SISSA
Scuola Internazionale Superiore di Studi Avanzati

CERN

UNIVERSITÀ DEGLI STUDI DI TRIESTE

Department of Mathematics and Geosciences
Master's Degree in Data Science and Scientific Computing

Improving throughput of CERN's multi-language distributed online data processing platform

Candidate:
Francesco Andreuzzi

Advisor:
Prof. Gianluigi Rozza
Co-advisor:
Dr. Nicola Demo
External advisor:
Dr. Vito Baggiolini

Academic Year 2022-23

Acknowledgments

I would like to thank first and foremost my family, my parents Alessandra and Giorgio who have always cheered my curiosity, my sister Laura, and Nicol, for supporting me despite the distance.

Secondly, I want to thank my internal advisors from SISSA: Nicola, for his help during the development of this thesis, and for his invaluable assistance and mentoring during my months at *mathLab*; and Prof. Rozza, for giving me the opportunity to join his great research group.

Finally, thanks to all my colleagues from the CMW team, Wojtek, Marcin, Lajos, Martin, and my external supervisor Vito, for their relentless review work and feedback on the project.

Summary

The present master thesis investigates optimization strategies for a critical software framework developed at the *European Organization for Nuclear Research* (CERN). Leveraging a performance-aware serialization strategy, and a carefully designed distributed communication pattern, we aim to enhance the throughput of a critical online data processing platform largely adopted by the engineers, physicists, and equipment specialists who control and monitor the particle accelerators and the particle beam produced. The work presented in this thesis has been implemented successfully and validated in a production-like environment with a realistic data inflow. The expected improvements were measured and reported as part of the present document.

This thesis is organized as follows:

- Section 1 (*Context*): we will introduce the context, starting with a brief introduction to the scientific goals and the organization of CERN, and a general discussion on data processing software. We will then provide a brief overview of the software that we aim to optimize, focusing in particular on its architecture and current limitations.
- Section 2 (*Inter-process communication*): we are going to dive into relevant topics for the practical portion of the work, namely inter-process communication (IPC), data formats, and data serialization. For each of these subjects, we will discuss both the theoretical parts, to gain valuable insights to be leveraged in the implementation, and the practical aspects, namely available standard alternatives in the industry and their shortcomings.
- Section 3 (*Language interoperability in UCAP*): finally, we will present our approach for the optimization, highlighting the advantages with respect to the legacy approach and the expected performance gains. We will then proceed with a practical benchmark in a production-like environment to assess the materialization of the theoretical improvements.

Contents

List of Figures

List of Tables

List of Listings

1 Context

In this first section, we are going to provide some general context on the work presented in this document. We are going to open the discussion with a general overview of CERN, its internal organization, and its functions in the scientific community. Then, we will motivate the importance of *data processing platforms*, and we will briefly discuss a possible categorization. After that, we are going to examine one of the most important practical requirements of data-processing software products, namely cross-language interoperability, which is going to be the main driver of the present work. Finally, we will present a high-level overview of the software which we aim to optimize as part of the work, as well as the practical motivations which have driven the work and make it an important contribution to the CERN control system.

1.1 European Organization for Nuclear Research

The *European Organization for Nuclear Research* (CERN) is the world's largest particle physics laboratory. It was founded in 1954 by a consortium of twelve European countries, and throughout the decades it grew both by funding and member states [66]. CERN hosts technicians, engineers, and scientists from all over the world, thanks to international collaborations with Universities and institutions. The main research area of the laboratory is the field of *high-energy physics*; however the technical challenges faced every day while operating the massive accelerator complex hosted at CERN requires fast-paced innovation in a wide span of knowledge areas, like Software, Civil, Mechanical Engineering [18,118], as well as Electronics, Cryogenics, and Material Sciences.

CERN has been the cornerstone of several scientific achievements in both the 1900s and the first decades of the 2000s. The most well-known is arguably the observation of the long-sought *Higgs boson* in the context of the ATLAS collaboration [1], which was predicted by the *Standard Model* but never observed experimentally. However, as we stated earlier, CERN introduced important innovations in other areas, not necessarily related to experimental physics. An important example is the development of the World-Wide Web, which was invented by Tim Berners-Lee during his residency in the laboratory [13].

In the next two paragraphs, we are going to present very briefly the focal point of CERN, namely its important accelerator complex; then, we are going to get closer to the main topic of the present work, namely the control system, which enables human operators to interact reliably with the complex machinery that controls the experimental infrastructure. This small introduction to the topic, although not exhaustive, will be beneficial to understanding the importance of

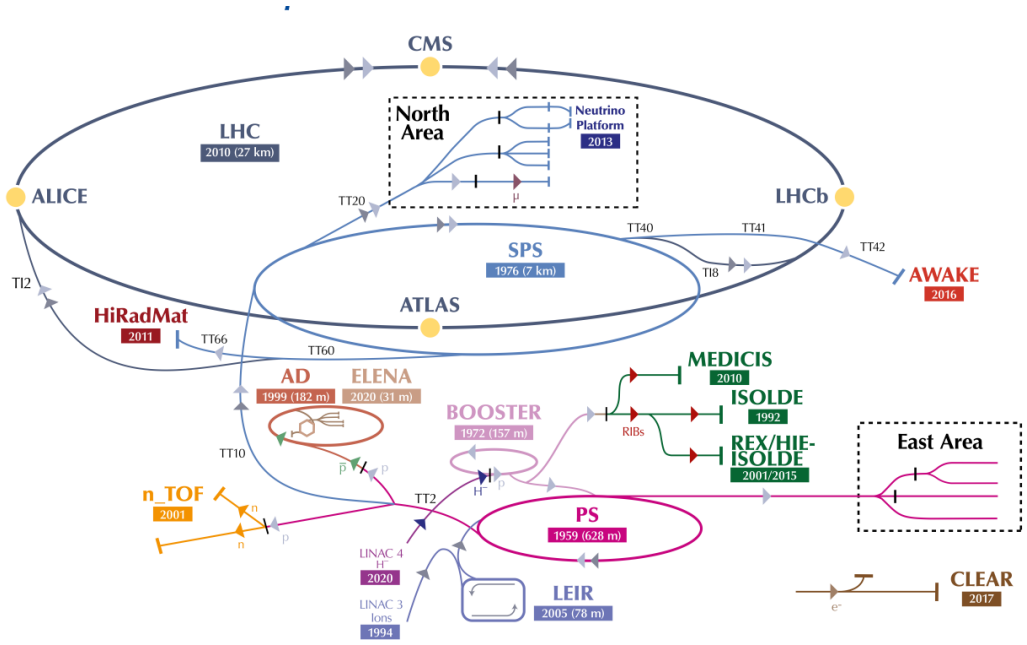

Figure 1: The CERN accelerator complex as of 2024 [91]. The picture depicts also the dependency relations between the accelerators. The *Large Hadron Collider* (LHC) is the largest oval (dark grey) in the picture; the related experiments (CMS, ALICE, ATLAS, and LHCb) are highlighted on the perimeter.

the work discussed in the present document.

