Exploring a Data Flow Design of the CARTA System

Zainab Adjiet
Department of Computer Science
University of Cape Town
adjzai001@myuct.ac.za

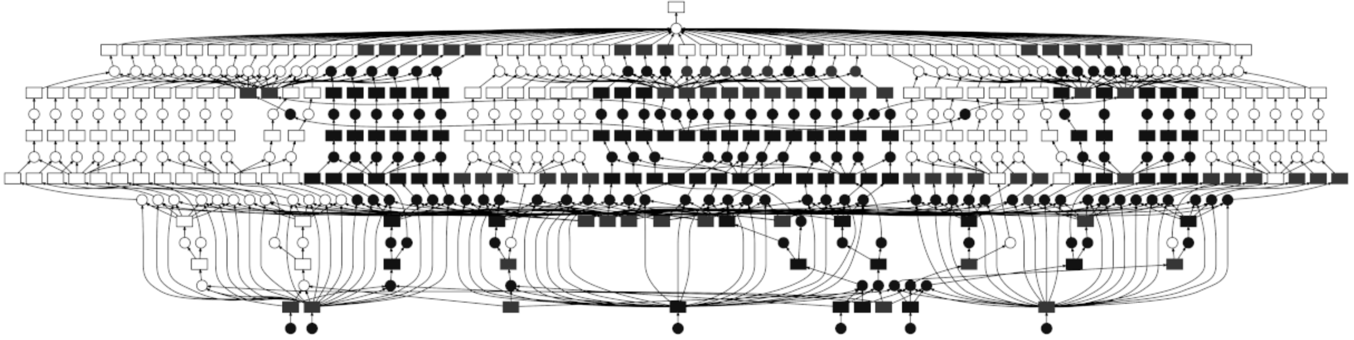

Figure 1: Distributed and parallel computing with Dask [9]

ABSTRACT

The Cube Analysis and Rendering Tool for Astronomy (CARTA) [7] is designed to visualise and analyse data from large modern telescopes. The CARTA code base consists of a front-end web client which receives processed information from the back-end server implemented in C++. One of CARTA's core libraries is being rewritten to experiment with the Dask [12] Python-based data flow environment to cope with processing the increasingly large data sets. This project aims to investigate the implications of such a change for the CARTA back-end system by conducting a backend re-design using this same Dask environment.

The CARTA back-end implementation along with its most recent Interface Control Document [8] were used to capture the requirements of the system and its various use cases. These use cases were represented in use case diagrams (App. A) and used to construct structural (App. B) and behavioural (App. C) UML diagrams to represent the proposed system design. The structural diagrams included package and design class diagrams and the behavioural included sequence and data flow diagrams.

The proposed design revealed that the data flow model using Python Dask leads to a simpler code base than the C++ system and better server modularity. The Python language leads to simpler code that is more comfortable to follow and Dask offers the benefit of executing operations over a scalable, possibly heterogeneous, cluster of machines with minimal input from the programmer. Apart from an expected performance decrease due to lack of optimisation, the shift to the Dask data flow environment for the CARTA back end is a worthwhile venture and Dylan Fouche's prototype lays reasonable grounds from which to start.

CCS CONCEPTS

• **Computer systems organization** → **Parallel architectures**; **Distributed architectures**; **Data flow architectures**; *High-level language architectures*; • **Software and its engineering** → *Massively parallel systems*; *Distributed systems organizing principles*; *Software prototyping*; Scheduling; • **Information systems** → *Enterprise information systems*.

KEYWORDS

dataflow, exascale, architecture, Python, visualisation, astronomy

1 INTRODUCTION

The Cube Analysis and Rendering Tool for Astronomy (CARTA) [7] is designed to visualise and analyse data from the Atacama Large Millimetre Array (ALMA) [2], the Very Large Array [28], and the Square Kilometre Array (SKA) [15] pathfinders. CARTA uses a client-server architecture to visualise the large images obtained from these modern telescopes as they would be challenging to process on personal computers or laptops. The data storage and computation are handled by enterprise-class servers or clusters, and the processed information is sent to the front-end web client for visualisation (see Fig. 2).

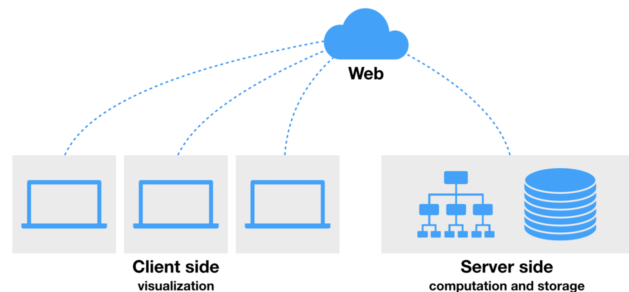

Figure 2: CARTA Client-Server Architecture [6]

The current CARTA back end [10] is implemented as a multi-threaded imperative program written in C++ [5], a high-performance, compiled language. CARTA [4] uses several libraries to provide its functionality, a key one being the Common Astronomy Software Applications (CASA) [25] library, which is maintained by a CARTA partner, the National Radio Astronomy Observatory (NRAO) [27], and predominantly implemented in C++ with an IPython interface.

As the image size produced by modern telescopes has been rapidly increasing over the past years, visualisation and analysis have become computationally expensive tasks. The design of software architecture is shifting as High-Performance Computing (HPC) moves into the exascale era, where computers are required to perform a quintillion calculations per second to cope with the increasingly large data sets that they are expected to process. With the first Exaflop computer expected around 2020 [19], the NRAO is exploring the use of a dataflow model [13] for the CASA library. The prototype will be implemented in an interpreted language, Python, with the use of the Python Dask library.

Dask [12] is a flexible Python library for parallel programming, which comprises dynamic task scheduling and parallel collections like arrays and lists for larger-than-memory environments. Computations are structured as directed acyclic graphs called task graphs, and these task graphs can be run with different schedulers. The parallel collections are then built atop this infrastructure and based on their single-threaded counterparts. Due to Dask's task graph structure, Dask programs are run in an implicit data flow environment which opposes the traditional von Neumann architecture used in regular sequential programs.

