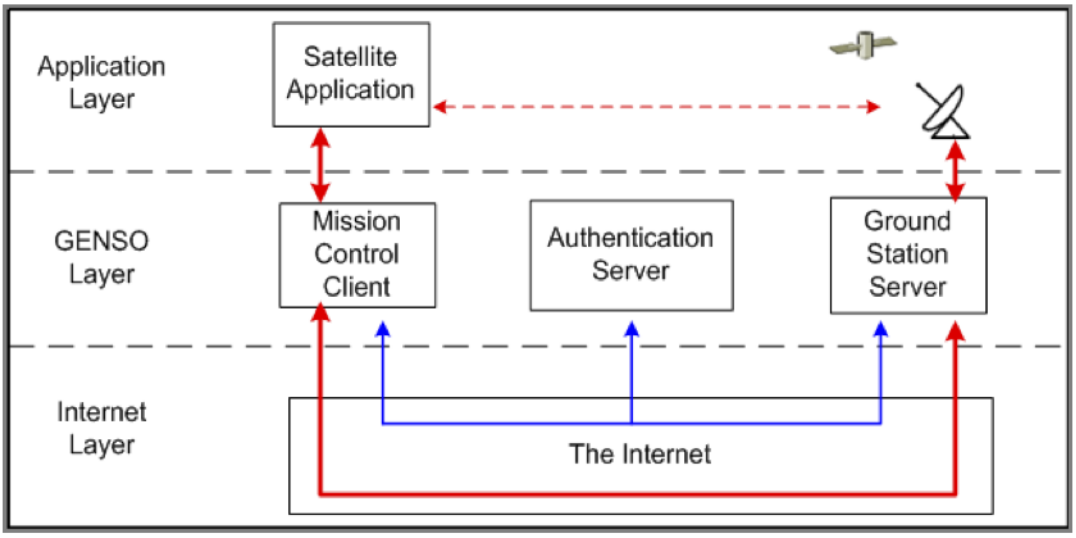

Figure 3.6: Genso Network [23]

The system architecture of the GENSO network consist of 3 parts: a Mission Control Client (MCC), a Ground Station Server (GSS) and an Authentication Server (AS). The Mission Control Client provides Satellite Application to the users, a means to control and retrieve the spacecraft data. Software consists of a standardized GUI and an interface for custom software. The Ground Station Server is essentially a dedicated server for Antenna control and Satellite data (packets) encapsulation to XML-format which will be networked across Genso Network. In order to increase hardware support software is integrated with open source Radio Amateur libraries such as HAM Radio Control (Hamlib) and GNU Radio.[23] The final element is the Authentication Server, it contain information on all GSS and MCC, and keeps track of the network changes. GENSO network supports different satellite communications (e.g. frequencies and keying), but because not all GSS are compatible with all communication features, AS will provide a lists for the users with compatible stations (location on earth globe, etc).

In contrast to EPFL ground segment, the network communication is not centralized, the data is sent in peer-to-peer (P2P) communication type between MCC and GS (this is indicated by red line in figure 3.6). Initially, a server request is sent to AS to obtain list of compatible ground station. Whereupon server returns a list of compatible GS where user is eligible for. Once a GS is selected by user, MCC is then connected to appropriate GSS directly (hence P2P). The connection is however not properly documented, so neither communication protocol is defined, neither the data sent. Most probably it is used to uplink the command or download the telemetry packets. Furthermore, it is unclear whether telemetry is stored on the ground station and requires user to download it manually, or any database is present on the MCC side.

GENSO network is a collaboration between many teams, with largest contribution of Stanford University (Mercury Network) and University of Tokyo (UNISEC Network). It is clear that GENSO is build up on the existing software of both network, therefore both Mercury and UNISEC will be analyzed.

3.2.1 Mercury Ground Station Network project (MSGN and FGN)

Mercury Ground Station Network development started around 1999 by Stanford University as ground segment for OPAL micro-satellite. The initial purpose of the software is controlling of the Ground Station Hardware (Antenna and Radio) ether manually (using SSH/command line commands) or semi-automatically by orbital prediction (with frequency tuning as the main goal). Additionally, received data was parsed and presented to the users through a simple web interface.

After OPAL mission, the project expanded and more ground stations were added. In 2001,

after ground segment expansion, the project was renamed to Federated Ground Station Networks (FGN). As project expanded, more compatibility was required along with increased reliability. In order to comply to the last requirement, team decided to apply virtualization to all Ground Station Servers. [9] The new, or rather improved, system design is based on the end-to-end principle, meaning that the lower layer of systems should support the widest possible verity of services and functions. In other words each system element can be controlled using both high level commands (e.g. "connect to a GS") and low level commands. Furthermore, any high level function can be implemented in any time in future without need to adjust low level elements.

The communication systems relays on a XML messaging system for command and control. XML is essentially a meta-language that can be adjusted to the software needs, therefore MSGN developers created a Ground Station Markup Language (GSML). It is a set of XML "tags" that can be interpreted by the servers and is intended for the machine-machine communications [12], but are still readable for a human. The GS server runs on a Virtual Machines, this is one of the most suitable solution to increase compatibility with different hardware (both PC and Antenna control).

The server is divided into functional layers Mercury Design Team [24]. The Virtual Hardware Level is meant as the control of low-level hardware resources. Essentially this level contains the hardware running the VM and the hardware required to communicate with the spacecraft[11]. The Session Level provides single station automation services. Essentially it is responsible for the scheduling, antenna high level control such as Auto Tracking and Auto Tuning etc. The scheduling service accepts reservation for the ground system usage and keeps track of system availability. From available documentation it seems that creation of schedule remains a non-automated task that should be performed by satellite operators. The session Level contain remote access server, that provides a secure and encrypted connection to the Ground Station Control (ssh?). Along with previously mentioned functions, the data processing is performed on the ground station side.

The network level captures the services of the federal ground station network. It makes it possible for FGN to register services for the GS and allow users to query for the availability of the GS. The second function of the network level is the Scheduler, which handles automated scheduling events. It enables static an dynamic optimization of the whole system network by controlling multiple GS and tracking the spacecraft passes.

Since the software is outdated (latest release 1.2 in 2003 Cutler [10]) and its mainly focused on the antenna control rather dan telemetry capabilities, no further software analysis was performed. Secondly, it should be noted that the design was never completed by the Stanford team. The paper describes architecture and high level functions that should be performed by the software, but the actual implementation was never completed.

It is clear that MSGN is reused in the GENSO design when functionality and documentation is compared. Secondly, the P2P XML communication within GENSO network is most probably the GSML of the Mercury network. Again, the scheduling and satellite tracking is undefined for both ground segments.

3.2.2 UNISEC Ground Station Network (GSN)

The initial projects dates back to 1998 as a collaboration between multiple Japanese Universities. UNISEC stands for University Space Engineering Consortium which is a non-profit organization with goal to support space related activities in the universities. GSCN is a network-based ground station system that consist mainly of Japanese universities.

According to official website [34] system architecture consists of two parts: Ground Station Management Service (GMS), GS Remote Operation Web Service (GROWS). GMS provides antenna control functionality and a protocol to connect and uplink data to the satellite. GROWS is essentially a graphical user interface, based on a static website (HTTP), and a communication protocol (XML) to communicate within the server network. Initially, [27] GSN was not meant to be a software package, but a protocol (software specification). In the proceeding paper [28], more attention was spent on the integration with other universities and a ground segment architecture has been proposed. It should be noted that design mainly focuses on the antenna/GS control and uplink, and not on the downlink and information storage. Nevertheless it is interesting to

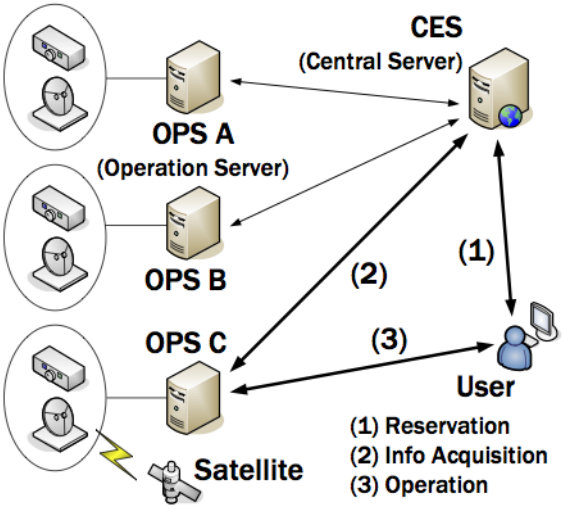

Figure 3.7: UNISEC GSN [28]

investigate the overall system architecture. Two architectures has been proposed, one based on distributed and one on centralized methodologies. In distributed system each GS Server contains all user information and the authentication (whether user is eligible to connect or hardware supports communication link) is done on the GS side. The selected centralized approach 3.7, consists of a dedicated authentication server that handles all user requests along with scheduling (who can access what server when).

Latest system update was on July 18th 2006 with organization of "The 1st International Workshop on Ground Station Network". From available information, no further development was done since then. It makes sense since both GSN and FSN are predecessor of the GENSO network. It is clear that network architecture of GENSO network is based on GSN, the presence of Central "Authentication" Server and the overall communication scheme relates closely to UNISEC GSN.

3.3 Open-Source Off the shelf software

Instead of using complete software solutions and comply to design decision of other developers, it is possible to combine existing separate utilities.

3.3.1 Wireshark

One of the functions of the telemetry server is processing of the incoming data. In order to do so, the incoming frames has to be decoded to human readable data. In the Delfi-n3Xt implementation, frames are dissected directly in the processing script. This is a robust solution as only two types of frames were broadcasted by the satellite. As discussed in chapter **??**, the DelFFi mission could use a higher number of reconfigurable frames Schoemaker [30] making downlink more efficient. In this case hardcoded frame dissection is no longer a feasible solution.

Wireshark is a state-of-the-art package dissector with enormous user base consisting mainly out of IT professionals. Wireshark is designed to capture network packets and to display the captured data as detailed as possible. A network packet consist out a number of layers, with actual data embed in the Application layer. In order to explain the Wireshark working principle a HTTP request packet will be used. A packet is its essence a stack of layers, each containing information HTTP-request is stored in the Application Layer and contain actual command (request). One level higher (Layer 4), contains TCP information. TCP is a transmission protocol is used for the connection definition (e.g. port number etc.). Layer 3 contains IPv4 information and contains address information (IP). Layer 2 contains Ethernet information. Ethernet protocol ensures

that the hardware within network can read the content of the packet. Level 1 is the actual frame. Each layer contains information about layer below, for example Layer 3 (IPv4) contains information that Layer 4 is TCP. In order to display packet data, Wireshark reads each layer, select correct dissector, dissects the data and proceeds to the next layer. Wireshark contains number of dissectors and is it possible to create a custom dissector which can read the data within "data" field of AX.25 frame. However, this is only possible if header of AX.25 frame defines the encoding of the data field.

The dissected data can be visualized graphically or be exported in predefined format. It is a great tool for decoding the raw frames that can be run both on the server and client side. Furthermore it is possible to integrate the Wireshark to the processing part of the software by using 3rd party software such as Sharktools Awile [3].

In order to address the flexible telemetry requirement, the type of the flexible frame should be defined in the header section, specifically in Protocol Identifier or Control bits. This is required by Wireshark in order to be able to slice the data in the Information Field of the frame. EPFL software and QB50 in general predefines that AX.25 frame and require both filed to be filled in with a certain value.

3.3.2 YODA++ (leave it out?)

YODA (Your Own Data Analysis) is a semi-automated system of the data handling and analysis developed for PAMELA space experiment which is design to measure matter and antimatter components of the cosmic rays in space. The downlink component of the ground segment is controlled by launch and operations provider (NTSOMZ), YODA system is therefore only responsible for data storage and processing. Raw data storage is located on the receiving side and is read by Raw Reader. The Root files storage is a directory containing ROOT libraries. ROOT is a object-orientated application and libraries developed by CERN for partially physics data analysis. Package can be used to visualize the data to visualize distributions, perform curve fitting, vector computations and statistical functions Rademakers [29]. The contents of the root directory contains objects to perform the processing: Packet Type Objects (PTO, wrap data of the satellite payload modules), Algorithm Type Objects, Utility Methods Class (provide an interface between Yoda reader and the ROOT objects) etc. Since all data processing is based on ROOT which is a very targeted processing unit, the whole design is not relevant to the DelFFi mission and will not be discussed in detail. The design of YODA++ is however very similar to the Delfi-n3Xt design. Note the presence of multiple storage units and separate readers.

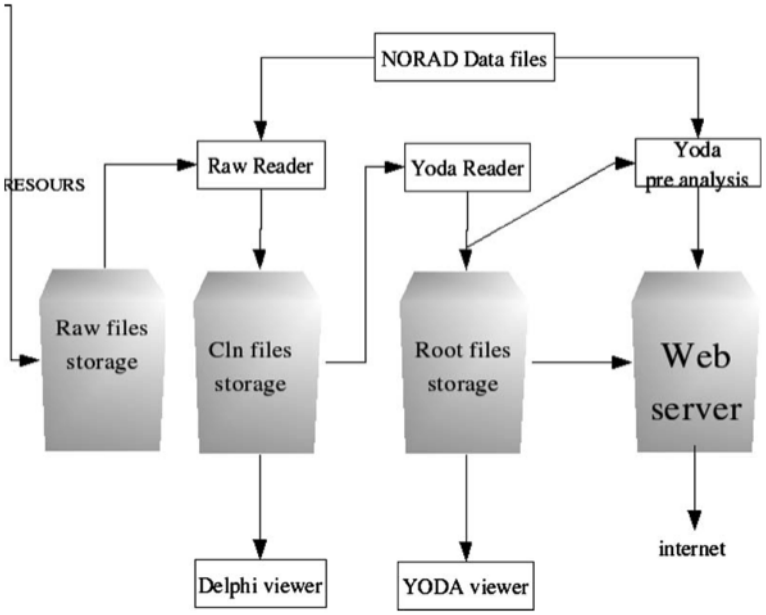

Figure 3.8: YODA++ system overview Rademakers [29]

4 Delfi-n3Xt subsystem level overview and analysis

Data collection, storage and visualization are single the most important telemetry server functions. According to software design guidelines [13, 14] it is considered to be a good practice to have a strong cohesion and low coupling. Both functions are therefore separated both on software as on hardware level. According to the theory of the current software systems engineering key components should be separated in order to reduce risk and increase safety. The Delfi-n3Xt telemetry server is designed according to this principle, both data collection and storage are separated as can be seen in the Figure 4.2.