Accelerator complex

The CERN accelerator complex (depicted schematically in Figure 1), is comprised of many machines that serve a diverse set of needs. The high-level idea is to accelerate particles, typically protons or ions, until they match an energy level suitable for collisions. This is the reason for the interconnections represented in Figure 1: machines occurring earlier in the chain are suitable for picking up particles at lower energies and accelerating them before pushing them to the next stage. An example chain is summarized in Table 1, which prepare protons for collisions in the *Large Hadron Collider* [25].

In addition to the accelerators, the complex includes several experimental facilities. Experiments related to LHC, namely CMS, ALICE, ATLAS, and LHCb, are definitely some of the most well-known ones. Moreover, SPS and PS respectively serve the *North Area* and the *East Area*; each of them comprises several

Table 1: The CERN Accelerator injection chain for protons. The uppermost row corresponds to the first accelerator in the chain.

Accelerator name	Acronym	Year	Peak energy level
Linear Accelerator 4	Linac4	2013	160 MeV
Proton Synchrotron Booster	PSB	1972	2 GeV
Proton Synchrotron	PS	1959	26 GeV
Super Proton Synchrotron	SPS	1976	450 GeV
Large Hadron Collider	LHC	2008	7 TeV [109]

experiments that target a diverse set of research areas. A few examples are the *Antiproton Decelerator* (AD), whose purpose is to study the properties of antiparticles like anti-hydrogen, and CLOUD, which aims to study the interaction between elementary particles and the atmosphere [25].

The accelerator complex is, relatively speaking, an extremely dynamic infrastructure. As the reader may infer from the third column of Table 1, accelerators were commissioned at different times in history; in a few cases, new machines replaced obsolete ones in the middle of an existing chain, to improve the efficiency and the *luminosity* (i.e. the number of collisions) of the complex. Ongoing discussions are being carried out to initiate the construction of a new, massive accelerator, which will surpass LHC in terms of dimension and peak energy. This humongous project, which will set important challenges both in terms of engineering and funding, is approximately scheduled for completion in 2040 [12].

Control system

Interacting with the tens of thousands of actuators, detectors, and schedulers that drive the accelerator complex requires an important software infrastructure to streamline the work of human operators. A few example use cases are monitoring accelerator variables, like temperature, magnet currents, beam position and size, as well as controlling relevant reference variables to impose the desired behavior on a machine [35], which is commonly referred to as "steering the machine". Scaling these operations on the scale required by the laboratory is an important effort that requires careful design and development of efficient and reliable solutions. This set of software artifacts is commonly referred to at CERN as the *Control System*. The control system provides solutions to translate inputs given by human operators into low-level operations on hardware components, database CRUD actions, as well as message events on distributed messaging systems like Apache Kafka or JMS. Similarly, it enables a unified acquisition and processing mechanism

to monitor and process relevant variables in an online/offline fashion [35].

The Control System comprises both low-level software, which is intended to be used by experts and developers, as well as high-level applications, which are directly targeted at operators running the accelerator complex. Several examples of the first category are:

- RDA [39, 90, 151], the default solution for remote read/write/subscribe operations;
- JAPC [6], which provides unified access to the diverse set of data sources included in the Control System through a simple Java API.

The second category is well represented by:

- CESAR [51], which aims at simplifying the experience in the context of the management of the many short-lived experiments within CERN experimental areas;
- LSA [77], whose purpose is to streamline storing and retrieving configuration values for accelerators and experiments.
- NXCALS [163], which provides a unified and robust solution for long-lived data, considering storage, querying, as well as visualization, supporting an important influx of data generated by accelerators, experiments, and user-defined post-processing pipelines.

The software whose architecture we are going to discuss in the present document is a recent addition to the Control System, which builds on previously existent software to provide a simplified and more reliable solution to the members of the experimental community.

1.2 Data processing frameworks

Data processing platforms are critical pieces of software in any technological facility [132]. This need is mostly driven by the important digital transition that the industry has faced during the last decade [54], namely the digital transformation of an impressive number of companies [40] and the transition toward data-friendly computing models like Cloud Computing [85]. In addition, we have witnessed the emergence of data-greedy computational needs, for example in the context of Artificial Intelligence (AI), whose declinations require huge amounts of data to obtain an accurate model [108]. With some approximation, data processing platforms may be described as scalable, robust, and versatile computing frameworks. The

range of use cases is extremely vast, ranging from data aggregation and transformation [142], simulation of physical complex systems [9] and high-performance look-up into humongous data collections [3].

The main categorization of data processing platforms, which we are going to present shortly, is mainly driven by the emergence in the last few decades of recurrent computing patterns. Citing only a few examples:

- *Extract, Transform, Load* (ETL) [156] was presented as a very general yet simple computing concept. It consists in loading a chunk of data in memory from a persisted data storage, transforming it in bulk by applying some user-defined function, and then loading the output in the storage. ETL is a very general pattern that may be used to model a plethora of use cases.

- The *Map-reduce* paradigm [34], as the name suggests, consists in the subsequent application of two user-defined functions. The first maps entries from a potentially distributed file system (e.g. HDFS [87]) to a key/value pair. Values are then arranged into buckets according to the key. Finally, buckets are reduced to a single value by applying the second user-defined function.

- *Event-sourcing* is a peculiar pattern for the design of complex software products. The main idea is to let the whole application evolve as a response to immutable events appended to a log-like queue. The queue contains the whole ordered set of events from the application start-up [87, 158]. This architectural pattern provides important scalability guarantees, thanks to the possibility of scaling the queue and the components around it independently. Moreover, the queue can be replayed at any time, thus allowing to restore the state at any given instant in time.

Following the identification of those patterns, there have been important efforts to standardize their definition and to provide generic implementations. Nowadays, the open-source community provides and maintains actively many solutions for each possible use case. Apache Airflow [64] has gained some traction in the context of ETL, while Hadoop and MongoDB MapReduce are open-source tools to implement the MapReduce pattern. The main difference between all these computational patterns resides in their expected performance and run frequency. ETL jobs, for instance, process huge amounts of data in one shot, and they typically run only a few times per day [87]. By contrast, applications implementing the event-sourcing pattern are expected to react instantaneously to a new event in the queue, and the same holds for all stream-like processing platforms. We will now discuss in some more detail these differences by presenting a common categorization of data processing platforms.