The data flow architecture differs from the traditional von Neumann architecture in that a program counter is not used to govern program flow. Instead, a *firing rule* specifies when instructions can execute, and this is based on the availability of the instruction input data. Furthermore, the architecture does not allow for global state storage, which eliminates side-effects [1].

Inspired by the work being done by the NRAO, an architectural re-design of the current of the CARTA back end system was investigated using the Python-based Dask data flow environment to explore the implications of this shift to a data flow model. Alongside this re-design, a prototype was implemented by Dylan Fouche, which was ofttimes used as inspiration for the design.

This report will first cover the related work done on the topic of using the data flow architecture over the traditional architecture. The methods used to investigate the aims of this project will then be explored along with references to various constructed diagrams to explain the novel system design. The report will then cover how the design was evaluated and the results of the evaluation. Conclusions will be drawn on what the project findings imply for implementation of the CARTA back end system using the Python Dask data flow environment.

2 RELATED WORKS

Jack Dennis [17] presented the first concept of the data flow architecture in 1974. While not as popular as its contrast, the data flow model can offer advantages for parallel processing. Some aspects of this model could also be combined with the traditional model to form a hybrid with combined advantages. Since the flow of data determines the flow of control in a data flow model, the model can be represented as a directed graph (see Fig. 3) with the *nodes* as the instructions or functions and *arcs* connecting nodes as the data dependencies. The input and output data that flows along the arcs are termed as *tokens*, and these tokens contain a *tag* that identifies the token's destination node [1]. Not only is this representation intuitive, but it can also present some advantages for program design [31].

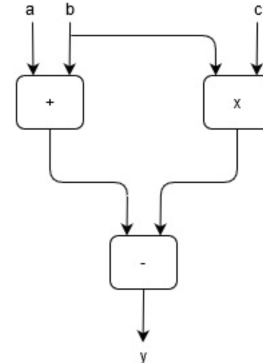

Figure 3: Simple Data-Flow Model of $y = (a + b) - (b * c)$

The data flow model offers efficient use of fine-grain parallelism for computation [22]. It offers implicit parallel task synchronisation, and does not constrain instruction sequencing apart from enforcing data dependency constraints. It also tolerates memory latency as it processes other instructions while waiting for a response from memory, and thus has this advantage over the von Neumann control flow model as well [13]. Data flow programs can be composed effortlessly to form more extensive programs by connecting the output of one graph to the input of another [16, 31], which presents a way of dividing a program into distinct components.

Iannucci [21] describes the von Neumann and data flow architectures as two extremes on a spectrum of architectures instead of being independent of each other and speculates that there is a whole family of architectures between these extremes as well as some optimum point. A typical hybrid of these two models can be termed as a *coarse-grain data flow* model [31], and this groups instructions into larger instruction blocks or grains instead of smaller tasks as in pure data flow. These grains are then scheduled using the data flow approach and the instructions within the grains are scheduled using control flow, which combines the data flow's exploitation of parallelism and the control flow's efficient execution [22].

When designing a multi-threaded program, a compiled language like C or C++ will be favoured because this offers custom optimisation according to the developer's requirements. Traditionally, if there were no critical performance requirements, interpreted languages were a good substitute as they offer simplicity, and Python, in particular, offers a variety of useful modules. More recently, Python has attracted developers in the scientific community as it allows them to build custom environments based on compiled languages like Fortran, C and C++ [14]. Hence, Python is being used more often in large-scale, parallel applications.

The Dask library [12] provides a data flow environment for Python and while it offers simple, powerful tools for parallel programming, many similar tools exist. Mehta et al. [26] have evaluated how various Python libraries perform when completing tasks on large-scale image analysis systems. Dask is shown to have more scheduling overhead than other Python libraries, Spark [3] and Myria [29], when processing smaller, partitioned data sets (see fig. 4. However, Dask does outperform its competitors with a faster run time on larger data sets which is attributed to its more efficient pipelining and data caching capabilities after overcoming the initial start-up overhead as seen with the smaller data sets.

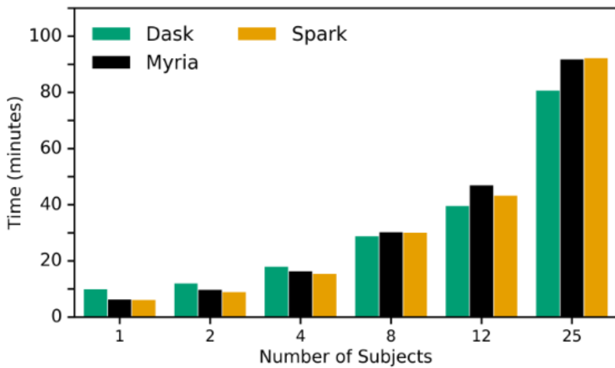

Figure 4: Run time of analysing diffusion MRI data using different parallel computing Python libraries with varying data size [26]

Since the computation and schedulers in the Dask environment are separated, this allows the same unaltered task graph to be computed on different schedulers, each with its own performance characteristics [12]. This gives developers the benefit of being able to choose how they want their program to execute. Dask schedulers also operate dynamically, which means that the execution order is determined during run-time rather than before [30]. Not only does this cater for uncertain execution environments, but it also minimises memory usage during execution. Scheduling overhead is also minimised by maintaining state about the current computation to select the next task quickly.

3 METHODS

3.1 Requirements Capturing

The requirements of the proposed system mirror the requirements of the current CARTA back end system. These requirements were mainly captured by reviewing an Interface Control Document for the CARTA system [8], which mostly comprised sequence diagrams for each use case of the system. Using this document, along with the open-source CARTA Github repository [4], various use case diagrams were composed to capture a high-level view of the system and its requirements.