Data acquisition for the Delfi-n3Xt is based on the radio amateurs. Radio amateurs are radio hobbyists that are licensed by government authorities to use allocated portions of the radio spectrum for non-commercial activities such as technical experimentation or personal communications. During the Delfi-n3Xt operations, radio amateurs are able to decode received data using DUDe client, while data is been transferred to the Delfi-n3Xt server. The functionality of the DUDe client is discussed in the section 4.1. The transmitted data from DUDe client is stored on the Delfi-n3Xt server, directly into the database. The processing of the data is discussed in section 4.2 , while the database design is presented in section 4.3. Once data is been processed, users can access it from the internet. More information on this subsystem can be found in section 4.4.

4.1 Data collection software

Data collection is separated from telemetry server both functionally and physically. As described by short functional description [SLR0320] and earlier in this chapter, the data signal from Delfi-n3Xt is received by Amateur Receivers all around the world, using DUDe (Delft Universal Data extractor) client. The received raw data is demodulated and the processed by the client and whereupon is displayed to the RA using Graphical User Interface (GUI).

Figure 4.1 visualizes the DUDe-client functionality. The incoming data is demodulated by the DUDe client. The DX-frame is decoded and presented through GUI to the Radio Amateur. Radio Amateur is able to see the received data and transfer it to Delfi server in an automated way.

The DUDe client has a build in authentication system. Upon DUDe start up, user is required to provide his credentials. The username and the hashed password are verified on the backend (server side). Each user is registered in the telemetry server database, this way it is easy to correlate telemetry data to the radio amateur, who provided the data to the server. This way DUDe client is separated from credentials database what forms an extra security layer.

If credentials match, DUDe client will check whether offline files are present in the system. If necessary the offline files (essentially frames received while not connected to the server) are injected in the server database. When this process is completed, DUDe client engages the sound card and checks for the incoming packets. The process of generation of offline files is however very ambiguous, it is unclear how files can be created as a database connection is required in order to engage the DUDe client.

4.2 Data Handling

Data Handling is defined as processing and migration of data within the telemetry server. The main processing tasks are checking the data for errors, merging the matching frames and con-

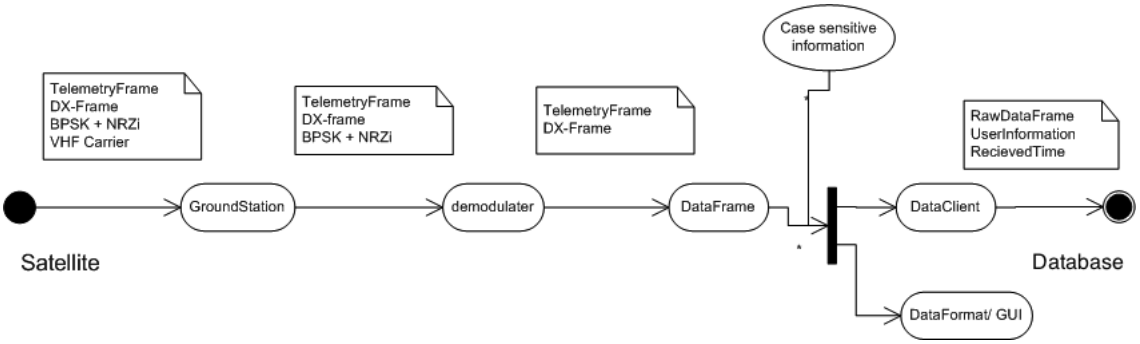

Figure 4.1: Overview of the telemetry client (DUDe) Kuiper and Hernando Bravo [22]

version of raw data to physical quantities.

4.2.1 Error checking

In order to assure that only the correct data is processed, the incoming frames are checked for the errors. This is done in two steps, first a frame-check sequence is performed, followed by duplicates control. The frame check is based on Cyclic Redundancy Check on 16bit checksum. Once the frame is verified to be a legitimate AX.25 frame, the satellite name field is read. If sender code in the telemetry frame is identified as Delfi-n3Xt, the frame payload is read and system determines whether data is already present in the system (raw data database). This is done on the basis of frame counter and the timestamp. Secondly, database contains a counter for each frame, this is used by the system to identify the latest available frame. In case a new duplicate frame is present, the counter of given frame is automatically updated to the latest counter. Faulty frames, along with existing duplicates are moved to the discarded database. Next to previously stated tasks, the error checking script can place tags to increase traceability of the processing status. The script is executed on regular basis by means of crontabs. Additionally it can be run manually to force recheck the whole database.

4.2.2 Frame merging

As stated earlier in chapter 2, the Delfi-n3Xt telemetry data size exceeds the single frame size limitation and is therefore sent in two separate frames. Because the decision on usage of multiple frames was made after the database and processing software design was completed and implemented, the actual software had to be adjusted. In order to solve the both matching frame parts share the same id and aren only identifiable by content of the frame.

In order to identify already merged frames in the database, all non identified frames are marqued with “0” tag. Essentially, tag is an extra row in the table that contains an integer. The merging script simply selects a single frame part with “0” tag and checks all other available records for a match. Once match is identified, tags of both records is replaced by “1”. Both frames are then merged and passed to the merged database. The merging process simply adds the second frame part at the end of the first frame part.

The main issue with this approach is that all non-processed frames are looped, for example if second frame part is not received, the system will keep reading first part on every iteration. This also means that the content of the first part will never be processed. Secondly, the frame id was used in initial design to perform frame filtering. Because a double frame is created with identical id, this approach can no longer be applied.

4.2.3 Processing

As the data frame is stored in binary format, it needs to be decoded to human readable and processable format. The data part of the frame is build up from (sub-) fields, each of which can be decoded to a decimal value which corresponds to a certain measurement of a subsystem. The

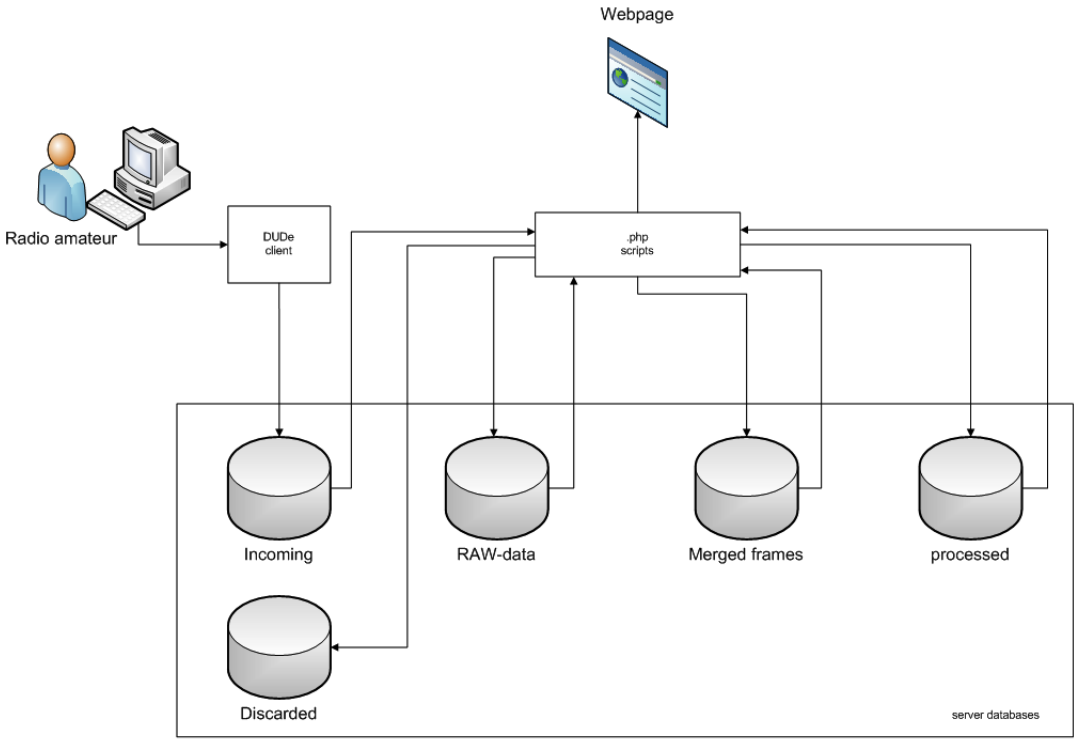

Figure 4.2: Top level overview of software ground segment architecture

bitwise position of each field is defined by an external file. It should be noted, that when extra field (extra measurement) is defined, the processing database would need to be updated to facilitate new data. This is performed automatically by the software. It is therefore important to define all fields beforehand, otherwise whole database should be repartitioned after the adjustment. A clear disadvantage of the Delfi-n3Xt implementation is that it does not facilitate the flexible telemetry frames. Secondly, the software is not optimized for user experience. For example, the adjustment of the frame layout is performed through the GUI. But adjustment of the processing computation definitions is separated and requires user to adjust the source code.

4.3 Data storage and general MySQL discussion

As documented by [8], in the current server implementation data storage and management is performed by MySQL database management system. SQL, structured query language, is a declarative database language that is by design optimized to for relational databases. In the relational databases that data is organized in tables. Each table is build up from records each stored in a field. Fields from different databases can be joined to create a temporary (output) database. The term relational database comes from fact that data can be brought into "relation" to each other to combine it.

In the figure 4.2 overview of high level operations is presented. It can be seen that this implementation consists of multiple databases, one for each data processing step (see section 4.2). The usage of multiple databases with inter-database operations is seldom used due to possible efficiency issues ([1], Cabral and Murphy [5] and Ab and Street [1]). In order to reduce the read-time of the database, caching is applied. A small, fast to read copy of the database is created that can be read faster than actual database even while database is been updated. Database update is defined as a change in the database, for example as appending of new data (a new row). Caching may become less efficient with increasing number of incoming data as a new cache file has to be recreated on each data injection. An another way to increase the performance is

indexing. Each table contains a key that contains a unique identifier for each record. By creating an index of the unique keys, MySQL is able to lookup required field faster and read only the required record instead of whole database.

Another important tier is the storage engine used (Cabral and Murphy [5]). For example, the older (MySQL versions 5.0 or lower) MyISAM storage engine priorities writing above reading statements. This means that reading statements (SELECT statements) would not be executed while writing is performed, as table would be locked up. Technically, following scenario may render server unfunctional for extended period of time if MyISAM would be used:

1. Satellite transverse region with high number of amateur receivers, these results in a high number of duplicate frames sent to server. Many receivers send data twice to make sure that frame is transmitted properly.
2. Each incoming frame is stored into incoming database, without performing duplicates filtering.
3. php script makes a connection with incoming database, which creates an open file on the server. MyISAM puts read request on hold (queue), as writing process locks incoming-table.
4. php script requests read access to the RAW-database. Again, it is put on hold, after an open-file is created. Because databases form a waterfall model, every sequential database request is put on hold as well.

Initially, the bottleneck would be formed on the incoming database, which will gradually shift to the final database in the database "waterfall". When maximum number of connections or openfiles is reached, server will refuse incoming connections which will result is loss of data.

InnoDB, engine used for the most tables of the telemetry server, is more liberal in its locking procedures but will lock the index file while cache is being updated. Scenario presented above, would have less impact on the database as InnoDB allow simultaneous reading and writing of the actual database. However, in some cases, such as frequent updating of the tables (hence frequent rebuilding the cache and index file) may cause unexpected behavior Cabral and Murphy [5]. Next to that data is been migrated between databases as it undergoes processing, each migration requires new database connections. On idle, with no incoming data-frames, database runs on average around 50 queries each second. These results in many connections to be (re-)opened every second. Each connection creates an open file, number of which is limited by the operating system.

Generally, when MySQL is used, the number of stand-alone database should be kept minimal as the increase in semantic relations is preferred. This will result in the complexity increase. However, according to Endes [14] software should have strong cohesion and keep coupling low. This approach is well known as Constantine law: "... reduce the complexity of programs by dividing them into functional modules. This can make it possible to create complex systems from simple, independent, reusable modules. Debugging and modifying programs, ... managing large programming projects can all be greatly simplified..." (Stevens, Stevens et al. [32]). This approach is often applied in software development and is known but modern term "modularity". It forms the core of the object orientated programming languages. Clearly, different strategies would result in different outcome. In case of the telemetry server a framework is preferred above highly-tailored design. This means that software should be easily adaptable to the future missions, decreasing the development time. One of the consequences of this decision is requirement on the overall system complexity. The Delfi-n3Xt waterfall database is therefore easy implementable as structure is fairly easy. A clear disadvantage is the high amount of data duplicates and less stable system under load.

4.4 Data Distribution

In order to deliver data to the users, system contains two interfaces. First interface, is the real time data presented by DUDe. Second interface is the Delfispace website. As discussed earlier,

different users are identified by the system. Radio Amateurs upon authentication are able to see their personal submitted telemetry and the overall telemetry statistics. The operators on other hand are able to see current status of the satellite and graphical representation of historical data. For processing purposes, operators can download the data in .csv (Excel) format.

The website is considered to be "static" as the data is being generated and updated on the server side. This means that data presented in the website is only updated when server updates information in php files. For the selection of the content in the Operator view, website relay on the backend and only small amount of tasks are performed on the front end.

5 Trade-off

The Ground Segment and the telemetry server in particular, contribute to the overall to the overall DelFFi mission outcome. For the upcoming QB50 mission, every participant has an unique opportunity to make use of flight proven Ground Segment. Therefore in order to increase overall mission success, a trade-off is required between available software solutions.

5.1 COTS

In order to fully assess situation, a short introduction will be given on usage of the commercial off-the-shelf software. Usage of COTS software in its essence means that the software development is based on requirements defined and analyzed by a 3rd party developer. The requirements are formed on the needs from users in different environments, which may or may not change in the future. Generally, an initial misalignment in requirements will be present, that will be solved on long run by adjusting the user workflow of the users or addition of an extra software packages. On other hand more generic requirements, mean that the software may contains functionality not yet identified by the DelFFi team or addition of an extra functionality later in the project.

One of the main advantages in usage of COTS software or components is testing and validation. The overall system reliability is superior compared to in-house applications. Secondly, the software is often available in a shorter period of time.

For large and advanced software packets and external involvement is required to achieve positive results. For example, the software configuration and fine-tuning. This is often required, as the source code is not presented to the users or when functionality is not fully documented. It should be noted that when the source code is edited, software can no longer be treated as COTS and testing&validation is required. Secondly, edited software cannot be updated as easily or even impossible. Another aspect is reliance on one company. This dependance may influence operations in the future, for example when development is discontinued.