1.2.1 Offline data processing

We may define a data processing activity as *offline* whenever the following conditions hold:

- The activity ingests an important amount of data, often measured in Gigabytes or Terabytes. Indeed, an important measure of the performance of an offline data processing pipeline is how well it scales compared to the magnitude of the input [87].
- Input data is usually not necessarily "new" from a computational point of view, and it is generated by some other actor that is not necessarily related to the data processing pipeline. A few examples are user session metrics, software telemetry, or even other data processing activities. Input data is usually retrieved from data storage facilities like databases and data warehouses.
- No actor (i.e. human or software) is actively waiting for the results of the computation. The activity is scheduled to run at some point (e.g. once a day overnight) in a *best effort* fashion. In case data is needed at a specific point in time, the previous execution result should be used if the newest is not ready yet.

Data may be structured (e.g. database entries) or unstructured (e.g. schema-less JSON documents, entries from a graph database, text files), and the results of the computation are usually stored back in some data storage facility where they can be easily fetched by downstream users. While the amount of data generated in small companies does not usually justify the complexity of an offline data processing pipeline, we may find countless applications of this model in medium and big companies [87]. For example, Google reported using this paradigm for very heterogeneous tasks, like large-scale machine learning problems, producing reports for services like Google Trends, processing web pages to improve the capabilities of software products, and computations on satellite imagery data [34]. Similar reports have been produced also by other important digital companies like Yahoo Inc. [55], and it is rather common for services providing some sort of recommendation system nowadays to rely at least partially on offline data processing [22].

1.2.2 Online data processing

As opposed to offline data processing, the online model aims to process data messages coming from input sources as soon as they are received. For this reason the tasks are usually short-lived, e.g. simple computations or aggregations of several

fields in a database entry [87]. We may divide online data processing further into two additional categories depending on the properties of the computation:

- *Stateless* data processing comprises workers that are not allowed to retain any state from one call to another. They typically implement pure functions, meaning that each run has no side effects and is fully deterministic. While this is very convenient from a computational point of view (workers may even run in Kubernetes clusters for additional fault tolerance [125]) there are simple use cases that do not fit into the pure function model, e.g. computing the running average of observed values.
- *Stateful* data processing, by contrast, allows retaining an intermediate state, thus extending the scope of problems that may be addressed. However, this makes the execution environment of the workers less flexible. Depending on the service level agreement (SLA), end users may state that sources shall be independent (e.g. they should not influence each other for what concerns worker state) or require fault-tolerance guarantees. A short summary of combinations with possible solutions is provided in Table 2. In addition to redundancy, a common solution to achieve some level of fault tolerance is including in the loop a fast key-value store accessible to all workers (e.g. Redis [24]) to make sure newly spawned workers are able to pick up the work from a crashed sibling [87].

In some sense, any web service that is able to receive inputs and provide outputs in response could be considered a sort of online data processing platform. However, scalability problems and robustness requirements emerge as soon as the processing enters the critical path of the business, and dedicated logic is needed to extract, store, and treat data coming from input sources. These are the main

Table 2: Example solutions for several service level agreement combinations for stateful online data processing platforms. By *redundant* we mean that a worker is spawned multiple times, only one copy being active. The inactive siblings shall "shadow" the active worker, mimicking all its actions (e.g. retrieving data from sources) without emitting any value [87].

Independent sources	Fault-tolerance	Example solution
✓	×	Fixed worker → source
×	×	Single worker
✓	✓	Fixed redundant source → worker
×	✓	Single redundant source → worker

drivers of the need for dedicated online data processing platforms like the *Unified Controls Acquisition and Processing platform* (UCAP) [29], which we are going to talk about in more detail in Section 1.4. Fermilab, which has comparable computational needs as CERN, developed a similar framework for online data processing named *art* [50, 60]. All such frameworks embed specific logic for the use cases they aim to solve, and are structured according to an architecture driven by the expected scalability needs and data volume manipulated. However, they also commonly share similar ideas and patterns [74]; for example, the software facilities to handle problematic input sources [163], or buffering techniques [47].

1.2.3 Data processing and cloud computing

Cloud computing has lately become ubiquitous in the software engineering realm, especially thanks to the advent of relatively inexpensive providers with countless solutions for all possible use cases. Providers tend to provide solutions on varying degrees of abstraction, going from simple Linux instances with limited software installations, to fully managed deep learning pipelines with extensive tooling and monitoring [98]. This new paradigm has become so important that is difficult nowadays to avoid at least considering cloud-related aspects when discussing software architecture. For this reason, in the following paragraphs we will provide a very brief overview of the most important concepts of cloud computing; then, we will discuss typical cloud-based solutions for data processing in the cloud.

Cloud computing

As we stated earlier, one of the most important trends we have witnessed in software engineering during the last decade is the transition towards cloud-based applications [85]. Cloud computing is typically defined as the act of providing some kind of service with a varying degree of externalized management [126]. The most basic level of management consists in what is commonly called *Infrastructure-as-a-Service* (IaaS), which generally includes hardware, virtualization software, and internet connectivity. A well-known example of IaaS is Amazon Web Services (AWS) EC2 [81]. Climbing up the abstraction ladder, the next level is the so-called *Platform-as-a-Service* (PaaS). In addition to the offering provided by IaaS, PaaS provides additional tooling to deploy and run user applications. Again, one of the most representative implementations of the PaaS model is AWS Beanstalk, which provides smooth deployment, runtime and scalability for web services [98]. Finally, the last level of abstraction is occupied by *Software-as-a-Service* (SaaS). This is the level which the general public is most familiar with, since it comprises well-known online applications like Gmail.

Data processing in the cloud

Due to the large number of available solutions, it is not a surprise to find many facilities for data processing among the offerings of many cloud providers. Several examples for the use case of offline data processing, of which we talked about in Section 1.2.1, are AWS Glue and AWS Redshift [98]. These services facilitate establishing ETLs or periodic queries on large datasets, thus removing the need for dedicated on-site servers. On the other hand, a whole new paradigm was introduced to address the use case of online data processing, namely *Function-as-a-Service* (FaaS). The idea is to define the logic to be run on incoming inputs (e.g. a user-defined function) and to let the cloud provider figure out scalability needs (i.e. the number of workers required), worker lifecycle, code loading, as well as server provisioning. AWS Lambda and Apache OpenWhisk are two among many popular solutions. The FaaS paradigm is sometimes referred to as *serverless computing*; indeed, it might appear from outside that these functions are not running on a real server [48]. However, the name is typically abused and misunderstood, and might hide the complexity behind the abstraction layer provided by the cloud provider.