The use case diagrams, along with all other diagrams discussed in the following sections, were constructed using the free, online Diagrams.net tool [18]. First, a typical use case diagram was constructed with a CARTA client user as the focus actor of the diagram to obtain system requirements from the user's view. While some subroutines for the use cases could be identified, these subroutines are the focus of the back-end design, so a second use case diagram was created using the CARTA front end as the focus actor (Fig. 7). This second diagram allowed a more in-depth look into the needed functionality of the CARTA back-end system.

3.2 Static System Design

A package diagram, paying specific attention to modules used, was constructed to depict how the back end should interact with the existing front-end packages and what other packages would be necessary for its functions. The diagram is based on Dylan Fouche's Dask prototype of the CARTA back end [20] and depicts the Python modules that the back end needs to calculate a per-cube histogram.

After gaining a high-level view, a design class diagram was used to narrow down on the classes and functionality within the back end. Fouche's prototype was used again as a base for the relationships between the classes and some initial class functionality.

Both the package and class diagrams were then revised and restructured (Fig. 8) to take the full system functionality into account. The CARTA Github repository [10] was used as a reference of how the back end can be structured to support the full functionality and inspiration was drawn from this to restructure the design.

3.3 Dynamic System Design

Data flow diagrams show how data moves between entities and processes in a system, and this can give the best representation of the Dask data flow environment. These diagrams provide a good view of how the system should behave irrespective of the structure of classes and entities; thus, they were constructed earlier in the investigation. A zero-level data flow diagram, commonly referred to as a context diagram, was constructed for the histogramming function of the CARTA system. Still focusing on this operation, another two data flow diagrams were constructed each with varying levels or depths.

The first-level diagram depicts the movement of data with the histogramming operation being run on a single computing node. The second-level uncovers the Dask scheduler entity embedded in the back end that orchestrates the operations on a set of Dask worker nodes (Fig. 10). It was essential to show that processes could run concurrently on these worker nodes and representation for this was decided upon with the guidance of Kechil Kirkham, a systems engineer at the Inter-University Institute for Data Intensive Astronomy. Based on the second-level histogramming data flow diagram, a generalised second-level diagram was created (Fig. 11). This diagram focuses on the interaction of the Dask scheduler with individual worker nodes and shows how the flow can be generalised.

Contrary to a data flow diagram, a sequence diagram shows the behaviour of the system paying specific attention to the execution order of system interactions. A sequence diagram was constructed (Fig. 9) to depict the flow of the histogramming operation as it would execute on the revised version of Fouche's back end prototype [20]. This sequence diagram expands on how the back-end classes interact with each other to compare this to how the existing CARTA back end [10] executes the same operation.

4 EVALUATION

Due to the software engineering design nature of this project, the design could not be quantitatively tested in terms of speedup unless it were to be fully implemented; thus, the design underwent qualitative testing. The design documentation itself was continuously evaluated by Kechil Kirkham to ensure the proper design documentation practises were exercised. The actual design, however, was evaluated against Dylan Fouche's implementation [20] and the existing CARTA back end [10] to identify any noticeable differences.

The design itself was also evaluated against the requirements captured from the existing system to note if there is any functionality missing or possibly functionality that cannot be added to the data flow design. The proposed system was checked for any significant performance issues or benefits inherent of the design and if it can be fully implemented using Python Dask.

5 FINDINGS AND DISCUSSION

5.1 Scope of Proposed Design

Use case diagrams can abstract the complexity of a system and help focus the system design on operations that directly impact the system user, ultimately making the system more proactive [24]. However, this simplicity makes it challenging to capture the interactions between requirements as well as non-functional requirements [23]. For this reason, the proposed design based on the use case diagrams (Fig. 7) overlooks these types of requirements which may exist for CARTA, leading to a substandard design.

Furthermore, the proposed system design structure in the revised class diagram (Fig. 8) does not cover the full functionality of the current CARTA back-end system due to time constraints. The design can be intuitively extended, however, by following the pattern of the back-end class structure and extending the classes as necessary.

5.2 Server Component Modularity

The revised class diagram differs from the other based on Fouche's prototype [20] in that there exists more separation of roles in the back end than all responsibilities laying on one *Server* class.

The server in the proposed design is only liable for receiving messages from the front end and managing the Dask Distributed cluster. All front-end messages will be handled in a *Session* class which initiates the correct corresponding operation depending on the incoming message. Once the operation is completed or runs into an error, a response message will be constructed and passed back to the front end. This operation flow can make for a more logical system structure which more closely mirrors the CARTA back end implementation.

As established in the related works, data flow models can be inherently modular [16, 31], and this can be seen by how easily the histogramming operation can be abstracted to any operation. The Dask scheduler takes a function and the corresponding data as input and uses the worker nodes to execute this function. The second-level general data flow diagram (Fig. 11) shows how the scheduler would interact with each node in the cluster. This generalisation will aid the modularity of the back-end system and minimise the amount of code to be written for each different operation.

5.3 Overall Code Simplicity

The proposed structure does not offer any more benefits than the current back-end structure, and if the shift to the Dask data flow environment is made, it will be the case of duplicating the existing structure to Python. However, the dynamically-typed nature of Python does allow for a more straightforward code base, especially in the case of mapping message types to functions in a dictionary (see Fig. 5). The methods in the *Server* class can stay relatively simple, and there is no need for lengthy switch functions to cater for each type of front-end message as in the CARTA back end.

```
MESSAGE_TYPE_CODE_TO_EVENT_HANDLER = {
    enums_pb2.EventType.REGISTER_VIEWER: __on_register_viewer,
    enums_pb2.EventType.OPEN_FILE: __on_open_file,
    enums_pb2.EventType.SET_HISTOGRAM_REQUIREMENTS: __on_set_histogram_requirements,
    enums_pb2.EventType.SET_STATS_REQUIREMENTS: __on_set_statistics_requirements
}
```

Figure 5: Example of Python Event-Mapping Dictionary [20]

The simplicity of Python can offer benefits in terms of system development as well. It can make the code base more comfortable to follow and thus easier to modify and extend if need be. In development teams that can vary and change, this will also ensure that a developer who is not accustomed to the CARTA system can easily understand how it works compared to being introduced to an intricate, well-developed C++ code base.