The developers of the SEER [16], software to predict cost of COTS product usage, identified a great number of COTS risks and issues. Many of these issues are related to IT environment, but some are strongly related to the EPFL software and DelFFi mission:

- Component that may not fully integrate with the architecture
- Component functionality must be modified
- The consequences of the implementation and wrappers is hard to predict.
- Vendor may cease further development and supply of the product.
- Clients may not be allowed to modify the software

5.2 In house software

Main advantage of in the house software is flexibility. By designing software from scratch, a set of requirement is specifically set on the users needs. It is however challenging to determine full set of requirements, some of which will only be discovered after initial implementation. The keyword of the in-house software development is the control. Software is developed according to the users needs and the architecture is tailored accordingly. The performance is not fixed as in the COTS implementations and there is plenty of room for optimization. The source code is also available and the bugs can be resolved in a shorter term.

In-house development can be seen in many cases as a re-invention of the wheel. The quality of which is highly dependent on the skill level of the developer team. In this case, the re-usage

of the legacy software may seem as a good alternative. It should however be noted that re-usage of existing software often requires redesign of certain components. Reverse engineering of which may undo all expected time advantages.

5.3 GENSO

GENSO network is discarded from ground segment software list on the early phase. This is mainly caused by lack of updates on the status of the GENSO system design. From the available sources, it is clear that the development is delayed and won't be completed before the QB50 launch. The single ground station supposed to make use of GENSO software is COSMIAC, Configurable Space Microsystems Innovations & Applications Center at University of New Mexico in Albuquerque, NM [7]. The mission Trailblazer, was cancelled after 2 unsuccessful launches, and GENSO functionality was never not fully verified.

5.4 Trade-off EPFL and Delfi-space

In order to make the selection between usage of the Delfi en EPFL software, a trade-off process should be performed. First technique is based on the assessment of the available documentation and user experiences with the program. Second approach is based on the analysis of the source code [2]. It should be noted that both EPFL and Delfi software are hard to evaluate equally, as no source code is available for EPFL software (software not yet complete) and the documentation of both software packages in incomplete. Another technique is the evaluation of the actual software. It is performed on the basis of user cases and mission scenarios. This was however not possible at this stage of the process.

According to ISO-9126 standards Quality of the source code is assessed on the basis of Functionality, Reliability, Usability, Efficiency, Maintainability an Portability [2]. The assessment is usually performed in 4 steps. First, Code Quality is assessed on the technical parameters such as reliability and complexity. Second step is architecture assessment. During this step the overall consistency, modularity and intended-to-actual architecture assessment is performed. Third step is focused on Internet and Web Services. The main goal of this step is to check the vulnerability of the system (stability), cross-platform compatibility and performance. Last step covers the Databases. It focuses on integrity of the databases and model modularity.

Clearly, this approach is hardly applicable for the trade-off process due to lack of the actual code and ambiguity of ISO-9126 concepts. Quality assessments techniques of ISO-9126 are known to be misleading [2] and result in double count of certain parameters (for example function implementation completeness - function implementation coverage). Therefore, the documentation based trade-off is most suitable for the current situation.

5.4.1 Suitability

Before suitability of the EPFL software can be assessed, it should be noted that the processing of the frames, such as frame decomposition should be developed by the QB50 participants. Secondly, the user interfaces are not clearly defined and every extra processing step should be developed by the QB50 participants. This means that the EPFL software should be extended before it can be applied for DelFFi mission.

Both (bare-bone) EPFL and Delfi-n3Xt telemetry software does not comply with the DelFFi Telemetry Server requirements. In case of Delfi-n3Xt, software should be updated to allow WOD to be sent to the QB50 server. From user experiences, it was determined that the user interface should be improved, hence the "website" requirements. The EPFL software only provides a basic functionality. All user interfaces prescribed in the requirements should be developed.

5.4.2 System and Software Flexibility

Both software packets can be adjusted to mission specific needs. The EPFL software requires knowledge of .NET framework developing in order to perform any adjustments. It is unknown

Flag	AX.25 Transfer Frame Header (128 bits)				Information Field	Frame-Check Sequence	Flag
	Destination Address	Source Address	Control Bits	Protocol Identifier			
8	56	56	8	8	0-2048	16	8

Figure 5.1: AX.25 frame

whether server source code will be provided and whether only plug-in types of the software updates are possible. Delfi-n3Xt telemetry server implementation is less thoroughly documented, it requires reverse engineering in order to adjust the software.

The flexible telemetry frames, as described by Schoemaker [30], will require ether an unique identifier or header definition. This is required by the processing software, as the bit string would not be decodable otherwise. The flexible frame layout identifier or a frame id can be written in the Protocol Identifier field of the AX.25 frame 5.1. This way AX.25 frame utilize the unused 8 bit and frame become decodable by the Wireshark.

EPFL software require a fixed AX.25 Transfer Frame Header lay-out. Specifically, the Protocol Identifier should be set on 0xF0. This requires the frame id to be stored in the Information Field and waste 8 bit of frame. Another point of the ambiguity is the ground station, according to EPFL documentation, every ground station should fulfill needs of every participant, so the GS are scheduled and controlled by EPFL.

5.4.3 Compatibility to Future missions

Delfi-n3Xt and Delfi-C3 telemetry software are both in-house products. It can therefore be modified and reused, furthermore it provides a guideline for the future design. The data acquisition is based on the radio amateur contribution that are unlikely to cease the collaboration in the near future.

In case of EPFL, it is not indicated whether both software and hardware can be used after the QB50 mission. The usage of EPFL software enables DelFFi to acquire telemetry data from Ground Stations of other QB50 participants during the mission. This expands the coverage and reduces the complexity of WOD downlink solution. However, because of the the complexity of the operations such as control of GS of the participant, it is unlikely that the Ground Segment will be usable after QB50 mission.

5.4.4 Reliability

As discussed earlier COTS is considered to be a more reliable and stable product. This is mainly caused by more severe testing and validation requirements. It should however be noted that the processing scripts are build and tested by the clients, rather than by EPFL, reducing the overall reliability.

Both systems contain a single point of failure in this design. In case of EPFL this is the EGSE Router, where in the Delfi-n3Xt it is the database and the database connector on the server side. The connector ensures that the DUDe data is written directly into the database. In case of connector failure, the DUDe client would not be able to communicate with the database and Radio Amateur would not be able to login into the DUDe client and data would not be received by the server. It is however possible for Radio Amateur to store the data and re-transmit it once connector is up and running again. In case of the EPFL software, EGSE Router failure will cut the communication with ground stations in both directions. It is unclear whether GS contains a local database for the data storage, hence the consequences of EGSE Router downtime can not be assessed.

5.4.5 Documentation and system complexity

The available EPFL documentation is ambiguous in many aspects, what hints towards the incompleteness of the final software. This combined with a high system and architectural complexity may results in unforeseen implementation delays. The messaging protocol used by EGSE Router is designed for a different application and is considered to be overly complicated [19]. This requires a in-depth protocol knowledge before be able to be applied.

EPFL is a highly integrated system, that controls the antenna, performs schedules and transmits the telecommands. In case of Delfi-n3Xt and Delfi-C3 most of those functionalities are performed by separate software systems. This reduces overall risk and ensures system flexibility.

5.4.6 System coverage (end-to-end)

Both EPFL and Delfi-n3Xt can be seen as end-to-end software solutions. EPFL software has an overall larger geographical coverage as it both provides telecommands and GS control capabilities. By collaborating with the GS of all participants EPFL system ensures the all GS are scheduled automatically and WOD quota of all participants is met.

5.5 Summary of Software findings

The Delfi-n3Xt telemetry server is relatively simple in its architecture but lacks the functionality required for the DelFFi mission. The main issue is the incompatibility with the flexible telemetry frames. Addition of the second satellite would increase processing load on the satellite and would double number of the databases in the system. The whole design is scalable, which is achieved at expense of higher server load. The processing scripts are hardcoded, this requires extensive software adjustments in order to comply with flexible telemetry frame design. The data visualization is limited and contain a number of shortcomings. The overall reusability of Delfi-n3Xt software is therefore assessed to be low, as implementation is poorly documented. Work required to reverse engineer the software implementation and bug hunt required exceeds implementation timeframe. Therefore, rewriting the code by containing the current architecture is considered to be the best alternative.

The central checkout origin of EGSE Router and hence whole EPFL design is still visible in the implementation. Arguably, the additional complexity introduced by that, is unlikely to pay off in terms of the the system performance when it is used in for the DelFFi mission. The routing of the data and the actual architecture is logical for the checkout systems as each client (= sensor) is well defined before actual operations. In case of DelFFi, large share of data is contributed by Radio Amateurs. This has 2 consequences. First, a high number of amateurs are located within Western Europe, therefore system will receive number of duplicates from different GS. Because the GS are correlated to frames with a type 8 report, MCS will receive number of type 8 reports for a identical frames. It would not be possible to correlate a specific duplicate to a specific GS, as every duplicate frame would have an identical timestamp. Secondly, it may be possible that frame is damaged and certain amount of measurements is not accessible, again it would be hard to determine the origin of the given frame. Because the documentation of MCS software does not define this case, it is unknown how system would react to the duplicates. In the last update of the documentation [31] the EPFL indicates the Radio Amateur integration into the whole architecture. The role of Radio Amateur (RA) in the mission is rather ambiguous, as it can be seen in the figure 5.2. According to which, Radio Amateurs are only eligible to receive beacon data (BEA) only. Science (SD), Whole Orbit , Payload and Housekeeping data can only be received by GSx, a Ground Station part of the EPFL network, which is not aligned with the current policy of DelfiSpace. In order to resolve this issue the DUDe client can be reused with a back end at TU Delft side, that injects data directly into EGSE network.

Another point of ambiguity in EPFL documentation is the lack of high level information on the system architecture and overall system integration. The main point of ambiguity is mainly the integration of the ground stations (of all QB50 participants) within the local EGSE networks of each participant. Fact that it is undefined, arguably means that the design decisions are

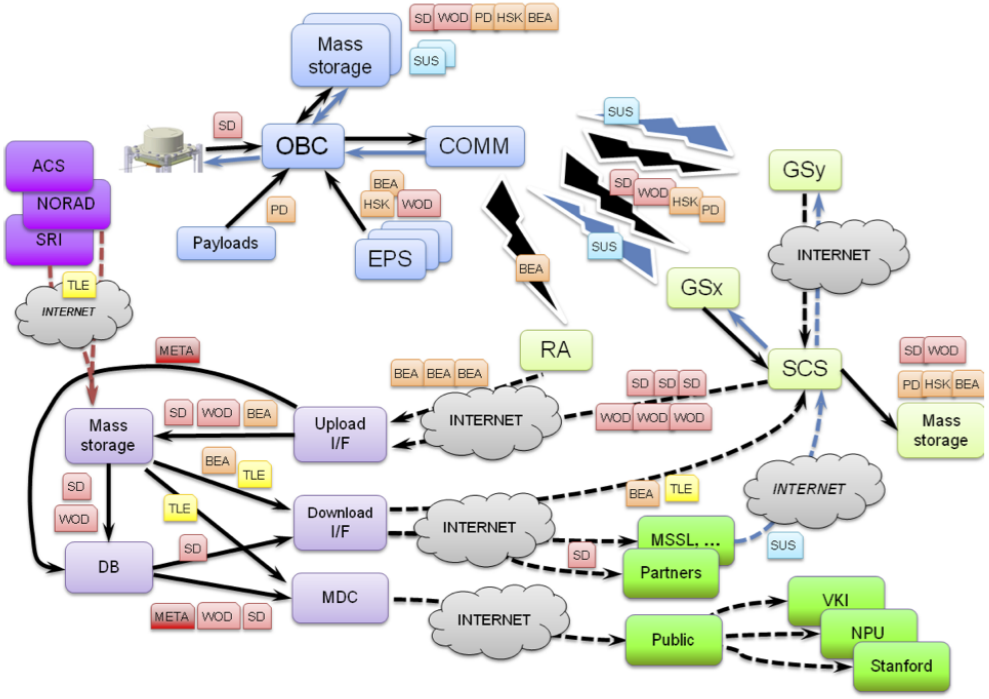

Figure 5.2: VKI proposed End-to-End data flow for a QB50 CubeSat [31]

not yet made. Furthermore, the design documentation [18] suggests automatic scheduling of the ground stations, what implies the need for the control software for the antennas and telecommand equipment. It is unsure whether satellite is tracked automatically or user action is required. In latter, the need for telecommands messaging is less relevant.

From all available EPFL documentation on the messaging protocol and system functions, it is clear that the security is been neglected. The core of the EPFL software was designed as a local network inside a secured IT environment and contains no authentication services. Because the EPFL system is operating over the internet, for instance the data communication between GS and EGSE router, it is most likely that the authentication services will be added later in the project. This is required to keep system robust and prevent mis-configurations. As stated in the ILT experience report[19], latter is rather complicated and requires a better traceability of the messaging system. From development perspective the command tokens are developed to increase the traceability, while the unique tokens should be generated on the client side. Extra client side intelligence is required to solve this problem. The client side control is however limited to plug-in scripts, so it is unknown how far traceability is present in the system.

5.6 Trade-off

The goal of this paper is to select the most optimal software solution to increase the overall mission success. The contribution of the software to the mission success can be evaluated using the utility tree, as it elicits and prioritizes the quality attributes that contributes to the project. The utility tree for a generic software product can be found in the figure 5.3, it emphases the major fields of interests and will be used as the guideline for the trade-off criteria selection.

By analyzing the mission requirements 1, a set of selection criteria has been determined that captures the mission need..