Even though the FaaS paradigm simplifies substantially the development and deployment of online data processing platforms, there are important distinctions to be made. First of all, functions running in such facilities are typically expected not to be stateful. This is because the provider expects to be able to spawn multiple identical workers, which is often referred to as *horizontal scaling* [87], and kill those that are not needed anymore without notice. Solutions addressing this problem are being explored [137], but they usually determine important performance penalties and reduce flexibility. Another important distinction between FaaS providers and *on-premise* online data processing platforms is the expected response time. Due to the internal scheduling mechanism of FaaS platforms, the two utilization extremes for a specific serverless function, namely *overused* or *underused* are typically problematic for the observed throughput [131]. In the first case, incoming inputs will be queued if all workers are busy, until one of them becomes free, or the platform manages to spawn a new one, which is typically not immediate; in the latter case, researchers observed an interesting *ripple* effect in the response time, which is most likely due to the fact that FaaS platforms tend to downscale active workers in such situations.

1.3 Language interoperability

In the present era of pervasive technology and widespread digital literacy, we have witnessed the proliferation of programming languages [73]. The factors that

determine whether a programming language will spread widely in the software development community or not are somewhat unknown *a priori* [129]. We would expect to observe the emergence of the most comfortable and smart languages, in particular those that provide compile-time error detection for nontrivial development mistakes. For instance, the GNU Fortran Compiler aborts the compilation whenever it detects a type or shape mismatch in the context of an algebraic operation. Nonetheless, *Fortran* is becoming a niche language, due to the widespread agreement on its complexity, in favor of popular alternatives like Python's *NumPy*, which as of early 2024 do not offer the same development *Quality of Life* (QoL) [65]. By contrast, *Javascript*, whose confusing design and idiosyncrasies are frequently criticized by practitioners [107], is one of the most used programming languages in the industry [73]. We may hypothesize that a language being successful is a matter of luck, tooling, and rapidity in occupying a space.

In such a context of extreme fragmentation, developers of frameworks based on *user-defined functions* (UDF) need to take into account the preference of their users for what concerns the base programming languages of the platform. A perfectly crafted and highly optimized data processing platform may be doomed to oblivion if it is perceived as hard to interact with. This is especially important if take into account the differences between mainstream languages for enterprise backend software development and the realm of scientific software. The former has been largely dominated by Java, C# and C++ in the last few decades [122]; conversely, the latter is much more fluid and open to changes. Python is still the most popular choice due to its friendly syntax and humongous open-source environment [138]; however, we are currently observing interesting alternatives like Julia taking some room in the field [116]. Requiring users to provide UDF code written using a single programming language is definitely the best choice in terms of performance and simplicity of integration with the platform codebase. In Java, for example, one could easily leverage a dynamic class loader (e.g. OSGi [147]) to load user classes at runtime; then, their lifecycle could be managed via reflection, provided that the performance penalty is acceptable [152]. This is indeed the approach taken by UCAP for Java UDFs [29]. However, enforcing the usage of a specific programming language may be largely suboptimal from a usability point of view. Such a design choice forces users – who should only be concerned about the logic for their use cases – to rely on a tool which they may not be familiar with. An immediate consequence is the observation of an increased rate of human error in the logic embedded in UDFs. This is a very important problem, since such software is rarely peer-reviewed. One of the most critical hazards is posed by language-specific type system behaviors, as evidenced by the *1991 Patriot Missile Failure* incident [166], in particular if the developer is used to a very permissive

language like Python. Thus, language interoperability is a need that can hardly be ignored during the development of data processing frameworks.

There are a few technologies on the market that we may leverage to achieve language interoperability, with varying degrees of coupling. We will start our short discussion by discussing our definition of *coupling* in the context of language interoperability, since it is one of the most important factors to take into account in order to choose the most appropriate technology. Then, we will mention a few practical alternatives to achieve language interoperability.

1.3.1 Coupling

Considering two communicating "snippets" written respectively using the programming languages A and B, we may define "coupling" as the amount of interference between the two pieces of code at the time of writing them. In other words, coupling is the degree of knowledge that one party has of the other, and vice versa. For example, interoperation via HTTP calls (e.g. in the context of a microservices architecture) requires only the knowledge of the host and the port the service is listening on, irrespective of the language or the technology. Stronger coupling usually means that all parties must be compiled and deployed together, even if only one has changed. While this is irrelevant for small codebases, it is a factor to take into account for medium or large projects; developers will experience more idle time while debugging and testing new code, thus making the feedback loop significantly longer [95]. By contrast, weak coupling results in much smoother development workflows, at the cost of more heterogeneous deployment practices. As an example, the reader may refer for instance to the differences that hold between deploying Java and Python applications to production [43, 71]. Besides the ease of development, a second consequence of weak coupling is increased communication latency. This is mainly due to the fact that keeping all parts of the codebase close – namely compiling and deploying them together – typically reduces the communication overhead to the minimum. In such cases, all parties are usually running within the same process, therefore communication happens via shared memory.

1.3.2 Interoperability techniques

In this section, we are going to discuss a few options we may leverage to achieve language interoperability. We will present all points in decreasing order of coupling.

Compile-time interoperation

Language interoperation may be defined as *compile-time* whenever the two languages are processed into compiled results (e.g. JVM bytecode) which are compatible to some extent. This is the case, for instance, for C and C++ [140], and JVM languages like Java, Scala, and Kotlin [79]. Such interoperability technique provides interesting benefits in terms of development quality of life (QoL): IDEs typically provide instantaneously synchronized cross-language auto-completion, and most debuggers are able to seamlessly jump from one language to another. It is evident that this technique maximizes the coupling. Projects requiring low-latency communication may leverage shared memory to achieve the maximum communication speed, but the downsides mentioned in Section 1.3.1 should be kept in mind in the design phase.

Bindings interoperation

Bindings are loosely defined as glue code that enables one language to call functions from another. Conceptually, this technique is quite different from the one we talked about in the previous paragraph, since the "provider" (i.e. the party which provides functions to be called) and the "receiver" are not compiled together. Indeed, bindings may be generated only after the provider is compiled. Bindings are usually generated using third-party libraries, since built-in mechanics are typically too low-level and error-prone [155]. These libraries take care of assessing the well-posedness of the links, and of generating the glue code, presented as a user-friendly interface. Some examples are the libraries for bindings Python to C (i.e. calling C and C++ functions from Python code) like *ctypes* (Python standard library) and *pybind11* [76], or those to bind Python to Fortran, like *f2py* [65]. In Listing 1 we provided an example of C++/Python bindings using pybind11. This case in particular has been extremely popular in the last decade and is most likely one of the strongest factors in the important adoption of Python which we have witnessed in the scientific computing environment [138]. Indeed, bindings allow leveraging low-level functions, thus enabling to harvest the performance speed-up without giving up the benefits of high-level languages. This is the whole idea behind NumPy [65], Python's most popular linear algebra library, which in the last 20 years has become the *de-facto* glue interface across the whole scientific Python ecosystem. In practical terms, this technique is extremely useful and powerful, at the cost of an important maintenance cost. Developers need to take care of the additional code to manage the bindings, which may eventually become convoluted and rather vast in the case of bindings towards large low-level libraries. From a performance point of view the latency is comparable to

```
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example_module, mod) {
    mod.def("add", &add, "Adds two numbers");
}

...

>>> import example_module
>>> example_module.add(1, 2)
```

Listing 1: Example of binding a simple C++ function for summing two integers from Python with pybind11.

the technique which we discussed in the previous paragraph; however, the cost of bindings is not fully negligible in low-latency contexts, especially as the number of inter-language jumps is not constant [101].