5.4 Distributed Environment and Scaling

The progressively more detailed data flow diagrams for the CARTA histogramming operation shows how well the operation can be adapted to a data flow model. This view makes it easy to see how any sub-process, or any process for that matter, can be run concurrently by passing the data to another node or process. Furthermore, the parallel combined fragment (see Fig. 6) in the sequence diagram shows that there is a possibility for part of the histogramming operation to be run concurrently as these operations only perform data accesses.

The second-level data flow diagram (Fig. 10) shows how the Dask scheduler plays the role of initiating the operation on the cluster of one-to-many Dask worker nodes. The histogram construction is distributed across the cluster of worker nodes, and the scheduler is responsible for assigning the tasks and data accordingly. While the diagram does not show this, the scheduler can choose to assign different processes to different nodes and the data would be gathered again and sent back to the front end.

This diagram also depicts the operation as it would compute on the cluster as a whole and abstracts from machine specifications of the individual cluster nodes. In a typical control flow environment, the programmer is required to specify which process should run on a which thread, as is the case with C++. Dask, however, handles this duty entirely and need only be supplied the cluster of machines to execute the operation over. Furthermore, this does not restrict

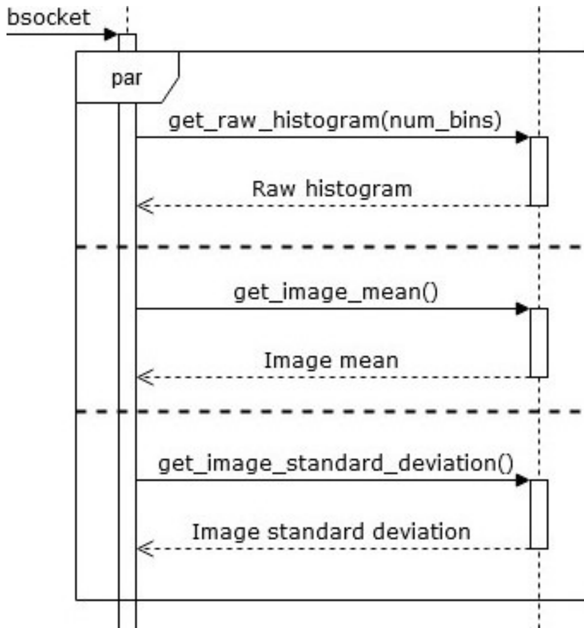

Figure 6: Parallel Combined Fragment of CARTA Per-Cube Histogram Calculation Sequence Diagram

the Dask cluster machines to any one type of machine specification, and the back-end operations can potentially be executed over heterogeneous clusters.

Dask can also allow for better scalability of the CARTA server than what is currently used. The only way to scale the CARTA server to account for increased performance requirements is to increase the number of cores on the single server machine. This method of scaling is not sustainable for the system as there exists a limit to the number of cores a machine can consist of, and with HPC approaching the exascale era [19], this limit may be surpassed soon. This outcome can be avoided with Dask as it provides the ability to scale computation over clusters of thousands of machines [11], allowing the CARTA server to scale as necessary.

6 CONCLUSIONS AND FUTURE WORK

Shifting the CARTA back-end system to a data flow model offers the benefit of code simplicity. The structure of the proposed system is logical and can be easily extended to include the full functionality of the current back end. The Python language itself leads to simpler code that is easier to understand and follow than the current C++ code, which may make the code base easier to modify and extend if need be.

The Dask data flow environment with the Dask scheduler allows the execution of CARTA operations to be generalised so that the server need only pass the necessary function and data to the scheduler to execute the required operation. This generalisation adds to the modularity of the server and allows for easy expansion of the operation list with minimal code additions.

While other Python parallel libraries do exist, Dask proves to outperform these libraries when applied to larger data sets. Dask offers the benefit of effeciently executing operations over a scalable, possibly heterogeneous, cluster of machines as it handles the distribution of tasks and data with no extra effort from the programmer.

While the design itself offers no drawbacks, the shift to Python from C++ may result in a notable performance decrease for some or all CARTA operations as the CARTA system is already optimised to perform these tasks. Nonetheless, the shift to the Dask data flow environment for the CARTA back end is a worthwhile venture. It can provide many long-term benefits for the CARTA system going forward, and Dylan Fouche's prototype lays reasonable grounds from which to start.

The structure of the current CARTA system is logical and efficient. Given more time, it would be preferable to examine the CARTA code base further and draw more inspiration from the system structure for the proposed design. The CARTA system could also be further analysed for non-functional requirements so these can be incorporated into the proposed system design.