- C1: *Suitability*

Suitability is in its essence measure of compliance of the software system to the mission requirements. Metrics of this category can be extracted from the mission technical documentation,

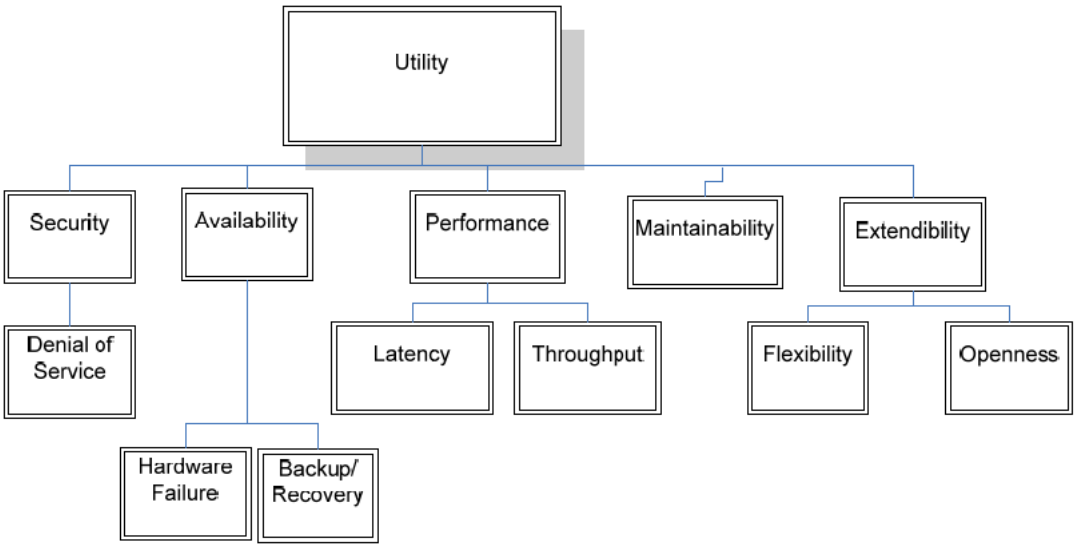

Figure 5.3: Utility tree

highlights of which are summarized in chapters 1 - 4.

- C2: *System and Software Flexibility*

Flexibility is a measure of adjustability of the software to a new mission requirement. Technically, this means the ease of adjusting the telemetry-frame definitions, processing and visualization parts. The main driver of this criteria is Maintainability.

- C3: *Compatibility to the Future missions*

Creating a reusable product, reduces the development time of the future missions and increases the interest from third party developer. Engaging which may increase the overall quality of the product and arguably the success rate of the future missions. The main metrics of this selection criteria is Extendibility, which goes along with scalability of the final product.

- C4: *Reliability and Risk*

Reliability and Risk form the trustworthiness selection criteria. It involves many characteristics such as risk in the overall design and architecture, but also metrics as the maturity of the application and the skill of developers. The main driver of the Reliability and Risk criteria is Availability.

- C5: *Documentation and system complexity*

Provides an insight on amount of work hours to adjust, install and configure the software.

- C6: *System coverage*

System coverage selection criteria focuses on the integration of the final product and additional software required to make system work.

In order to perform the trade-off Analytic Hierarchy Process will be applied. AHP is selected due to the ability of decomposition of the problem into a set of more comprehended subproblems. Process contains 2 deliverables, relative ranking of the criteria and relative ranking of the products. First, the relative ranking of the selection criteria will be determined. The table **??** combines relative ranking of each criteria set to specific reasoning. Criteria definition and coding can be found in section 5.2. The software solutions considered as a valid alternatives are:

- S1: Adjusted Delfi-n3Xt software

- S2: New implementation based on Delfi-n3Xt architecture
- S3: EPFL software

As stated earlier, the selection criteria are applicable for many companies and different software solutions. The challenge is therefore present in the field of the mutual weighting of the criteria rather than criteria themselves. The choice of the mutual weights of the selection criteria reflect on the priorities within the development of the current and the future missions. This is highly related to this mission specific critical points that define the overall success of the mission. The AHP process is well known in the literature [21] and will no be further explained in this document. The weight selected for the mutual weighting are on scale 1 to 5, where 5 represents the highest grade and hence highest importance.

The mutual weights can be found in the table 5.1, the rationale is summarized below:

- C1 - C2:

It is crucial to comply with the software and system requirements, when it is compared to flexibility of the software components, it becomes apparent that latter comes on the second place. Hence weight of 4. It should be noted that Flexibility represents ability to adjust the software while mission is running, for example addition of an extra processing frame or adjustment of the processing script.

- C1 - C3:

It is more important to fully comply to the current mission requirements, as the future mission requirements are unknown at the design stage and adding extra functionalities to fulfill guestimated future needs doesn't contribute to the success of current or even the future mission. A flexible framework usable for all future Delfi-Space missions is preferred on the long run (C3), which still required to fully comply with actual requirements (C1). Hence weight distribution of 5 - 4.

- C1 - C4:

It is straightforward that the reliability is more important than suitability, therefore a weight of 4 has been assigned to the C4. Suitability has weight of 3, as a very stable and reliable software still needs to perform mission required tasks.

- C1 - C5:

A less documented software may require more installation and configuration efforts, but complying to requirements is still far the most important. Documentation is vital to make system maintainable and usable for the current and future missions. In case of EPFL software, a lack of the specific installation documentation may result in a unusable product which may result in downtime of the ground segment. Hence weight of 4 on the C5.

- C1 - C6:

The overall coverage is less important compared to the system requirements as long as all system requirements are met. Therefore satisfying the requirements is prioritized. Coverage was weighted 2, as a well integrated solution is preferred.

- C2 - C3:

As C2 mainly focuses on the software adjustments (like an extra frame definition), C3 focuses on the overall capability and compatibility. It is concluded that a long-term supported software is more favorable as every mission would require a complete telemetry server redesign ("Reinvention of the wheel"). Therefore an established framework (C3) is preferred, hence weight of 4.

- C2 - C4:

Reliability is a major factor in a space mission. A failure of the telemetry server would make it impossible for the operators to receive and processes satellite data, hence weight of 3 for C4.

Table 5.1: Relative ranking of the selection criteria

ID1	Weight 1	ID2	Weight 2
C1	5	C2	4
C1	5	C3	4
C1	3	C4	4
C1	4	C5	3
C1	4	C6	2
C2	2	C3	4
C2	1	C4	3
C2	2	C5	2
C2	1	C6	2
C3	2	C4	5
C3	1	C5	1
C3	1	C6	1
C4	2	C5	1
C4	5	C6	1
C5	2	C6	1

- C2 - C5:

Documentation and system complexity is considered to be equally important when compared to system flexibility, this is mainly driven by the need of the system reusability.

- C2 - C6:

System coverage is considered to be more important compared to adjustability, as less development time is required (e.g. In case of EPFL software - control software for antenna)

- C3 - C4:

The reliability has a major impact on the overall mission outcome, therefore when it is compared to the readiness of the software to the future mission, the latter weight as set to 2 and C4 to 5.

- C3 - C5:

Documentation and system complexity have an equal impact on the mission as C3 as they both contribute equally to the re-usability of the software solution.

- C3 - C6:

Both are weighted equally as both contribute equally to the mission outcome.

- C4 - C5:

Reliability is more important than C5, but a complex system endues the risk significantly. Therefore weighting factor is 2-1.

- C4 - C6:

Reliability is preferred above end-to-end software, as it has most impact on the mission outcome.

- C5 - C6:

Well documented or low complexity system is preferred over end-to-end software as it increases the chance of the future development.

The relative ranking of software solutions can be found in the table 5.2 with rationale summarized below:

- C1:

  - S1 - S2: Both systems will comply with the requirements. A new implementation is however more favorable due to compliance with flexible telemetry data on the database design level.
  - S1 - S3: Both systems comply with the requirements after updates. However, as system is adapted for QB50 specifically, it has a higher score.
  - S2 - S3: A new implementation is more in favor due to compliance with flexible telemetry data design.

- C2:
  - S1 - S2: Redesigned system is extremely adaptable to specific needs for the upcoming mission.
  - S1 - S3: EPFL software contains plug-in capabilities and is therefore slightly more adjustable.
  - S2 - S3: A new, well documented, design ensures that software is easily adjustable to user specific needs on every software level.

- C3:
  - S1 - S2: A well documented and redesigned system provides superior reusability as it requires less efforts to understand and adapt the system.
  - S1 - S3: Reusability of both system is limited. In case of Delfi-n3Xt system, the adaptations require time investment in reverse-engineering the code. EPFL software may not be available after QB50 mission.
  - S2 - S3: The main issue is uncertainty whether EPFL system may be used for the future missions.

- C4:
  - S1 - S2: As both systems are designed and implemented by students, the risk of both systems is comparable.
  - S1 - S3: EPFL is professional system, hence the risk is significantly lower. However, due to presence of multiple systems and relative short implementation time, the risk is still present.
  - S2 - S3: EPFL is professional system, hence the risk is significantly lower. However, due to presence of multiple systems and relative short implementation time, the risk is still present.

- C5:
  - S1 - S2: Redesign and re-implementation of the system is more favorable compared to adjustment of the existing system. This is due to reverse-engineering time delay and time required to find and fix existing bugs (by doing so unwanted complexity will be introduced to the system). By adding documentation on the high priority task a great improvement can be achieved with a redesigned system.
  - S1 - S3: EPFL software is more favorable as external aid can be requested. This way the amount of work is reduced. The documentation is however ambiguous on number of concepts and software is likely still in the development phase. The overall complexity of the system is high and configuration is prone to the errors. Hence weight of 2.
  - S2 - S3: EPFL software is more favorable as external aid can be requested. This way the amount of work is reduced. The documentation is however ambiguous on number of concepts and software is likely still in the development phase. The overall complexity of the system is high and configuration is prone to the errors. Hence weight of 2. A freshly redesigned telemetry server requires more development time, but requires less configuration (setting up servers, configuring and developing the plugins etc), hence the overall weighting of both components is equal.

Table 5.2: Relative ranking of the software solutions

Criteria	ID 1	Weight	ID 2	Weight
C1	S1	1	S2	4
	S1	1	S3	2
	S2	3	S3	1
C2	S1	1	S2	3
	S1	1	S3	2
	S2	3	S3	1
C3	S1	1	S2	5
	S1	1	S3	1
	S2	5	S3	1
C4	S1	1	S2	1
	S1	1	S3	3
	S2	1	S3	3
C5	S1	1	S2	3
	S1	1	S3	2
	S2	1	S3	1
C6	S1	1	S2	1
	S1	1	S3	3
	S2	3	S3	3

Table 5.3: AHP analysis results

System	Grade - Author	Grade - Participant 1
S1	0.17	0.33
S2	0.44	0.35
S3	0.39	0.31

- C6:
  - S1 - S2: The Delfi-n3Xt software is already present and is used for the mission, as part of it can be reused for the DelFFi telemetry server, both systems are equally integrated.
  - S1 - S3: EPFL software is ready to be used and covers most of the essential functionality. The system requires plugins to be able to decode the data. Therefore weight of 3 is applied.
  - S2 - S3: A finished and redesigned telemetry server covers all required functionality and has therefore slightly higher score compared to S1-S3 combination.

From the results found in the table 5.3 it is clear that an in-house developed product is more in favor from author point of view. In order to validate this finding and reduce the bias, the weighting process was performed again by a different DelFFi member. Participant was unaware of outcome and distributions found in the tables and based their decision solely on the information presented in the technical documentations and summaries in this paper. It can be seen that the overall result is still in favor of in-house application, but differs in magnitude. This is caused due to more conservative selection of weights.

AHP method doesn't provide a final answer to the question, it is rather a tool to support the argumentation presented in the previous chapters. From available documentation and description of the EPFL software solutions, an in-house developed software is more advisable. It adds to certain extend risk to the system, but provides unmatched design flexibility and due to presence of the source code, contributes to the further refinement of the product.

Bibliography

[1] Mysql Ab and East Street. *Administrators Guide*. 2005. ISBN 0672326345.

[2] Hiyam Al-kilidar, Karl Cox, and Barbara Kitchenham. The Use and Usefulness of the ISO / IEC 9126 Quality Standard. pages 126–132, 2005.

[3] Omar Awile. Sharktools. URL `https://github.com/armenb/sharktools`.

[4] William A. Beech, Douglas E. Nielsen, and Jack Taylor. AX . 25 Link Access Protocol for Amateur Packet Radio. (July), 1998.

[5] Sheeri Cabral and Keith Murphy. *MySQL Administration Bible*. Wiley, 2009. ISBN 978-0-470-41691-4.

[6] Mariarosaria Cardone, Project Manager, and Technical Project Manager. OBC INITIALISATION SEQUENCE AND ATB. 2013.

[7] COSMIAC. Space Microsystems Innovations & Applications Center. URL `http://www.cosmiac.org/ground.html`.

[8] Gaillen Van Craen. DNX-TUD-TN-0883 [3. 2011.

[9] James Cutler. Ground Station Virtualization. .

[10] James Cutler. Mercury Ground Station Network project, . URL `http://sourceforge.net/projects/mgsn/files/`.

[11] James Cutler. Mercury Ground Station Network, . URL `http://mgsn.org`.

[12] J.W. Cutler. Ground station markup language. *2004 IEEE Aerospace Conference Proceedings (IEEE Cat. No.04TH8720)*, pages 3337–3343, 2004. doi: 10.1109/AERO.2004.1368140. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1368140`.

[13] John Dooley. *Software Development and Professional Practice*. Springer, 2011.

[14] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories [Hardcover]*. Addison-Wesley, 2003. ISBN 0321154207. URL `http://www.amazon.com/Handbook-Software-Systems-Engineering-Observations/dp/0321154207`.

[15] ESA. EGSE Toolkit, 2007. URL `http://telecom.esa.int/telecom/www/object/index.cfm?fobjectid=2857`

[16] Dan Galorath and Fred Brooks. Software Reuse and Commercial Off-the-Shelf Software.

[17] Florian George. Satellite Control Software (SCS) Mission Data Client Extensibility User Guide. (November):1–16, 2013.

[18] Florian George and Stephane Billeter. Satellite Control Software (SCS) EGSE Router Infrastructure ICD. (November), 2013.

[19] Steve Guest. Smooth Transition and the ILT Experience EGSE Systems in ILT / IST as a Whole, 2013. URL `http://herschel.esac.esa.int/twiki/pub/Public/LessonsLearned2013Agenda/egse.pdf`.

[20] Anil S. Jadhav and Rajendra M. Sonar. Evaluating and selecting software packages: A review. *Information and Software Technology*, 51(3):555–563, March 2009. ISSN 09505849. doi: 10.1016/j.infsof.2008.09.003. URL `http://linkinghub.elsevier.com/retrieve/pii/S0950584908001262`.

[21] Alexander Kossiakoff, William Sweet, Samuel Seymour, and Steven Biemer. *SYSTEMS ENGINEERING PRINCIPLES AND PRACTICE*. ISBN 9780470405482.

[22] M.S. Kuiper and J. Hernando Bravo. *DUDe Telemetry Client Software Design*. PhD thesis, 2013.

[23] Kyle Leveque, Jordi Puig-suari, and Clark Turner. Global Educational Network for Satellite Operations (GENSO). pages 1–6.

[24] Mercury Design Team. The Mercury Ground Station Reference Model, 2004. URL `http://mgsn.sourceforge.net/docs/model.php`.

[25] Misc. MySQL - open source database, 2014. URL `http://www.mysql.com`.