Dynamic interoperation

In addition to the options we listed above, we may consider a more "dynamic" way to establish cross-language communication by relying on some specific third-party libraries. We list some of them and explain very briefly their goal. We will focus on Java/Python interoperability as it is the main goal of this work. However, similar tools exist for different pairs of languages.

- *JPype* [102] is an open-source library whose goal is to make Java functions and objects reachable by the Python interpreter. The JVM is started from within the Python interpreter, therefore the two entities share the same memory, and they interface at a native level. This tool is extremely valuable for generating Python bindings towards critical libraries written in Java, and is already extensively used at CERN [71].

- *Jython* [115] presents itself as a full implementation of Python in Java. It enables executing Python scripts (stored as Java strings) within Java code, thus enabling the opposite path with respect to JPype. Due to its implementation strategy, users are fully dependent on the Jython development team regarding compatibility with newer Python versions; indeed this tool is known to be lagging quite significantly behind with respect to the present state of Python. However, since Python code is fully mapped to Java and

thus executed internally to the JVM – as opposed to starting a Python interpreter in a separate process – the interoperation is expected to be much faster than all other alternatives. This aspect makes it especially suitable to manage simple user-defined functions without complex external dependencies. This is for example the use case of Apache Pig [144].

- *GraalVM* is developed by Oracle as part of the OpenJDK Graal project, and it spans a wide range of revolutionary additions to the JVM-based ecosystem [133]. Among other impactful, GraalVM introduces *Truffle* and *Sulong*, which allow binding a wide range of high-level languages ranging from Python and R – leveraging intermediate compilation to Abstract Syntax Tree (AST) – to more low-level alternatives like C, C++, and Rust which can be compiled to LLVM bytecode. The effort aims at bringing to the community a fully interoperable ecosystem of heterogeneous programming languages, thus opening scenarios that have never been possible before. At the moment the project is still in the early stages, although initial benchmarks have been extremely promising [19]. Nonetheless, important problems like dependency management are yet to be addressed for productional use.

The outcome of this brief tooling overview is that the market offers many options to achieve dynamic language interoperability; however, this is often limited to rather simple use cases. Executing simple user-defined functions is a fully supported use case, albeit with vague scaling guarantees and important distinctions to be made. Most tools do not fully support third-party libraries, and in general desirable features like telemetry and tracing are quite hard to achieve at the present stage. Nonetheless, the progress in the last decade has been impressive, and the efforts of big players like Oracle (which is behind GraalVM) will certainly be beneficial to the process.

External service interoperation

The language interoperation technique which provides the least possible coupling is the collaboration between different services. This is for example base of the microservices idea [103]. Each service may be written in a different language depending on available tooling to achieve the intended goal and to the knowledge of the team that takes care of the development. Services are then expected to communicate with each other by using specific endpoints according to their Application Programming Interface (API), which declares which functionalities are available to the outside world and the usage policy. A well-defined API specifies univocally acceptable input, outputs, and possible errors. Communication usually happens over TCP, but many alternatives exist in the market, as we are going

to discuss in Section 2. In case exchanging data between services is needed, we might want to attach a payload to the messages. To this end, the API should also specify a standardized format, such that both ends are able to decode and parse the messages into an in-memory data structure. We are going to examine this topic with more details in Section 2.3. The fact that each service knows as little as possible about its peers – only the public interface – makes for very clean and maintainable code, which is only possible due to the extremely low coupling that characterizes this technique. However, this type of communication is typically the slowest among those that we talk about in this section due to network hops.

1.4 UCAP framework

Since the subject of this work is the optimization of language interoperability in the *Unified Controls Acquisition and Processing* (UCAP), in the present section we are going to provide some general information about the project. This information is going to be useful both to understand the scale and the importance of the project in the CERN control system, and to identify the practical consequences of architectural and technology choices that were taken during the development. We are also going to discuss the kind of inputs and outputs signals UCAP deals with, as well as the most relevant use cases it covers.

1.4.1 Purpose

UCAP is a framework that combines data acquisition, processing/transformation, and publishing in a single self-service software. By self-service, we mean that in productional contexts users are able to fully manage their UCAP instances thanks to the tools which are provided by the UCAP team, without the need for the intervention of an expert. In the following paragraphs, we will briefly describe each layer in order to provide a high-level view of the product.

Data Acquisition

Data Acquisition (DAQ) is the act of acquiring data from one or more sources and preparing a "message" that contains only the expected inputs from each source. For example, a common policy is to prepare messages containing one single input from each source. This is shown in Figure 2. Data acquisition is formed by two interconnected phases:

1. *Data collection*, pulling inputs from each data source and saving them in a local buffer;

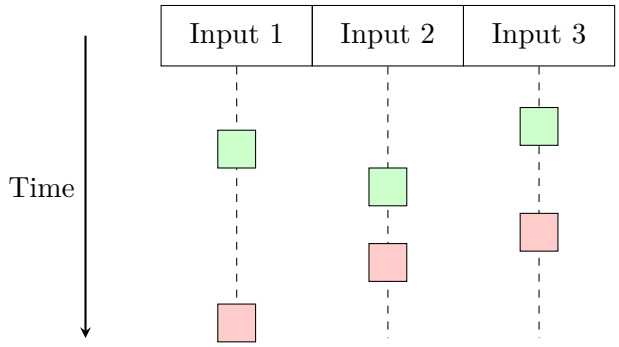

Figure 2: Typical policy for data acquisition. Inputs from three sources are grouped to form complete messages of three items each. Each message is represented by a different color.

2. *Data preparation*, packing a message according to a given policy.

This nomenclature is not backed by any literature, as data collection and preparation are rather common practices that occur in many different fields.

Data collection in UCAP relies on CERN internal libraries (RDA3 [90] and JAPC [6]). These libraries enable subscribing to various data sources to get notified of any newly emitted value, as well as providing dedicated callbacks to be run when errors are detected. RDA3 may be considered the standard software at CERN for data collection, with JAPC being the go-to solution for Java applications.