REFERENCES

[1] Tilak Agerwala et al. 1982. Data Flow Systems: Guest Editors' Introduction. *Computer* 15, 2 (1982), 10–13.
[2] A Worldwide Collaboration ALMA. 2020. Atacama Large Millimeter/submillimeter Array. https://www.almaobservatory.org/en/home/
[3] Apache. 2020. *Welcome to Spark Python API Docs!* http://spark.apache.org/docs/latest/api/python/index.html
[4] NRAO ASIAA, IDIA. 2020. *Cube Analysis and Rendering Tool for Astronomy.* https://doi.org/10.5281/zenodo.3377984
[5] British Standards Institute. 2003. *The C++ Standard: incorporating Technical Corrigendum 1: BS ISO* (second ed.). Wiley, New York, NY, USA. xxxiv + 782 pages.
[6] CARTA. 2018. CARTA Client-Server Architecture. https://carta.readthedocs.io/en/latest/_static/carta_intro_serverClient.png
[7] CARTA. 2018. *Cube Analysis and Rendering Tool for Astronomy.* https://carta.readthedocs.io/en/latest/index.html
[8] CARTA. 2020. *CARTA Interface Control Document — CARTA Interface Control Document documentation.* https://carta-protobuf.readthedocs.io/en/latest/index.html#
[9] Anaconda Cloud. 2020. Distributed and parallel computing with Dask: Memory and Parallelism. http://matthewrocklin.com/slides/images/grid_search_schedule.gif
[10] CARTAvis community. 2020. *CARTA Backend.* https://github.com/CARTAvis/carta-backend/
[11] Dask core developers. 2014. *Why Dask? — Dask documentation.* https://docs.dask.org/en/latest/why.html#dask-scales-out-to-clusters
[12] Dask core developers. 2019. *Dask: Natively scales Python.* https://dask.org
[13] David E Culler. 1986. Dataflow architectures. *Annual review of computer science* 1, 1 (1986), 225–253.
[14] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. 2008. MPI for Python: Performance improvements and MPI-2 extensions. *J. Parallel and Distrib. Comput.* 68, 5 (2008), 655–662.
[15] David Davidson. 2012. MeerKAT and SKA phase 1. *2012 10th International Symposium on Antennas, Propagation and EM Theory, ISAPE 2012*, 1279–1282. https://doi.org/10.1109/ISAPE.2012.6409014
[16] Alan L Davis and Robert M Keller. 1982. Data flow program graphs. (1982).
[17] Jack B Dennis. 1974. First version of a data flow procedure language. In *Programming Symposium.* Springer, 362–376.
[18] diagrams.net. 2005. Diagram Software and Flowchart Maker. https://www.diagrams.net/
[19] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. 2011. The international exascale software project roadmap. *The international journal of high performance computing applications* 25, 1 (2011), 3–60.
[20] Dylan Fouche. 2020. *CADaFloP.* https://github.com/DylanFouche/CADaFloP/
[21] Robert A Iannucci. 1988. Toward a dataflow/von Neumann hybrid architecture. *ACM SIGARCH Computer Architecture News* 16, 2 (1988), 131–140.
[22] Ben Lee and Ali R Hurson. 1994. Dataflow architectures and multithreading. *Computer* 27, 8 (1994), 27–39.
[23] J. Lee and Nien-Lin Xue. 1999. Analyzing user requirements by use cases: a goal-driven approach. *IEEE Software* 16, 4 (1999), 92–101.
[24] M. Lorenz. 1993. *Object-oriented Software Development: A Practical Guide.* PTR Prentice Hall. https://books.google.co.za/books?id=mJ9QAAAAMAAJ

[25] J. P. McMullin, B. Waters, D. Schiebel, W. Young, and K. Golap. 2007. *CASA Architecture and Applications*. Astronomical Society of the Pacific Conference Series, Vol. 376. 127.

[26] Parmita Mehta, Sven Dorkenwald, Dongfang Zhao, Tomer Kaftan, Alvin Cheung, Magdalena Balazinska, Ariel Rokem, Andrew Connolly, Jacob Vanderplas, and Yusra AlSayyad. 2016. Comparative evaluation of big-data systems on scientific image analytics workloads. *arXiv preprint arXiv:1612.02485* (2016).

[27] National Radio Astronomy Observatory. 2020. *National Radio Astronomy Observatory*. https://public.nrao.edu/

[28] National Radio Astronomy Observatory. 2020. *Very Large Array - National Radio Astronomy Observatory*. https://public.nrao.edu/telescopes/vla/

[29] PyPI. 2020. *myria-python*. https://pypi.org/project/myria-python/

[30] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*. Citeseer.

[31] Jurij Silc, Borut Robic, and Theo Ungerer. 1998. Asynchrony in parallel computing: From dataflow to multithreading. *Parallel and Distributed Computing Practices* 1, 1 (1998), 3–30.