[26] Misc. Delfispace, 2014. URL `http://www.delfispace.nl/delffi`.

[27] Yuya Nakamura and Shinichi Nakasuka. LOW-COST AND RELIABLE GROUND STATION NETWORK TO IMPROVE OPERATION EFFICIENCY FOR MICRO / NANO-SATELLITES. *IAC*, pages 1–8, 2005.

[28] Yuya Nakamura and Shinichi Nakasuka. GROUND STATION NETWORK TO IMPROVE OPERATION EFFICIENCY. 2006.

[29] F Rademakers. ROOT library. URL `http://root.cern.ch/drupal/`.

[30] Remco Schoemaker. Robust and Flexible Command & Data handling on board the DelFFi Formation Flying mission, 2014.

[31] VKI Scholz, T, VKI March, G., and EPFL Richard, M. Ground Segment Definition. 2014.

[32] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974. ISSN 0018-8670. doi: 10.1147/sj.132.0115. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5388187`.

[33] The Spacelinkngt Tm, T C Interface, Processing Unit, T C Ipu, Ssbv Space, Ground Systems, and The Tm. TM / TC Interface & Processing Unit (TM / TC IPU).

[34] Y.Sakamoto. UNISEC Ground Station Management Service, 2007. URL `http://www.astro.mech.tohoku.ac.jp/ gsn/en/index.php`.

D

The International Astronautical Congress (IAC) Paper

68th International Astronautical Congress (IAC), Adelaide, Australia, 25-29 September 2017.

IAC-17-B1.6.6

Scalable Data Processing System for Satellite Data Mining

Stefano Speretta[a*], Anatoly Ilin[b]

[a] *Delft University of Technology (TU Delft), Kluyverweg 1, 2629 HS Delft, The Netherlands,* s.speretta@tudelft.nl
[b] *Delft University of Technology (TU Delft), Kluyverweg 1, 2629 HS Delft, The Netherlands,* a.ilin@student.tudelft.nl
[*] Corresponding Author

Abstract

This paper describes the development of a new ground station infrastructure targeted towards massive swarms and constellations. Mission trends are first analyzed to derive the design drivers for such a system and then the general architecture is analyzed. The target architecture, based on "Big Data" processing architectures is presented, clearly showing how to re-use current data processing state-of-the-art systems for satellite operations. This paper also describes the ongoing developments to integrate standard data mining and artificial intelligence software frameworks in the data acquisition system to develop a complete system capable of acquiring data from multiple sources, autonomously process them and deliver them to users.
Keywords: CubeSat, swarm, constellation, ground station, data mining

1. Introduction

Satellite swarms and constellations are becoming more and more widespread thanks to hardware and launch cost reduction. Nano-satellites are proving extremely suited for such big constellations where the single satellite has very limited capabilities but, when combined, very powerful systems could be created (capable for example of observing the whole Earth once a day). One of the problems arising from this trend is the constant increase in data to be transmitted to ground and the increased complexity in running a constellation with more than 100 satellites.

Several institutions have invested consistent effort in the development of more capable ground systems to acquire and process all the data. The geographical distribution of such infrastructure is becoming critical and so does the capability of aggregating data from multiple sources (the spacecrafts): this infrastructure is getting more and more similar to the one used by most web companies (like Facebook, Google, etc...) to process analytics coming from web pages. Both infrastructures need to swallow big amounts of data (or "Big Data") in quasi-real-time: in both cases further data analysis (or mining) could help identifying hidden trends (such as, for example, possible failures). All these points lead to the design of this system, trying to re-use part of the existing data processing infrastructure and applying it to satellite data analysis. By coupling together a data-gathering section (acquiring data from the different ground stations) and a data analysis section, we aim at developing a full ground system capable of supporting massive constellation, up to the point where human operators would have problems running it.

In the following sections we will analyze the current trends in satellite systems (see Section 2 to justify our attention towards massive constellations). We will then analyze the architecture of such system by starting from the experience gathered with nano-satellites and looking at extending it to big constellations (see Section 3). Proposed Lambda architecure will be presented (see Section 4), together with some future work (see Section 5) and conclusions (see Section 6).

2. Space Mission trends

The steep decrease in launch cost in the past decade lead to an increase in satellite launches per year, as also shown in Figure 1. In particular, nano-satellites saw a tremendous increase in launches per year (264 CubeSats were launched in the first 3 quarters of 2017, as compared to the total number satellites launched between 2015 and 2016)[2]. But unfortunately the performances of a single satellite are still limited when compared to big-

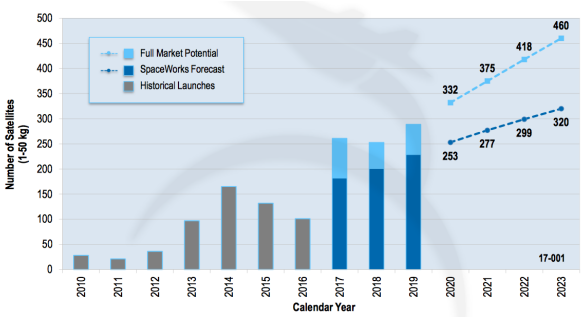

Fig. 1: Nano/Microsatellites launch trends[1].

ger missions and this lead, together with the dramatic launch cost reduction, to the diffusion of multi-satellite missions[3][4], that are also being proposed for several purposes, like climate science [5][6], atmospheric observations [7][8][9] or disaster monitoring[10]. At the same time, constellations and swarms are becoming as big as 197 satellites for Earth observation, such as the Flock constellation from Planet, whose goal is imaging the whole Earth at coarse resolution once a day[11].

The ground infrastructure for a massive swarm needs to handle a high number of satellite passes per day (approximately 350 with a fleet of 50 satellites[12], expected to grow to 1400 with the full constellation of approximately 200 satellites) from different locations worldwide. Multiple locations would be fundamental to achieve such goal (see Figure 2, for example) and a strong and fast data processing network will be fundamental too.

A lot of research going into massively distributed ground systems, as can be seen in [14][15][16][17]. But beside the pure data collection and archiving, handling massive amounts of data poses challenges in itself (system scalability, flexibility and fault tolerance) that are currently being addressed. Operations of such big swarms and constellations proves also critical, having the operators a huge number of satellites to monitor and control. All these reasons lead us to study the architecture of a massively distributed ground infrastructure, which will be presented in the next sections.

3. System architecture evolution

3.1 The legacy, no scalability

As a starting point of the discussion and to a familiarize the reader with the core framework functionality, consider the legacy implementation outlined in Figure 3. Telemetry processing system has been developed for an educational nano-satellite mission (Delfi-n3Xt): relying on the Delft ground station, as well a set of third party radio-amateurs submitting data through an ad-hoc client

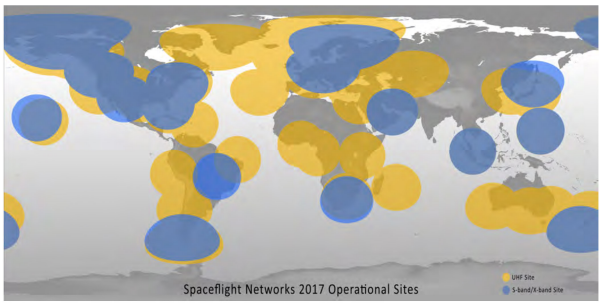

Fig. 2: Spaceflight distributed ground station network[13].

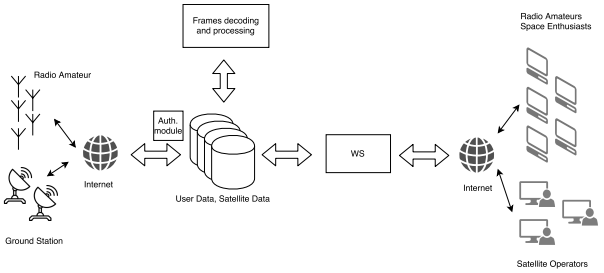

Fig. 3: Delfi Legacy architecture[21].

application [18]. The client application performed demodulation, decoding and limited data visualization [19]. Upon successful client authentication, received satellite data (data frames[20]) is injected directly to the SQL database on the Delfispace server. A set of processing scripts, controlled by a scheduler, decoded binary data frames to set of satellite metrics and store the latter in the database. Finally, the satellite functional parameters, filtered by client permissions, are made accessible to the stakeholders via a simple web server[21]. The legacy system in Figure 3 is a classic example of a "monolithic architecture" [22]. The limited scalability expresses itself in two ways: data processing and server scalability. Processing is limited to a single data frame definition and cannot be extended. Server scalability can achieved vertically, by allocating more resources, or horizontally, by running additional servers in parallel. In practice, horizontal scalability is preferred due to redundancy concerns. Arguably, aforementioned can be accomplished by deploying the legacy database in load-balanced configuration [23], with additional monitoring of processing scripts and load balanced web servers. This effort will grossly under perform compared to purpose built scalable systems [24][25].

3.2 Scale by leveraging clients

It should be noted, that in the case of Delfi-n3Xt mission, clients submitting data frames, simultaneously acts as clients retrieving processed data. A possible evolution of the previous system is shown by Figure 4, where the system relies entirely on a distributed database system for data transport and it is based on the unique server-side Couch DB and client-side Pouch DB ecosystem. PouchDB is a javascript implementation of CouchDB, a no-SQL, document database with out-of-the-box enabled sharing and data replication capability[26][27]. Being written in javascript, PouchDB runs in the web browser, serving database-stored web pages and performing data visualization, even while offline. By deploying this ecosystem, satellite data and web pages can be replicated to the clients, reducing the load on the central server. Any newly received satellite data by any ground station will

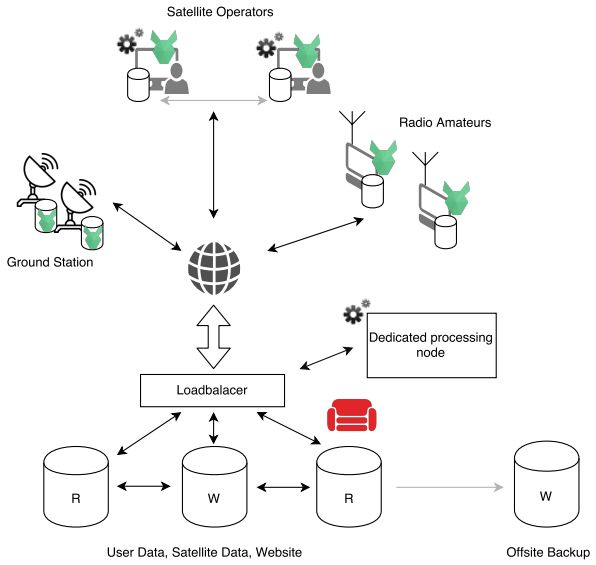

Fig. 4: Proposed client cluster architecture.

be replicated to the central server and replicated to all the clients.

To increase system stability, a load balancer is introduced and configured to perform write operations only to one node, and "read" on two dedicated nodes.

This correlates to the CAP theorem[28]: any data storage system can only ensure two out of three characteristics: Consistency, Availability or Partition Tolerance. Originally proposed by Marz[29], as a solution to reduce the system complexity by "the use of mutable state in databases and the use of incremental algorithms to update that state", provides a solution to the problem. This facilitates the partition tolerance (P) and availability (A). CouchDB[30], like many other no-SQL databases, prefers A and P over the Consistency (C), meaning that at a particular time after new data ingestion or data change, there will be nodes serving different versions of the data [28]. Hence the terminology "eventual consistency".

The choice of a single "write" and double "read" node is not arbitrary. Architecture is by design "read" centered, demanding a low latency for data replication to the clients. As a bonus, this contributes to the higher availability. The single "write" node eliminates issues with document duplication on the server side. To further streamline data replication between clients, especially replication to the new clients, peer-to-peer protocols have been studied[31].

Both CouchDB and PouchDB are document-based database systems. In this design, documents are equivalent to the row in SQL-like databases, but provide great flexibility by not enforcing any schema. This is required to facilitate flexible data frame design[20] and tolerate missing or corrupt data due to bit-flips. The satellite data frames database uses key-value pairs defined as JavaScript objects, e.g. JSON.

As stated earlier, PouchDB can serve complete HTML5 web pages and run JavaScript applications from its own database. Using this technique, clients can be configured to process the data, store in the client-specific PouchDB database, and replicate it back to the central server. The central server can perform a MapReduce operation[32] on all client-processed data, eliminating any inconsistencies, before adding to the main storage. MapReduce is a well-studied and understood parallelized data processing approach and literature provides numerous successful applications as well challenges faced by using this method[33][34]. The method is designed and therefore well suited for recurring queries and data preprocessing: batch-processing[35]. It should be stressed that setting up mappers and reducers for an on-demand one-time query is convoluted and performance is slower compared to the classical SQL querying (provided that the data fits into a single machine).

Any architecture relying heavily on client-side data generation requires an extensive security analysis that is beyond the scope of this paper. For the sake of argument, running a JavaScript based database and processing scripts makes reverse-engineering of the code and the authentication methods trivial. Additionally, the running system can be modified by a malicious user in runtime. Losing the edge on the data ingestion means that a single malicious machine would be able to 1) ingest a tremendous amount of data into the framework saturating the central database, 2) inject executables into the frame data, possibly compromising the central database or other client applications via a peer-to-peer connection.

Additional to the security threats, the architecture is a classic example of a vendor-lock-in: shifting to a different database system would require significant efforts reducing the flexibility of the future system development along with adding a set of risk factors[36][37].

3.3 Lessons learned

Merging the challenges faced with the production grade legacy system[38] with the experimental CouchDB-PouchDB partial implementation revealed a list of attention points to be addressed by the final system design and listed in the following sub-sections.

3.3.1 Security

It is evident that the system security should be considered in the earliest stages of the study. Considering a broader picture entailing data protection and recovery is especially important. As well the quantifying and categorizing users based on data access permissions and data

querying.

3.3.2 Client-side data replication and processing

Client-side data processing introduces many variables to the system and not necessarily pays off in the long run. Various client environments oppose different challenges, in case of PouchDB, the operating system and browser contain security features interfering with operations. As an example, the browser local storage is limited to 5Mb on iOS devices and on Safari (MacOS) requests users to validate the local storage permissions incrementally.

JavaScript has the capability to run in parallel, but executing scripts negatively affects the data replication speed. Additionally, after a browser restart, the earlier replicated data could be invalidated, requiring a complete re-download.