Data preparation is one of the hardest conceptual problems to be treated along the development of an online data processing platform, due to the massive amount of problems and special cases one has to deal with in order to provide a complete solution [87]. Some examples are windowing, triggering, discarding policies, and out-of-order data, and all of these topics may come in different flavors, or with different parameters. Most importantly, each policy can be combined depending on user needs, in order to build the data acquisition pipeline that best fits their specific use case. This makes the practical implementation not trivial as well: a first idea would be leveraging the *Decorator* pattern [80]. However, users will be well exposed to such code, therefore it is ideal to provide clear, unique, and unchanging pathways. Open-source implementations exist [2], as well as open-source projects based on them to take inspiration from like *Apache Beam* [70]. However, it was decided in the early days of the development of UCAP to develop an in-house solution to best fit CERN-specific requirements. This layer of UCAP is commonly referred to as *Event Builder*.

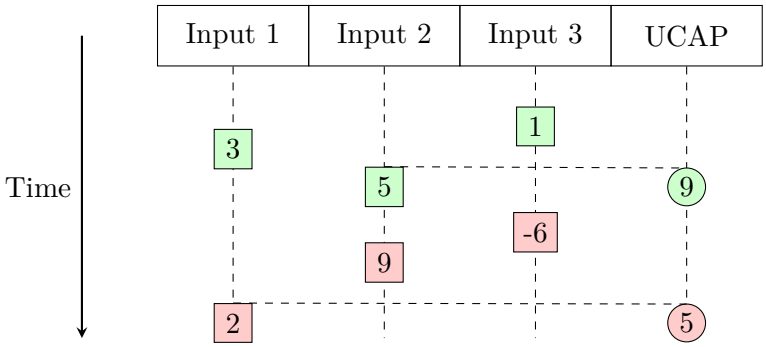

Figure 3: UCAP transformation layer along with the acquisition layer. The user-defined function implemented in UCAP is a summation of all inputs. Processed values are represented by circles.

Transformation

The transformation layer allows users to provide custom logic to process acquired inputs. This logic is commonly referred to as *user-defined function* (UDF) in the literature, however in the UCAP environment it is more frequently called *converter*. In order to employ a more generic language, we are going to use the former term for the rest of the present document. As the name suggests, a user-defined function could perform any kind of transformation on the input, as long as it respects some boundaries on the structure (e.g. one-to-one mapping, as opposed to one-to-many mapping). Figure 3 depicts a simple schema that shows the behavior of the data transformation layer. UCAP supports both Java and Python user-defined functions. The former language was supported since the first release and is somewhat trivial since Java is the main language employed for the development of UCAP. On the contrary, Python support was a strong requirement for more recent updates, which was driven by the increasing importance it acquired in the last decade in scientific computing, as we highlighted in Section 1.3.

Publication

The publication layer in UCAP adheres to the widespread standard declared by RDA3 applications. Each node (see Section 1.4.2) acts as a server that accepts direct queries (i.e. "get" requests) and subscriptions. Whenever a new value is produced by a transformation, it is forwarded to all listening applications instantaneously. UCAP subscribers may be any kind of software running JAPC/RDA3, long-term storage, and querying engines like NXCALS [163], as well as UCAP nodes.

1.4.2 Deployment architecture

In the present section, we are going to give some very high-level details about the structure of UCAP as a software. We will restrict the presentation to a subset which will ease the discussion for the rest of the work. We will start by describing the "content" of a single UCAP node, which is the deployment unit of a single UCAP instance. Then, we will provide some general numbers to let the reader understand the scale of the software and its importance in the environment of the CERN control system.

UCAP node

A UCAP node, in its simplest form, is a Spring-based Java process running in a JVM. The following are the most important components it contains:

- User-facing web interface: HTTP endpoints for all user interactions;
- Data collection and input source monitoring facilities;
- Code to handle all cases for *event building* (see Section 1.4.1);
- Internal buffers for unprocessed inputs;
- Factories, repositories, and executors for Java-based user-defined functions, as well as dynamic code loading facilities;
- Components for language interoperability with Python;
- Server transformation output server for publication.

Such Java processes are deployed on bare-metal machines, namely their lifecycle is fully managed by the UCAP team. A node is usually dedicated to a given experiment/accelerator (e.g. `UCAP-NODE-ISOLDE`) or to a specific application (e.g. `UCAP-NODE-TOMOGRAPHY-PSB`). Each can run zero or more transformations, which are registered by users as required via the web interface. Due to this kind of architecture, UCAP falls in the bucket of *distributed online data processing platforms* [87]. The node JVM should keep running continuously, as any crash implies the loss of all unprocessed messages in the internal buffer, and of all messages emitted while the node is down. Since UCAP is considered a critical software in the context of the CERN control system, this is a very unfortunate situation that should not happen in general.

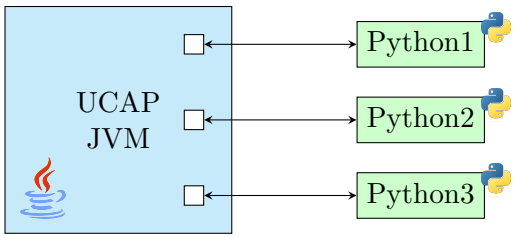

Figure 4: High-level architecture schema of UCAP-Python integration. Python processes are represented by green nodes, while the main UCAP process is represented by a single cyan node. The communication channel towards the internal handler is bidirectional.

Python transformations

Python transformations (i.e. user-defined functions written in Python) are hosted in dedicated Python processes. The lifecycle of such processes is fully controlled by the main UCAP process, which may start and stop them as needed. Each Python process hosts:

- A tiny server that replies to requests generated by the main process;
- The logic that implements the user-defined function.

Figure 4 depicts a simple schema that summarizes the architecture. Each Python transformation is represented internally in the Java process as a "stub" which abstracts away the implementation details like transport and serialization/deserialization. This is represented in the image as a small white square linked by a bidirectional channel to a specific Python process.

UCAP adoption statistics

At the time of writing the UCAP team manages about 250 nodes. In Figure 5 we visualize the growth over time of the number of nodes. As we mentioned earlier, each node is dedicated to a particular team or experiment. Notably, not all of them are running the latest release of the UCAP framework. While it is not recommended to keep many versions of the same software running at the same time, it is difficult to upgrade since shutting down a node is a disruptive operation that must be negotiated with end users.