A REQUIREMENTS CAPTURING DIAGRAMS

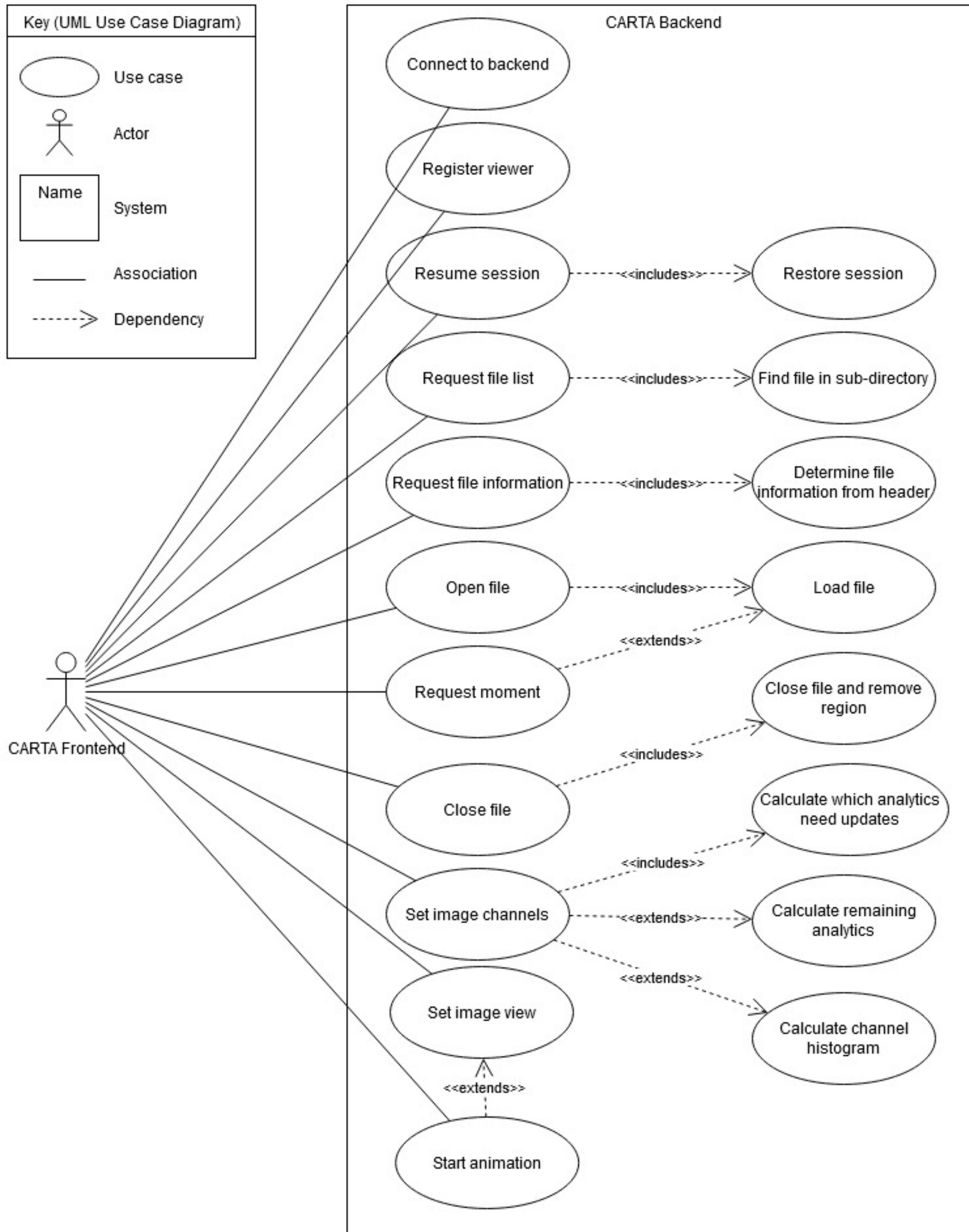

Figure 7: Part of CARTA Front End Use Case Diagram Set

B STATIC DESIGN DIAGRAMS

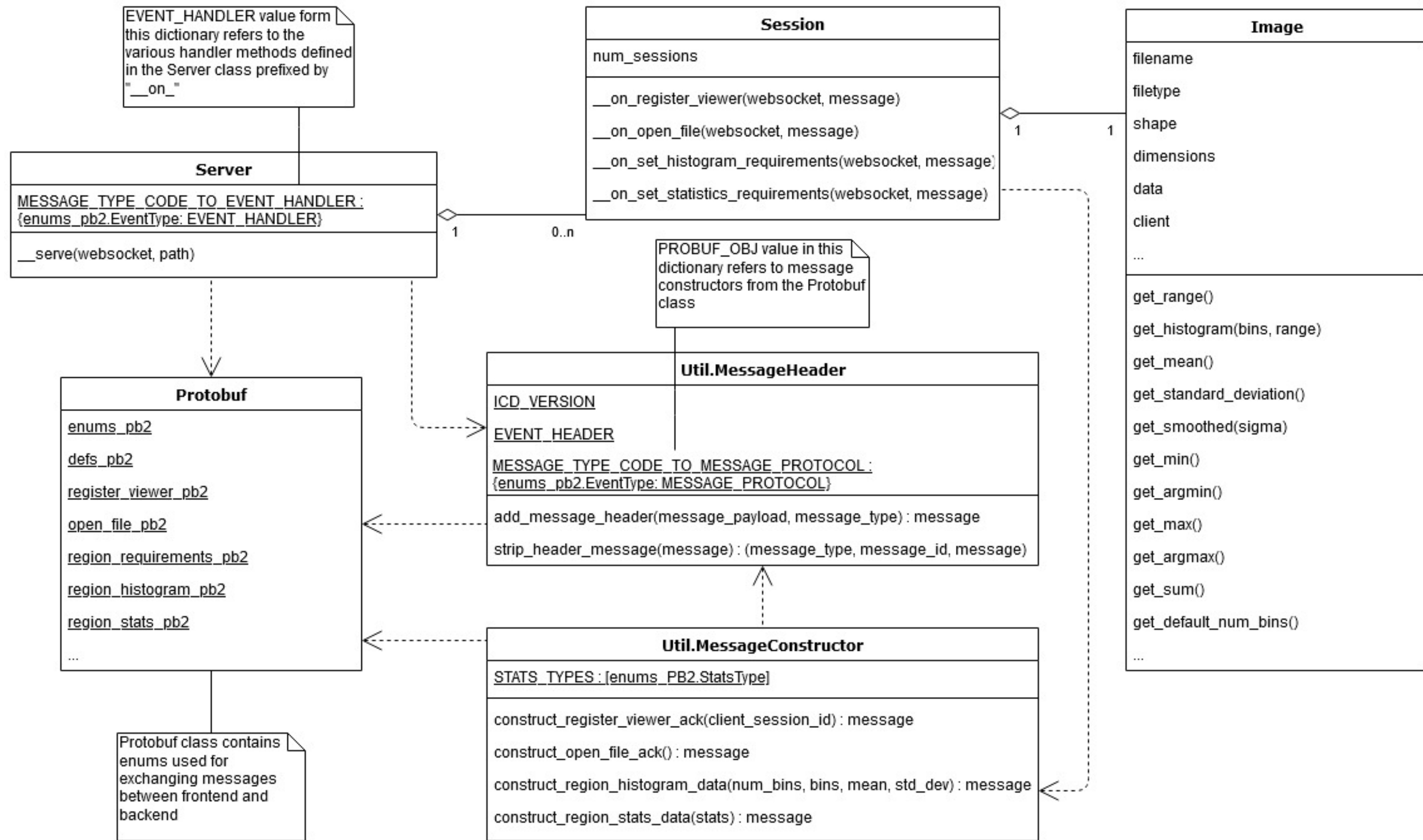