3.3.3 Data ingestion

The analysis of the historical data revealed that radio amateurs are sparsely scattered around the world, with a density peak in proximity to the Delft ground station, resulting in data duplication. A similar result could also happen due to the non optimal planning in a distributed ground station network or due to redundancy in receiving stations. Different network performances (or the temporary unavailability of network connection) could also result in out-of-order data frame ingestion on the database side. Data processing scripts may contain errors, requiring re-computation of the complete datasets.

3.4 Transition to Big Data: Lambda architecture

Looking at previously described systems exposes a common flaw. Telemetry frames received by the telemetry server, are processed and added to the central server, at which point the data distribution to the clients take place. Running both systems revealed that next to machine-tolerance, system should have human-tolerance, as bugs in data processing are frequent [38] and arguably unavoidable. Additionally, the system faces a more general challenge: on the one hand, near-real time data processing is required, while on the other hand, datasets are expected to be consistent and reliable. This correlates back to the CAP theorem [28]: any data storage system can only ensure two out of three characteristics: Consistency, Availability or Partition Tolerance. Originally proposed by Marz [29], as a solution to reduce the system complexity by "the use of mutable state in databases and the use of incremental algorithms to update that state", provides a solution to the problem. A common implementation is an architecture consisting split into two parts, one for incremental state update: speed layer and one containing immutable data used for analysis: batch layer [39]. The system architecture following this approach is commonly known as

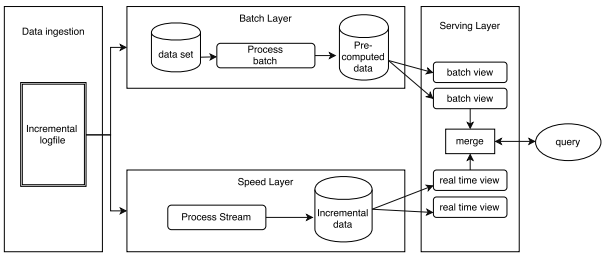

Fig. 5: Lambda Architecture

Lambda architecture. Figure 5 provides a high level functional overview. It should be noted that each node on the diagram represents a server cluster.

3.4.1 Data consumption

As shown in figure 5 data consumption is the data entry point to the system. Architecture does not oppose requirements on the design of this component, however, all incoming data has to be split into two identical streams, one consumed by batch layer, one by speed layer.

3.4.2 Batch layer

The batch layer has two functions, appending new data to the immutable data storage, and computing the batch view. It should be stressed that all received data is stored permanently, preferably without the ability to be modified, preventing data corruption due to human interaction. Once stored, data ought to be processed in batches, eliminating the data inconsistencies such as duplicates and out-of-order frames.

3.4.3 Streaming layer

Running large batch jobs is both time and resources consuming, and is therefore expected to be executed on intervals. The streaming layer, designed to compensate for the data between the batch intervals, depends on the real-time data arriving the system and is therefore completely independent of the immutable storage. An interesting consequence of this is that the stream processing generates own stream views, potentially containing out-of-order and duplicate data.

3.4.4 Serving layer

Serving layer is responsible for running queries on the collection of the streaming and batch views. As previously stated, all historic data is present in the batch view, while all the newly received data can be found in the stream views. The techniques of executing the queries and removing redundant stream view data upon batch completion are not enforced by the architecture and is part of the implementation.

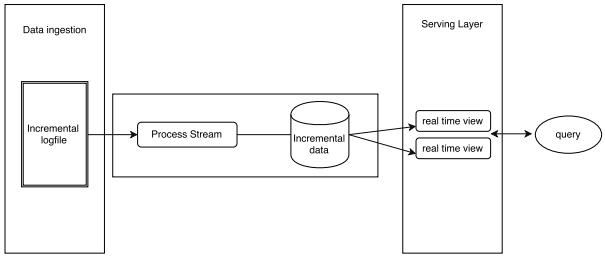

Fig. 6: Kappa Architecture

3.5 Transition to Big Data: Kappa architecture

When introduced, Lambda architecture relied heavily on MapReduce as batch processor and Apache Storm and Apache Flink for stream processing layer. With the maturity of Spark Streaming API, a number of alternatives architectures has been proposed, Kappa architecture as the most popular one. As shown in Figure 6, Kappa is a simplification of Lambda architecture, with the batch processing and stream processing aggregated to a single Spark cluster. Spark batch processing, sharing the code base with Spark Streaming, can be executed on demand to reprocess the complete historic data set. While for the day-to-day operations system would utilize the Spark streaming. This approach simplifies the serving layer, removing the double views.
When working with stream processing, it is important to keep transactions stateless whenever possible. Relying on database reads to validate the data, to remove duplicates or out-of-order frames, is not welcomed. In some cases batch processing will be required to remove or update past event states, what in its turn invalidates earlier streaming views, requiring extra complexity to mitigate the downtime.

4. Selected architecture

The data processing system is designed with two main functions in mind: provide satellite monitoring functionality and facilitate data mining. The satellite monitoring entails telemetry data processing and visualization. Data mining entails identifying data sources, building and validating data models iteratively, and potentially embedding data model outcome in the monitoring dashboard. From the earlier proposed architectures, only Lambda and Kappa can be considered as viable options. For data transformation and visualization purposes, Kappa architecture proves as well suited candidate, due to its streamlined and lean approach. However, by oposing data mining requirements, frequent batch jobs become a necessity for building, validating and improving of the data models. Additionally,by regularly recalculating the complete historic data set, ensures data consistency in the serving layer, making data more accessible for querying by satellite operators. The abstraction of the Batch layer opens opportunities to run jobs on separate (cloud compute) clusters, cutting processing time for resource demanding computations without affecting satellite operations. This leads to the selection of Lambda architecture for the project.

Following sections cover high-level decisions and framework selection. The groundwork of available applications is well covered in the literature [40] [41] [42],therefore only high-level description will be provided.

4.1 Serving layer

The serving layer is designed to aggregate and serve data from streaming and batch layers to the client application. This function can be fulfilled in many ways, for example by running a single database system or a query engine on two different database systems. The implementation depends on the user requirements, in scope of this project, users require near-real-time graphical and tabular views of the satellite status (dashboard). Additionally, the user should be able to execute custom, on-demand queries for data analysis and satellite troubleshooting.

Since the satellite telemetry is techincally time-series data, the majority of the existing log data visualization frameworks can be applied out of the box. Kibana and Grafana are two most popular and powerful open-source visualization tools [43]. Grafana is designed with a time-series database on the backend in mind, while Kibana utilizes ElasticSearch. Grafana supports multiple databases following strict time-series schema, while Kibana, only supports ElasticSearch but allowing more flexible schema. Recently published work [44] proves both frameworks comparable on the visualization aspects but requiring further research on graphing capabilities for the actual satellite telemetry. ES enables users to execute queries and calculations within Kibana. While querying time-series database, such as InfluxDB, requires an additional interface to bind to the database API. Providing this functionality to users over the internet increasing overall complexity and requires careful design and implementation.

For the project Kibana and ElasticSearch has been selected, due to its operational simplicity, features and ease of implementation and maintenance.

Utilizing Kibana requires both Streaming as Batch processed data to be stored in ElasticSearch. Aggregating this data requires removal of redundant Streaming Data upon Batch completion. This is not a unique problem [45] and can be resolved using ElasticSearch Curator by assigning retention to the streaming data. Another solution to the problem is to overwrite all data in ElasticSearch on Batch completion, actively removing all Streaming Data.

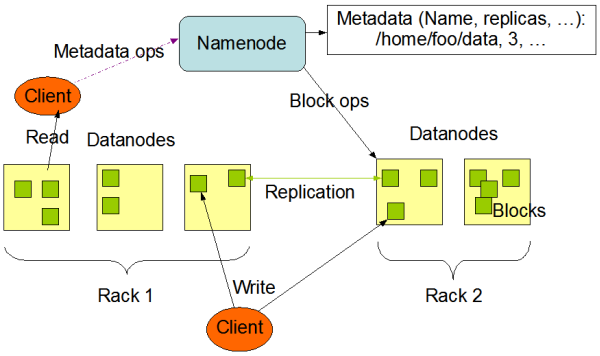

Fig. 7: HDFS Architecture [46]

Further research is required to determine the optimal technique.

4.2 Batch layer

The batch layer is introduced to the Lambda architecture with the purpose of bulk data processing and retention of the immutable data set.

4.2.1 Immutable data

The storage of the immutable data set requires a system that ensures file integrity while allowing access to multiple (simultaneous) readers. This can be achieved using (distributed) file system or a database system. Hadoop Distributed File System has been selected over databases systems due to

- Tolerance to unstructured data: satellite data can use different formats, from binary blobs to image and video file(streams)
- Scalability: deployed and extended to multiple machines without configuration changes to the client applications
- Analytics: HDFS, as part of Hadoop ecosystem, is universally supported for data processing, includes API's and query engines for custom data processing systems.

Hadoop Distributed File System (HDFS) is a distributed file system designed to provide scalable, fault-tolerant and consistent data storage across large clusters. Ability to store files greater than server capacity, while providing parallel access on multiple machines in a cluster, makes HDFS an attractive choice for Big Data applications.

As shown in the figure 7, HDFS cluster consists of two components: a Name Node (NN) and a set of Data Nodes (DN). HDFS exposes a file system, allowing clients to store files that are automatically broken up into blocks with the predefined size (128Mb by default) and stored redundantly on the Data Nodes. The Name Node acts as a controller, splitting data into blocks and managing blocks location while optimising read and write performance. CRUD as well file open and close operations are handled by Name Node. In this master-slave architecture, only a single Name Node is allowed at all times, making it a weak point of the system. To mediate this issue a secondary Name Node is assigned by Yarn containing a copy of the edit log, reducing the recovery time of the system.

4.2.2 Batch processing: MapReduce, Apache PIG and Apache TEZ

MapReduce is briefly covered in section 3.2 as part of CouchDB stack. In scope of BigData MapReduce runs natively within Hadoop stack, on top of the HDFS. The general concept is similar, and processing depends on Mappers to transform and Reducers to filter the data. Typically, MR batch job is controlled by Yarn (resources) and systems like Oozie for execution (time allocation). MR follows a master-slave approach , inherited from HDFS, with a single Node Manager running MR Application Master controlling, determining and allocating Map and Reduce tasks over the cluster. It performs an optimization of the job based on resources (CPU, RAM) as well the nodes locally available data to minimize the network bottlenecks. The Application Master is monitored by Yarn and in case of failure, will be relaunched automatically on a different node along with required information to resume the job. The Reduce jobs often, if not always, require aggregation of data from multiple Mappers, likely executed on different nodes, all handled by MR without being programmed in the query. The main drawback of MR is the two-stage process limitation, that can be medicated by chaining multiple MR operations, but decreasing the overall efficiency. Furthermore, the intermediate Mapper results are stored on the nodes hard drive, further degrading the performance [47]. Apache PIG is an infrastructure and a high-level language, PIG Latin, for data analysis programs, evaluating directly on MapReduce and Tez. Apache TEZ is a high-performance MapReduce alternative that relies on complex directed-acyclic-graphs (DAG) and Hadoop Yarn [48].

MapReduce, being part of the Hadoop ecosystem, is added automatically to the data analytics toolset by selecting HDFS as persistent storage. Due to limited performance of MR and processing limitation of PIG, both systems will not be utilized for batch processing use. However, PIG in conjunction with TEZ serves purposes for

data analytics and troubleshooting of the system.

4.2.3 Data analytics: Apache Hive and HBase

While Apache PIG is designed with scripting in mind, other abstractions have been developed to emulate a SQL database. Apache Hive is an analytics querying framework within Hadoop ecosystem. By design, Hive is optimized for analytics: online analytical processing (OLAP). In short, Hive provides a SQL-like interface (HiveQL) to access the data stored in HDFS file while only enforcing a schema on read. It should be noted that Hive does not provide record-level updates, inserts or deletes.
Apache HBase, a No-SQL alternative, is designed for online transaction processing (OLTP), similar to Google Big Table. Data records, stored in HDFS, are parsed to column and column families to mitigate missing data.
Nor Hive or HBase is required for batch processing. Hive is part of the Hadoop ecosystem, and will be available for the data analytics.

4.2.4 Apache Spark

Apache Spark is a popular framework used for big data analytics. Spark is deployed as a cluster application and can be monitored by YARN. In contrast to the two stage MapReduce, Spark executes multi-stage jobs in-memory, drastically improving the overall system performance. The core of Spark relies on the resilient distributed datasets (RDD) [49], abstraction for the partitioned collection of records. This ensures fault-tolerance and an ability to recompute damaged partition with data distributed over the cluster. The fault-tolerance is achieved by keeping all RDD's read-only, ensuring that every transformation creates a new RDD, making each RDD traceable and re-computable. The key to performance is DAG and the policy of transforming the RDDs only when directly dependent downstream RDDs are requested: lazy transformation.
The Spark stack consists of Spark Core, Spark Streaming, Spark SQL, MLlib and GraphX. Spark Core exposes high-level RDD and dataset API for batch data manipulation. Spark API supports a number of programming languages, Scala and Python being the most popular. Spark SQL exposes a SQL-like language for interaction with RDD, utilizing the structured data API. Spark MLlib is a module for machine learning utilizing the RDD abstractions. MLlib provides classification functionality, for example, K-Means clustering, providing the necessary frameworks for basic anomaly detection.
Spark is one of the most versatile batch processing tool available. This is required since the satellite data frame format [20] requires additional processing or decoding tools such as AVRO. Additionally, Spark Streaming allows the majority of code (structured data API) to be reused for both stream and batch processing.

4.3 Speed Layer

The speed layer requires fast processing while ensuring fault-tolerance and reliability to deliver data timely to the connected clients. At the time of writing, three distributed stream processing framework dominate the scene: Apache Storm, Apache Spark and Apache Flink. The frameworks are well studied and number of publications are made on the trade-off and benchmarks. [50] [51] For the purpose of this project Apache Spark has been selected. The core of data frame processing is identical for Speed and Batch layers; utilizing the same data processing framework, allows reuse of the code as well cluster, reducing the overhead.

4.4 Ingestion layer

Ingestion layer ought to provide a secured interface for the client applications to communicate with, undependable from server implementation and frameworks used. The API design is out of the paper scope and will be ignored for the discussion. The ingestion layer should be horizontally scalable and provide (temporary) data storage in case of immutable storage malfunctions (resilience).

4.4.1 API + HDFS

Hadoop Distributed File System (HDFS) exposes a programming interface that can be easily connected to the API used for client communication (data ingest), enabling direct data consumption by the cluster. This solution, however, requires high availability HDFS deployment to cover for any malfunctions, and a system to feed the streaming data to the Speed Layer.