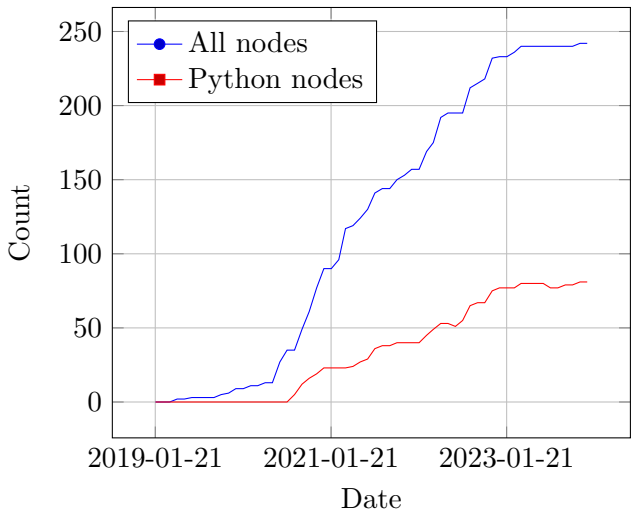

Figure 5: UCAP nodes count growth over time. The figure shows both the total count of nodes (blue) and the count of Python nodes (red).

1.4.3 Input/output structure

UCAP inputs and outputs are those that are supported by the underlying transport layer, which is delegated to RDA3/JAPC [29]. Typically, values are either proper data or errors. Due to the intrinsically distributed architecture of CERN software, errors must be propagated to let end users understand why no value was published at a given moment. It is up to the code developer to decide how to react to the occurrence of an error, either by republishing it or by conceiving some logic to compute a value nevertheless. UCAP user-defined functions follow the same approach. Moreover, values that contain data could be simple (e.g. an integer, or a floating-point array) or composite, namely sets of key-value pairs (similar to Java's `Map<String, Object>`). The list of accepted data types is listed in Table 3. In addition to standard Java types, JAPC introduces `Enum`, `EnumSet`, `DiscreteFunction`, `DiscreteFunctionList`. The first two are respectively on-wire formats for Java's enums and `Set<Enum>`, which require special treatment because enums are not trivially serializable from one JVM to another due to classpath problems. The last two are respectively a pair of equally sized `double` arrays, and the serializable equivalent of `DiscreteFunction[]`. These types were introduced to support more variegated use cases, however they are not commonly used. The UCAP-Python integration does not officially support them due to a lack of requirements by end users since the first release (late 2020). In

Table 3: Types supported by JAPC. The number of bits is unspecified for several types because it either depends on the chosen on-wire representation (`Enum`), or it is driven by the number of elements in the collection (`EnumSet`, `DiscreteFunction`, `DiscreteFunctionList`).

Data type	Number of bits	Base type
`bool`	1	n/a
`byte`	8	n/a
`short`	16	n/a
`int`	32	n/a
`long`	64	n/a
`float`	32	n/a
`double`	64	n/a
`Enum`	n/a	n/a
`EnumSet`	n/a	`Enum`
`DiscreteFunction`	n/a	`double`
`DiscreteFunctionList`	n/a	`DiscreteFunction`

addition to the types mentioned in Table 3, all of them are supported as $1-$, $2-$, and $N-$dimensional arrays.

1.4.4 Applications

As we mentioned earlier, the UCAP framework has been widely adopted by the CERN accelerator controls community. Its flexibility enables applications in extremely wide contexts, with varying degrees of complexity. The typical use case is subscribing to a set of data and subsequently applying some user-defined transformation, possibly triggering actuator cycles conditionally on the current value of some real-world quantity. This flexible model encourages using the framework as the foundation of more high-level applications, whose low-level needs are exactly those covered by UCAP. In this paragraph, we are going to present several outstanding applications that have been based on UCAP during the last few years.

Linac4 autopilot

A dedicated instance of UCAP, leveraging the Python integration, is deployed to function as the foundation of the *LN4 Autopilot*. This software serves as a fully automatized monitoring and control framework to drive the Linac4 ion source [25]. It leverages UCAP features, in particular the possibility to run Python

user code, to control the value of multiple parameters to guarantee the proper functioning of the machine [117]. Moreover, it simplifies real-time logging of relevant measurements to expert GUIs and NXCALS [163]. This software was very well received by the operators community in the CERN control system, thus another instance was deployed to provide the same functionalities for *Linac3*.

GeOFF

In addition to the processing of user-defined functions, there have been recent developments to leverage UCAP as a stateful FaaS server for online optimization problems. The goal of these problems is to leverage a control, typically imposed via some kind of real-world actuator, to optimize some desired measurement, which is continuously measured and fed back as input to the UCAP node. In the context of the CERN accelerator complex, the controlled output to be controlled may be, for example, the intensity of some noise-related frequencies in the FFT of a measurement. The effort resulted in the release of *GeOFF* [70, 154]. UCAP is the foundation of such a product, which simplified significantly the integration of the software with the control system.

Software Interlock System (v2)

The *Software Interlock System* (SIS) is a critical application in the CERN control system. Instances of the software continuously monitors relevant parameters in accelerators, and triggers a *beam dump* (i.e. the beam is stopped for safety reasons) when a security threat is detected [162]. Security conditions are implemented as lengthy boolean conditions involving multiple parameters, structured as deep condition trees. The software has important robustness requirements: having an automatized security system is an extremely important need for the CERN accelerator complex. The first version of SIS has been running for almost 20 years as of 2024, but it was perceived as difficult to maintain and interact with. For this reason, a complete renovation was initiated to leverage a more flexible architecture and up-to-date technology. UCAP was selected as the foundation for SISv2, serving as both the acquisition and computational layer. The computation of the nodes of condition trees implementing SIS security conditions is delegated to UCAP Java user-defined functions, running in dedicated UCAP nodes. This application of UCAP testifies its importance in the control system, due to the degree of criticality and availability requirements of SIS.

1.5 Goal and motivation

The present implementation of the UCAP-Python integration, which we have briefly outlined in Section 1.4, allowed UCAP to gather a significant number of users in the Python community at CERN. Such an important addition, which amounts for a substantial part of the UCAP framework in terms of lines of code, and is by far its most complex software component concerning cognitive and architectural complexity, was implemented in 2020 with the goal of providing a stable, powerful, and robust API. The high-level objective was to allow implementing UCAP user-defined functions in Python, supporting all value types which we have mentioned in Table 3 without any specific scalability requirements. The goal was achieved as testified by the encouraging numbers which we have presented a few paragraphs above. However, the steady increase in the amount of data produced has shown the limitations of the approach adopted at the time. In 2017 the aggregated amount of data collected at CERN passed the line of 200 Petabytes, mostly driven by the impressive stream of information provided by LHC [53]. Computational needs are shifting from simple CPU-bound use cases towards more bulky memory-bound problems, even for online data processing platforms. This is driven by the increased amount of data produced daily in the CERN accelerator complex, as well as by new data processing paradigms in the context of machine learning.