Figure 8: Revised CARTA Design Class Diagram

C DYNAMIC DESIGN DIAGRAMS

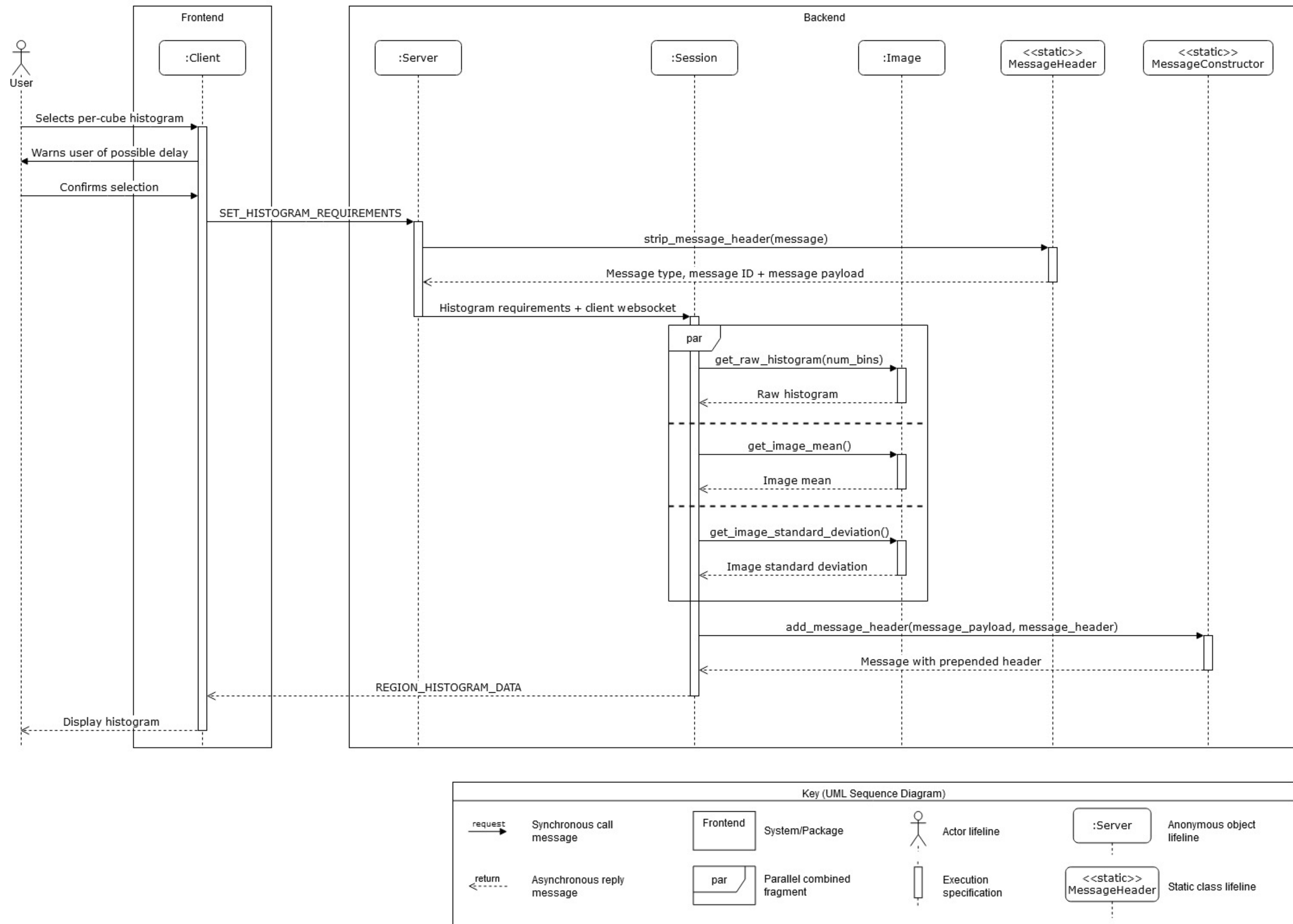

Figure 9: CARTA Per-Cube Histogram Calculation Sequence Diagram