4.4.2 Kafka

Defacto framework used for the ingestion layer in lambda architecture is Apache Kafka. Designed as a system to deliver high volume event data to subscribers, Kafka utilizes a write-ahead commit log on persistent storage and provides a pull-based messaging abstraction to allow both real-time subscribers such as online services and offline subscribers such as Hadoop and data warehouse to read these messages at arbitrary pace. [52].

The stream of records, published by API, are categorised in topics. Topics are used to define data pipelines and are consumed by subscribers: Speed layer and Batch layers. In case of clustered deployment, topics are build up from partitions, collectively called log. As shown in the Figure 8, each partition is an immutable sequence of received data. Offset, the unique id of each record is used to keep track of the last retrieved record per subscriber.

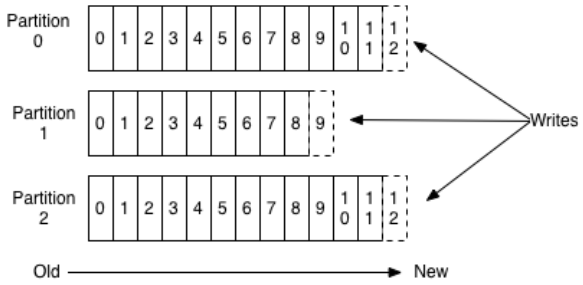

Fig. 8: Kafka Log Anatomy [53]

This enables topic subscribers to consume data at different rates. To keep track of offset and subscribers Kafka utilizes Apache Zookeeper. Additional to the messaging broker, Kafka contains Connect framework, an extensive set of ready-to-use Sink and Source connectors for integrating with the majority of existing databases and data providers. The framework is optimal for data migration from legacy system and load testing of the complete system implementation.
The choice for Kafka is made due to the following considerations:

- Horizontally scalable
- Ability to serve multiple data consumers at different rates
- Resilient data log, redundancy for temporarily HDFS system failures.
- Data delivery guarantee

4.5 Final Architecture

Selection process briefly outlined by previous sections, leads to the architecture shown by the Figure 9.
Satellite telemetry data submit through client application and API, is appended to commit log of Apache Kafka. Log serves as a temporarily data storage, until it is consumed by Spark Streaming and inserted to HDFS file system by Apache Connect HDFS Sink. Spark batch processing is executed on regular intervals, result of which overwrites all entries in the ElasticSearch system. Apache Spark Streaming is executed in micro batches with sub minute intervals. Processed data is appended to ElasticSearch with retention period. Kibana is configured to consume ElasticSearch data.

The architecture ensures interoperability with different components, for example an additional No-SQL database for specific customer needs, withouh major code overhaul. The batch layer, with aid of Spark can be used for a wide range of tasks, from model fitting (K-means) to full fledged machine learning.

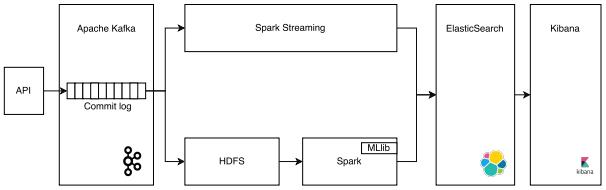

Fig. 9: Proposed Architecture including frameworks

5. Future Work

One of the main long-term goals of the work presented here (and still under development) is the use of satellite data (including payload data as well as on-board telemetry) for data mining and autonomous operations. Data mining is defined as the analysis of large amounts of data to extract further information from it: a clear example could be the analysis of performances indicators to predict system maintenance[54]. Spacecrafts could, for example, benefit from a predictive model calibrated around selected telemetry parameters, to predict eventual faults and implement corrective strategies in a completely autonomous way. This latter approach can be very fascinating, especially for deep-space probes that experience long communication delays and requires complex autonomous operations, but it would have to be ported to the satellite hardware.

Using historic telemetry data coming from space probes has also been exploited recently by the Mars Power Challenge[55], where the best modeling techniques were compared to predict 1 year worth of telemetry on the Mars Express probe based on 3 years worth of data. Possible artificial intelligence applications in this case would have to be integrated either on the satellite hardware or in the ground station infrastructure, leading to further implementation work. In our case, a future implementation of data mining algorithms will be simple to add because we already relied on standard applications used in the artificial intelligence / data mining field to realize the database and the data distribution system since this could be implemented by the processing layer already present. This approach will allow us to perform further research even during normal mission operations with the clear goal of creating an automated system to handle common anomalies.

6. Conclusions

In this paper we looked at the current trends in space missions, especially looking at nano-satellites, and fo-

cused on swarms and constellations. From these missions, we looked at the required ground segment to fulfill the mission requirements of handling up to thousands of passes per day. This requires the development of a distributed ground station system capable of scaling in a simple way. We presented an architecture to achieve such goals based on industry standard applications in the domain of data analytics and mining. We also presented some preliminary results on the implementation of such a system to clearly show the advantages of the selected architecture.

We also highlighted possible future developments making use of the described infrastructure to perform data mining and possibly autonomous operations by adding a data mining / artificial intelligence application to the existing distributed database. This new concept could provide huge benefits to big constellations by heavily reducing the operators work.

References

[1] "2017 Spaceworks Nano/Microsatellite Market Forecast." http://spaceworksforecast.com. (accessed 01.09.2017).

[2] "Gunter's space page." http://space.skyrocket.de. (accessed 02.09.2017).

[3] N. Crisp, K. Smith, and P. Hollingsworth, "Launch and deployment of distributed small satellite systems," *Acta Astronautica*, vol. 114, pp. 65 – 78, 2015.

[4] R. Sandau, *Implications of new trends in small satellite development*, pp. 296–312. Vienna: Springer Vienna, 2011.

[5] J. Esper, P. V. Panetta, M. Ryschkewitsch, W. Wiscombe, and S. Neeck, "Nasa-gsfc nano-satellite technology for earth science missions," *Acta Astronautica*, vol. 46, no. 2, pp. 287 – 296, 2000. 2nd IAA International Symposium on Small Satellites for Earth Observation.

[6] L. Dyrud, S. Slagowski, J. Fentzke, W. Wiscombe, B. Gunter, K. Cahoy, G. Bust, A. Rogers, B. Erlandson, L. Paxton, and S. Arnold, "Small-sat science constellations: why and how," in *Proceedings of the 27th Annual AIAA/USU Conference on Small Satellites*, (Lugan, UT), American Institute of Aeronautics and Astronautics (AIAA), 8 2013.

[7] D. J. Barnhart, T. Vladimirova, A. M. Baker, and M. N. Sweeting, "A low-cost femtosatellite to enable distributed space missions," *Acta Astronautica*, vol. 64, no. 11, pp. 1123 – 1143, 2009.

[8] R. Sandau, K. Brie, and M. DErrico, "Small satellites for global coverage: Potential and limits," *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 65, no. 6, pp. 492 – 504, 2010. ISPRS Centenary Celebration Issue.

[9] W. Saylor, K. Smaagard, N. Nordby, and D. Barnhart, "New scientific capabilities enabled by autonomous constellations of smallsats," in *Proceedings of the 21th Annual AIAA/USU Conference on Small Satellites*, (Lugan, UT), American Institute of Aeronautics and Astronautics (AIAA), 8 2007.

[10] D. J. Barnhart, T. Vladimirova, and M. N. Sweeting, "Very-small-satellite design for distributed space missions," *Journal of Spacecraft and Rockets*, vol. 44, no. 6, pp. 1294 – 1306, 2007.

[11] "Planet web page." https://www.planet.com. (accessed 04.09.2017).

[12] K. Colton and B. Klofas, "Supporting the flock: Building a ground station network for authonomy and reliability," in *Proceedings of the 30th Annual AIAA/USU Conference on Small Satellites*, (Lugan, UT), American Institute of Aeronautics and Astronautics (AIAA), 8 2016.

[13] "Spaceflight web page." http://spaceflight.com. (accessed 04.09.2017).

[14] B. Klofas, "Planet labs ground station network." 13th Annual CubeSat Developers Workshop, 4 2016.

[15] C. Venturini and T. McVittie, "Current and future ground systems for cubesats working group," in *Ground Systems Architecture Workshop*, The Aerospace Corporation, 3 2014.

[16] E. F. Moreira, A. Ceballos, C. Estvez, , J. C. Gil, S. Kang, J. Guiney, , and V. Ivatury, "Architecting oneweb's massive satellite constellation ground system," in *Ground Systems Architecture Workshop*, The Aerospace Corporation, 3 2017.

[17] K. Casey, W. Al-Masyabi, and M. Nagengast, "A visit to 2037," in *Ground Systems Architecture Workshop*, The Aerospace Corporation, 3 2017.

[18] "Delfi Space: TU Delft Small Satellite Program." http://www.delfispace.nl/operations/radio-amateurs. (accessed 02.09.2017).

[19] M. Kuiper, "DUDe Telemetry Client Software Design ," tech. rep., Delft University of Technology, 2013.

[20] R. Schoemaker, "Robust and flexible command & data handling on board the delffi formation flying mission," Master's thesis, Delft University of Technology, 2014.

[21] G. van Craen, "Design of the telemetry server," Master's thesis, Delft University of Technology, 2011.

[22] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," 2014.

[23] D. Haney and K. S. Madsen, "Load-balancing for mysql," *Kobenhavns Universitet*, 2003.

[24] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference (10CCC), 2015 10th*, pp. 583–590, IEEE, 2015.

[25] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, *et al.*, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*, pp. 179–182, IEEE, 2016.

[26] J. Justin and J. Jude, "Go offline," in *Learn Ionic 2*, pp. 79–97, Springer, 2017.

[27] "Pouchdb: The database that syncs." `https://pouchdb.com`. (accessed 02.09.2017).

[28] S. Gilbert and N. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[29] N. Marz, "How to beat the cap theorem," *nathanmarz. com*, 2011.

[30] "Couchdb: the definitive guide." `http://guide.couchdb.org/draft/consistency.html`. (accessed 02.09.2017).

[31] R. Leeds, "Chrome to chrome pouchdb." CouchDB Conf Berlin, 2013.

[32] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[33] S. N. Khezr and N. J. Navimipour, "Mapreduce and its applications, challenges, and architecture: a comprehensive review and directions for future research," *Journal of Grid Computing*, pp. 1–27, 2017.

[34] S. A. Thanekar, K. Subrahmanyam, and A. Bagwan, "A study on mapreduce: Challenges and trends," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 4, no. 1, pp. 176–183, 2016.

[35] R. Singh and P. J. Kaur, "Analyzing performance of apache tez and mapreduce with hadoop multinode cluster on amazon cloud," *Journal of Big Data*, vol. 3, no. 1, p. 19, 2016.

[36] D. S. Kusumo, M. Staples, L. Zhu, H. Zhang, and R. Jeffery, "Risks of off-the-shelf-based software acquisition and development: A systematic mapping study and a survey," 2012.

[37] A. Shvets, *Design Patterns Explained Simply*. Source Making, 2017.

[38] S. van Kuijk, "Delfi-n3xt forensics: A hybrid methodology," Master's thesis, Delft University of Technology, 2016.

[39] N. Marz and J. Warren, "Big data: principles and best practices of scalable real-time data systems," 2013.

[40] V. Chavan and R. N. Phursule, "Survey paper on big data," *Int. J. Comput. Sci. Inf. Technol*, vol. 5, no. 6, pp. 7932–7939, 2014.

[41] D. Singh and C. K. Reddy, "A survey on platforms for big data analytics," *Journal of Big Data*, vol. 2, no. 1, p. 8, 2015.

[42] V. B. Bobade, "Survey paper on big data and hadoop," *Int. Res. J. Eng. Technol*, vol. 3, no. 1, pp. 861–863, 2016.

[43] A. Yigal, "Grafana vs. kibana: The key differences to know." `https://logz.io/blog/grafana-vs-kibana`. Accessed: 2017-09-02.

[44] I. Nurgaliev, E. Karavakis, and A. Aimar, "Kibana, grafana and zeppelin on monitoring data," Aug. 2016.

[45] P. Kleindienst, "Building a real-world logging infrastructure with logstash, elasticsearch and kibana,"

[46] "Hdfs architecture guide." `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[47] P. Kannan, "Beyond hadoop mapreduce apache tez and apache spark," *San Jose State University. URL: http://www. sjsu. edu/people/robert. chun/courses/CS259Fall2013/s3/F. pdf*.

[48] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, pp. 1357–1369, ACM, 2015.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, USENIX Association, 2012.

[50] A. Shukla and Y. Simmhan, "Benchmarking distributed stream processing platforms for iot applications," in *Technology Conference on Performance Evaluation and Benchmarking*, pp. 90–106, Springer, 2016.

[51] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pp. 1789–1792, IEEE, 2016.

[52] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1654–1655, 2015.

[53] "Kafka documentation." `https://kafka.apache.org/documentation/`.

[54] P. Bastos, I. Lopes, and L. Pires, "A maintenance prediction system using data mining techniques," in *World Congress on Engineering 2012*, vol. 3, pp. 1448–1453, International Association of Engineers, 2012.

[55] "Mars express power challenge." `https://kelvins.esa.int/mars-express-power-challenge`. (accessed 01.09.2017).

Bibliography

[1] J. Puig-Suari, C. Turner, and W. Ahlgren, *Development of the standard CubeSat deployer and a CubeSat class PicoSatellite,* in *Aerospace Conference, 2001, IEEE Proceedings.*, Vol. 1 (IEEE, 2001) pp. 1–347.

[2] J. Puig-Suari and B. Twiggs, *CUBESAT Design Specifications Document*, Tech. Rep. (CalPoly, San Luis Obispo, 2001).

[3] E. Buchen, *2014 Nano / Microsatellite Market Assessment,* (2014).

[4] J. Straub, *CubeSats: A Low-Cost, Very High-Return Space Technology,* AIAA Reinventing Space Conference (2012).

[5] V. Scholz, T, V. March, G., and E. Richard, M., *Ground Segment Definition*, Tech. Rep. (Von Karman Institute for Fluid Dynamics, 2014).

[6] F. George, *Satellite Control Software (SCS) Mission Data Client Extensibility User Guide,* (2013).

[7] G. V. Craen, *Design of Telemetry server*, Master's thesis, Delft University of Technology (2011).

[8] M. Sweeting, *Modern Small Satellites - Changing the Economics of Space,* Proceedings of IEEE (2018).

[9] S. V. Kuijk, *Delfi-n3Xt Forensics*, Master's thesis, Delft University of Technology (2016).

[10] E. Buchen, *Nano / Microsatellite: market forecast 2018*, Tech. Rep. (SpaceWorks, 2018).