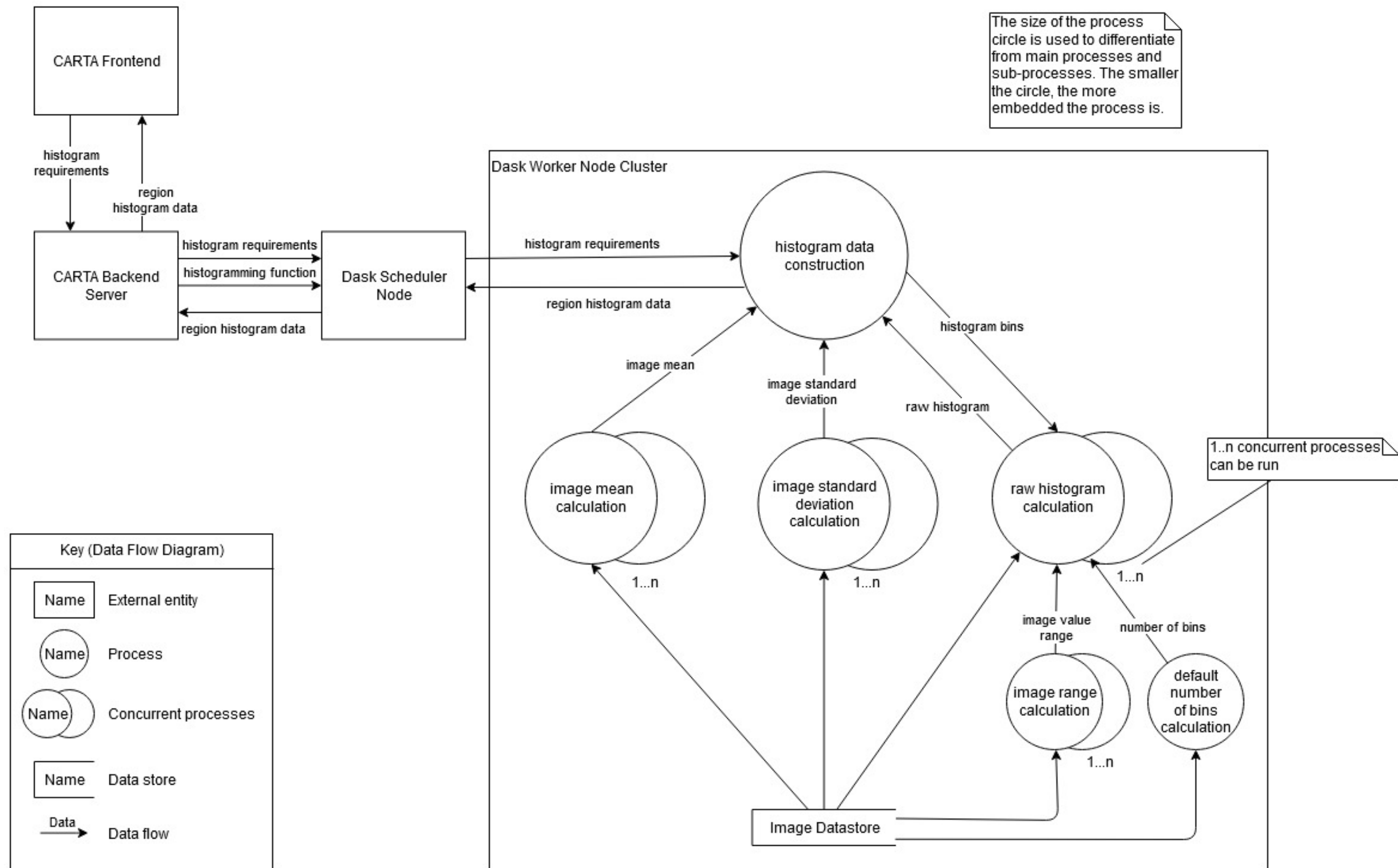

Figure 10: CARTA Per-Cube Histogram Calculation Data Flow Diagram (Level 2)

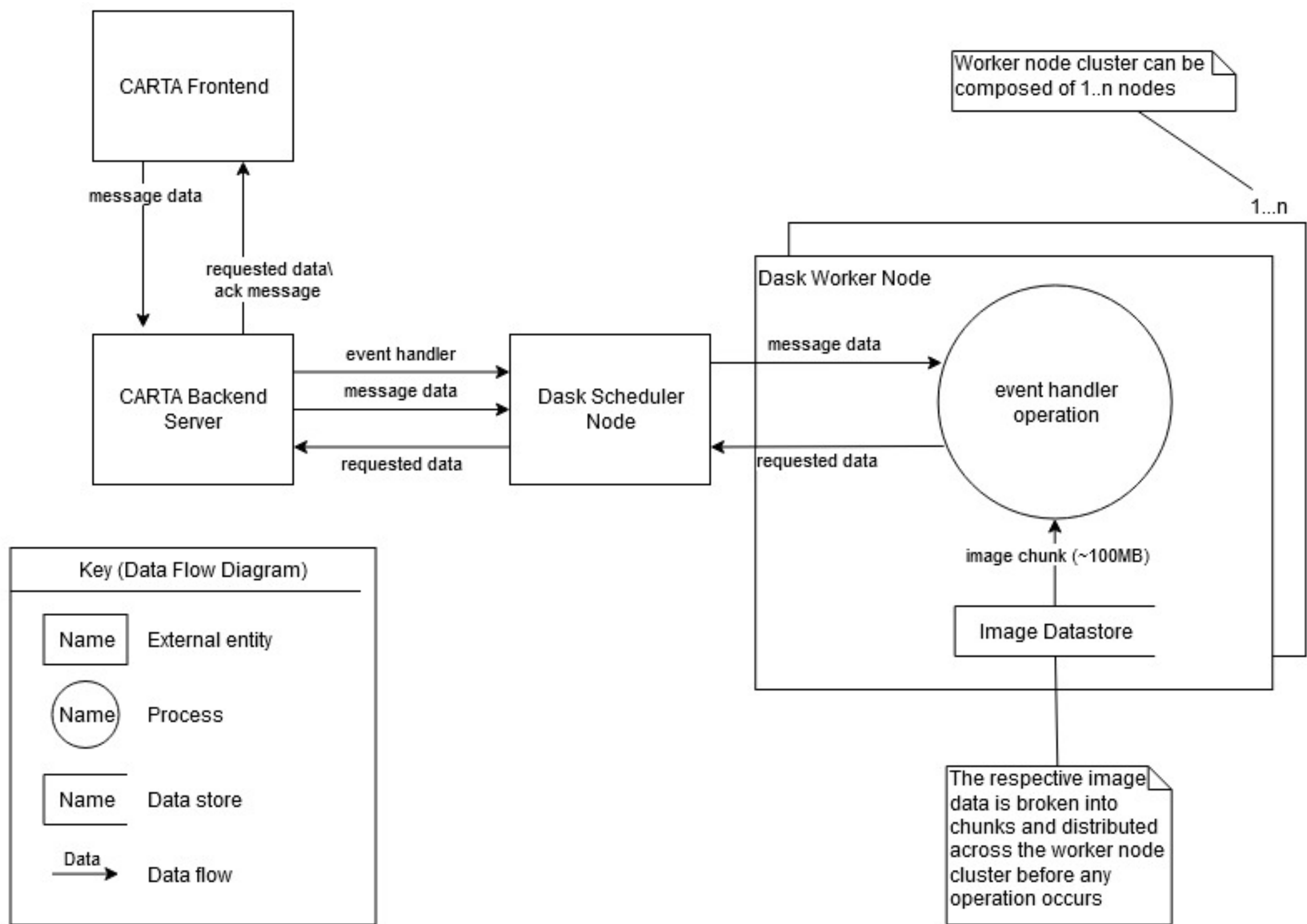

Figure 11: CARTA Abstracted Data Flow Diagram (Level 2)