[11] S. Speretta and A. Ilin, *Scalable Data Processing System for Satellite Data Mining,* (2017).

[12] B. Boehm, *Spiral Model of Software Development and Enhancement,* Computer (1987).

[13] A. Ilin, *Literature Study,* (2014).

[14] S. P. Berczuk, *Organizational Multiplexing : Patterns for Processing Satellite Telemetry with Distributed Teams,* Pattern Languages of Program Design , 1 (1996).

[15] Delft University of Technology, *Delfi Space Programme,* (2018).

[16] Nuand, *BladeRF: USB 3.0 Software Defined Radio,* (2016).

[17] M. Ossmann, *HackRF One,* (2018).

[18] D. Hartanto, *Reliable Ground Segment Data Handling System for Delfi-n3Xt Satellite Mission*, Master's thesis, Delft University of Technology (2009).

[19] M. d. Miliano, L. Boersma, and A. Tindemans, *Delfi-n3Xt: Top-Level Design of Communication Subsystem*, Tech. Rep. (Delft University of Technology, 2012).

[20] J. McGuire, *FX-25 Performance,* (2007).

[21] H. Zimmermann, *OSI Reference Model-The ISO Model of Architecture for Open Systems Interconnection,* IEEE Transactions on Communications **28**, 425 (1980).

[22] T. L. Fox, *AX.25 Amateur Packet-Radio Link-Layer Protocol,* (1984).

[23] A. M. Grønstad, *Implementation of a communication protocol for CubeSTAR*, Master's thesis, University of Oslo (2010).

[24] M. Kuiper and J. Hernando Bravo, *DUDe Telemetry Client Software Design*, Tech. Rep. (Delft University of Technology, 2013).

[25] R. Schoemaker, *Robust and Flexible Command & Data handling on board the DelFFi Formation Flying mission*, Master's thesis, Delft University of Technology (2014).

[26] E. Klitzke, *Why Uber Engineering Switched from Postgres to MySQL | Uber Engineering Blog,* (2016).

[27] J.-H. Le Roux, *Development of a satellite network simulator tool and simulation of AX.25, FX.25 and a hybrid protocol for nano-satellite communications*, Master's thesis, Stellenbosch University (2014).

[28] H. By and F. Of, *4th INTERNATIONAL CONFERENCE ON DESIGN, DEVELOMENT & RESEARCH Proceedings,* (2014) pp. 183–202.

[29] U. Wendel, *MySQL Document Store: unstructured data, unstructured search - Ulf WendelUlf Wendel,* (2016).

[30] Oracle Corporation, *MySQL 8.0 Reference Manual: 13.2.6 INSERT Syntax,* (2018).

[31] J. Hayden, *MySQL Performance: Stop hoarding. Drop unused MySQL databases,* (2017).

[32] L. Dr. Rosenberg, T. Hammer, and J. Shaw, *Software metrics and reliability,* Proceedings of the 9th International Symposium on Software Reliability Engineering , 1 (1998).

[33] X. Wu, X. Zhu, and G. W. W, *Data Mining with Big Data,* IEEE Transactions on Knowledge and Data Engineering **26**, 97 (2014).

[34] M. Gao, G. Ayers, and C. Kozyrakis, *Practical Near-Data Processing for In-Memory Analytics Frameworks,* Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT **2016-March**, 113 (2016).

[35] T. Haerder and A. Reuter, *Principles of transaction-oriented database recovery,* ACM Computing Surveys **15**, 287 (1983).

[36] K. Eswaran, N. J. Gray, R. Lorie, and I. Traiger, *The Notions of Consistency and Predicate Locks in a Database System.* Comm. of the ACM **19** (1976).

[37] E. Brewer, *Spanner, TrueTime & The CAP Theorem,* White Papers **2015**, 1 (2017).

[38] E. Brewer, *Towards Robust Distributed Systems,* Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing , 7 (2000).

[39] S. Gilbert and N. Lynch, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services,* ACM SIGACT News **33**, 51 (2002).

[40] G. Maxia, *The Data Charmer: How to break MySQL InnoDB cluster,* (2017).

[41] C. Perera, R. Ranjan, L. Wang, S. U. Khan, and A. Y. Zomaya, *Big data privacy in the internet of things era,* IT Professional **17**, 32 (2015).

[42] M. Mether, *Scaling MySQL and MariaDB | SCALE 14x,* (2016).

[43] Codership Ltd., *Node Failure and Recovery — Galera Cluster Documentation,* (2014).

[44] A. van Scheppingen, *Top mistakes to avoid in MySQL replication | Severalnines,* (2017).

[45] Sandvine, *2016 - Global Internet Phenomena - Latin America \& North America*, Tech. Rep. (Sandvine, 2016).

[46] C. Strozzi, *DB-Engines Ranking - popularity ranking of database management systems,* (2018).

[47] N. Leavitt, *Will NoSQL Databases Live Up to Their Promise?* Computer **43**, 12 (2010).

[48] Apache Software Foundation, *Apache Avro™ 1.7.7 Specification,* (2014).

[49] E. Brewer, *CAP twelve years later: How the "rules" have changed,* Computer **45**, 23 (2012).

[50] A. Siddiqa, A. Karim, and A. Gani, *Big data storage technologies: a survey,* Front Inform Technol Electron Eng **18**, 1040 (2017).

[51] F. Gessert, W. Wingerath, S. Friedrich, and N. Ritter, *NoSQL database systems: a survey and decision guidance,* Computer Science - Research and Development **32**, 353 (2017).

[52] J. Gray and P. Helland, *The Dangers of Replication and a Solution,* SIGMOD **6**, 173 (1996).

[53] MongoDB, *Ranged Sharding — MongoDB Manual 3.6,* (2018).

[54] I. DataStax, *Consistent hashing | Apache Cassandra 2.1,* (2018).

[55] V. Srinivasan, B. Bulkowski, and R. Iyer, *Aerospike : Architecture of a Real-Time Operational DBMS,* Pvldb **9**, 1389 (2016).

[56] K. P. Phyu and W. Z. Shun, *Data lake : a new ideology in big data era,* ITM Web of Conferences 17 **03025**, 1 (2018).

[57] R. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design* (Prentice Hall, 2017).

[58] P. Biondi, *Scapy 2.4.0documentation,* (2018).

[59] A. Bulski, T. Filiba, and C. Simpson, *Construct documentation,* (2018).

[60] V. Stinner, *Hachoir documentation,* (2014).

[61] A. Herrera, A. Bulski, J. Plum, C. Leimbrock, M. Yakshin, and T. Koczka, *Kaitai Struct: declarative binary format parsing language,* (2018).

[62] Google, *Protocol Buffers,* Google Developers , 1 (2016).

[63] Sadayuki Furuhashi, *MessagePack: It's like JSON. but fast and small.* (2013).

[64] O. Ben-kiki, C. Evans, and B. Ingerson, *YAML Ain ' t Markup Language (YAML ™) Working Draft 2004-12-28,* Language (2004).

[65] D. J. White, I. Giannelos, A. Zissimatos, E. Kosmas, and D. Papadeas, *SatNOGS: Satellite Networked Open Ground Station,* in *Engineering Faculty Publications* (2015).

[66] E. Dijkstra, *On the role of scientific thought (EWD447),* (1974).

[67] C. Reade, *Elements of Functional Programming* (Addison-Wesley, 1989).

[68] Edgescan, *Edgescan*, Tech. Rep. (Edgescan, 2016).

[69] Edgescan, *Vulnerability Statistics Report*, Tech. Rep. (Edgescan, 2018).

[70] A. Cockburn, *Hexagonal Architecture,* (2005).

[71] J. Palermo, *The Onion Architecture,* (2008).

[72] R. Martin, *Clean Architecture,* (2012).

[73] C. Schults, *An Introduction To Clean Architecture - NDepend,* (2017).

[74] W. Ul Haq, *A Brief Summary of thoughts on Clean Architecture and MVP,* (2018).

[75] M. Nagy, *Thoughts on Clean Architecture – AndroidPub,* (2017).

[76] R. Martin, *The Clean Architecture,* (2012).

[77] C. Gorman, *Future Proof,* (2016).

[78] M. Thurau, *Akka framework,* University of Lubeck (2012).

[79] S. McConnell, *Code Complete: A Practical Handbook of Software Construction, Second Edition* (Pearson Education, 2004) p. 652.

[80] W3C, *Web Audio API,* (2015).

[81] B. Smus, *Web Audio API,* (2013).

[82] Fyrd, *Webaudio API support,* (2018).

[83] P.-T. de Boer, *WebSDR: Software-Defined Radio receiver connected to the internet,* (2018).

[84] J. Dolske, *A JavaScript 1200 baud audio modem,* (2017).

[85] B. Armstrong, *QuietJS,* (2018).

[86] M. Melhus, *Web Audio Modem,* (2017).

[87] W3C, *HTML 5: Draft 2009,* (2009).

[88] World Wide Web Consortium, *Web Storage (Second Edition),* (2009).

[89] J. Resig, *DOM Storage,* (2017).

[90] K. Yasuda, *Quota Management API,* (2015).

[91] MDN, *Storage API - Web APIs | MDN,* (2017).

[92] I. Hickson, *Web SQL Database,* (2010).

[93] W3C, *Indexed Database API (http://www.w3.org/TR/IndexedDB/),* (2013).

[94] Fyrd, *Can I use... Support tables for HTML5, CSS3, etc,* (2013).

[95] Cykelero et al, *Using IndexedDB - Web APIs | MDN,* (2018).

[96] A. Ilin, *HTML5 applicaties voor Maple TA and Mobius,* (2018).

[97] A. Feyerke, *Say Hello to Offline First,* (2013).

[98] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, *On Secure and Usable Program Obfuscation: A Survey,* arXiv preprint arXiv:1710.01139 (2017).

[99] M. Kleppe, *JSFuck - Write any JavaScript with 6 Characters: [](!+,* (2016).

[100] MDN, *Using Web Workers,* (2018).

[101] Apache, *Apache CouchDB Documentation: Technical Overview,* (2014).

[102] Apache, *Apache CouchDB Documentation: ACID properies,* (2014).

[103] Apache, *Introduction to Replication — Apache CouchDB Documentation,* (2018).

[104] IBM, *Cloudant Query - IBM Watson and Cloud Platform Learning Center,* (2018).

[105] Apache, *Configuring Clustering — Apache CouchDB 2.1 Documentation,* (2018).

[106] M. Rhodes, *CouchDB 2.0's read and write behaviour in a cluster,* (2015).

[107] PouchDB, *PouchDB Adapters,* (2018).

[108] G. Ornaghi, *Filtered replication: from Couch to Pouch and back,* (2015).

[109] M. Justicz, *Remote Code Execution in CouchDB,* (2017).

[110] NIST, *CVE-2017-12635,* (2017).

[111] YatHit Developer Network, *YDN-DB javascript library,* (2018).

[112] R. Munroe, *Exploits of a Mom,* (2018).

[113] N. Marz, *How to beat the CAP theorem - thoughts from the red planet - thoughts from the red planet,* (2016).

[114] J. Lin, *The Lambda and the Kappa,* IEEE Internet Computing **21**, 60 (2017).

[115] B. W. Lampson, *Hints for computer system designs,* ACM SIGOPS Operating Systems Review **17**, 33 (1983).

[116] O. Boykin, S. Ritchie, I. O 'connell, and J. Lin, *Summingbird: A Framework for Integrating Batch and Online MapReduce Computations,* Proc. VLDB Endow. **7**, 1441 (2014).

[117] R. Singh and P. J. Kaur, *Analyzing performance of Apache Tez and MapReduce with hadoop multinode cluster on Amazon cloud,* Journal of Big Data **3** (2016), 10.1186/s40537-016-0051-6.

[118] J. Kreps, *Questioning the Lambda Architecture - O'Reilly Media,* (2014).

[119] P. Helland, *Immutability changes everything,* Communications of the ACM **59**, 64 (2015).

[120] Apache ZooKeeper, *ZooKeeper,* (2018).

[121] Apache, *ZooKeeper Recipes and Solutions: Leader Election,* (2018).

[122] G. Sharpira, *Time Based Release Plan - Apache Kafka - Apache Software Foundation,* (2018).

[123] J. Kreps, *It's Okay To Store Data In Kafka,* (2017).

[124] D. Borthakur, *HDFS architecture guide,* Hadoop Apache Project http://hadoop apache … , 1 (2008).

[125] D. Borthakur, *Maximum number of files in hadoop,* (2008).

[126] G. MacKey, S. Sehrish, and J. Wang, *Improving metadata management for small files in HDFS,* Proceedings - IEEE International Conference on Cluster Computing, ICCC , 1 (2009).

[127] J. Liu, L. Bing, and S. Meina, *The optimization of HDFS based on small files,* Proceedings - 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology, IC-BNMT2010 , 912 (2010).

[128] N. Petrov and A. Tanev, *Software reliability modelling of risk automotive system,* in *IFAC Proceedings Volumes (IFAC-PapersOnline)*, Vol. 45 (IFAC, 2012) pp. 227–230.

[129] S. S. Rao, I. N. Sahitha, G. Sireesha, and P. Manoj, *Evaluating Software System Reliability Using Architecture Based Approach,* International Journal of Intelligent Information Systems **7**, 1 (2018).

[130] M. Monperrus, *Principles of Antifragile Software,* in *Companion to the first International Conference on the Art, Science and Engineering of Programming* (2017).

[131] A. Tseitlin, *The antifragile organization,* Communications of the ACM **56**, 40 (2013).

[132] T. Chandra, R. Griesemer, and J. Redstone, *Paxos Made Live - An Engineering Perspective (2006 Invited Talk),* Perspective **7**, 398 (2007).

[133] Y. Izrailevsky and A. Tseitlin, *Netflix Chaos Monkey Upgraded – Netflix TechBlog – Medium,* (2016).

[134] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar, *Why Does the Cloud Stop Computing?* Proceedings of the Seventh ACM Symposium on Cloud Computing - SoCC '16 , 1 (2016).

[135] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao, *Gray Failure,* Proceedings of the 16th Workshop on Hot Topics in Operating Systems - HotOS '17 , 150 (2017).

[136] J. Geerling, *Getting Gigabit Networking on a Raspberry Pi 2, 3 and B+ | Jeff Geerling,* (2017).

[137] The Apache Software Foundation, *Hardware Provisioning - Spark 1.2.1 Documentation,* (2018).

[138] CCSDS, *XML Telemetric and Command Exchange (XTCE), Recommended Standard, Issue 1 [CCSDS 660.0-B-1]*, Tech. Rep. October (Consultative Committee for Space Data Systems, 2007).