In order to enable treating such cases, important architectural and technical changes are required for what concerns the UCAP-Python integration. Preliminary benchmarks, whose numbers will be reported in Section 3.3, have shown the inadequacy of the present solution with respect to emerging use cases involving, for example, large numeric arrays. The impressive processing speed achieved by Python user-defined functions, thanks to bindings towards high-performance languages like C++ (see Section 1.3), is annihilated in UCAP by the slowness of internal components. An increasing number of stakeholders has been demanding a solution to the problem, in order to allow harvesting the large scientific Python ecosystem for an extended set of use cases. Among other reasons, such demands are especially due to the gradual adoption of the UCAP framework as a critical software in the CERN control system. Such an important work package has been delayed due to its extended scope and complexity, and has now become a blocker for many users.

In the present work, we are going to present the theoretical and practical aspects of the journey toward the production-ready implementation of a high-performance UCAP-Python integration. We are going to review the available technologies on the market, as well as the ideas that allowed us to improve by orders of magnitude the response time of UCAP-Python user-defined functions.

Our claims are supported by an extensive retrospective analysis, based on *ad-hoc* software metrics, which will testify the quality of our additions. Moreover, a gradual migration via blue-green deployment [87] was conducted as the work progressed, thus providing important feedback in the form of both performance measurements and software correctness.

2 Inter-process communication

Inter-process communication (IPC) is the act of exchanging information within two processes. The processes could be, for example, instances of the same application, sharing data for caching purposes [87]; or else, they could implement a master/slave communication pattern, the latter receiving input data to be processed from the former; finally, they could be peers cooperating to solve a large-scale problem treated with domain decomposition [62]. Inter-process communication is a critical topic in software engineering, as it enables treating problems on a much larger scale, leveraging different technologies to solve unique problems (see Section 1.3), and splitting software in specialized applications with a single high-level responsibility [103]. However, these benefits do not come for free. Indeed, introducing this new degree of freedom increases significantly brings in a plethora of problems and considerations to be done. In this section, we are going to present a brief exploration of these topics, in order to provide an overview of the matter for what concerns the ideas and results which we are going to present in Section 3.3. In particular, we will first focus on communication strategy, namely on the high-level orchestration pattern for inter-process communication; then, we will examine practical ways to achieve IPC; finally, we will discuss the data serialization format, a critical topic in the context of high-performance IPC.

2.1 Communication strategy

We are going to start our theoretical discussion with an overview of communication strategies. As we briefly mentioned in the beginning of Section 2, a communication strategy is a certain pattern that is adopted to organize information exchange in the context of inter-process communication. Such a pattern must specify when the communication should happen (in terms of time elapsed, or triggering events), the initiator, namely the process that initiates the communication, and the type of communication established among the parties (blocking or asynchronous).

We propose a basic example in the context of the *producer/consumer* problem, which we will deepen in the following paragraphs. We consider two communicating processes A and B, the former produces some data which shall be acquired by the latter. Data is produced by A (the producer) in packages at random time intervals. A possible approach is to let the producer process initiate a new communication channel every time a new data package is available, such that B (the consumer) receives it as soon as possible. This solution would work in most cases; however, one should be careful about the expected data production frequency. If data is produced with high frequency, many communication channels could be

opened in overlapping time windows. If too many are open at the same time, depending on the transport implementation, the processes may experience a variety of problems, like running out of file descriptors [87], or in general being too overloaded to operate in normal conditions. To address these issues, buffering produced values and reducing the communication frequency help in mitigating the issue. Alternatively, we could keep a long-lived communication channel for the whole lifetime of the application. Still, the channel might file at the same point, and appropriate strategies to recover it and avoid data loss should be put in place.

As evidenced by this simple example, there are a number of variables to keep into account while designing a communication strategy. Optimizing the performance should be among the most important concerns, keeping in mind that the strategy will have a sensible impact on the software architecture and its reliability. Depending on the problem that we intend to tackle, we might have a plethora of solutions, each with its particular collection of pros and cons, like the producer/-consumer problem; alternatively, we might encounter problems whose structure dictates very specific communication patterns. An example of the latter is the implementation of the distributed *Jacobi algorithm* for the approximated solution of the diffusion equation:

$$\frac{\partial \Phi}{\partial t} = \Delta \Phi.$$

On an appropriately discretized 2D domain, leveraging finite differences to replace differential operators we obtain an approximated iterative solution for the equation. For each iteration of the method, the correction to the partial value of $\Phi(x_i, y_i)$ involves the values of $\Phi(x_{i+1}, y_i)$, $\Phi(x_{i-1}, y_i)$, $\Phi(x_i, y_{i+1})$, $\Phi(x_i, y_{i-1})$ obtained after the previous iteration [62]. This translates into the need, due to the application of domain decomposition, of exchanging an *halo* (i.e. the values of Φ on the border of the subdomains) before each iteration. Therefore, this problem does not admit the application of buffering or similar solutions to ameliorate the communication impact on performance: the algorithm cannot proceed to the next step if *halos* are not exchanged. In this case, the communication pattern is imposed by the structure of the problem, and the strategy adopted for domain decomposition.

In the following paragraphs, we are going to treat several communication strategies that will be of interest in Section 3.3. In particular, we will focus on the aforementioned *producer/consumer* problem. This choice is mostly driven by two factors. The first is that the problem treated in the present work, which we discussed in Section 1.5, is an instance of a stateful producer/consumer problem with feedback [29]. The second is that a significant amount of problems in data

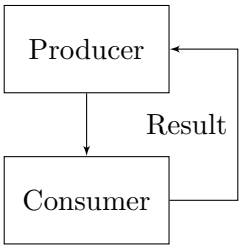

Figure 6: Schema of the feedback variant of the *producer/consumer* problem.

processing, especially online, can be reduced to some variant of the producer/consumer problem [86]; therefore, we believe the work presented in this document is general enough to be of interest to a large fraction of practitioners. We are going to start the discussion with a more precise definition of the problem; then, we will introduce several possible strategies to tackle it in practice, making an important distinction between the *shared-memory* approach and the *distributed* one. Finally, we will introduce some additional topics in the context of communication. These topics are not necessarily related to the producer/consumer problem; however, they are definitely of interest for the optimization of communicating processes.

2.1.1 Producer/consumer queue

As we said earlier *producer/consumer* queue problem recurs quite often in distributed computing, since an important number of systems are based on one of its variants [78, 99, 164] like single/multi-producer, single/multi-consumer. Some practical examples we may encounter in the wilderness are:

- User-defined function remote execution: the producer is embodied by the triggering process, which requires a remote actor (the consumer) to execute a function multiple times, every time a new input package is ready;
- The MapReduce paradigm [34], which we have mentioned in Section 1, is a multi-producer/multi-consumer variant of the problem.

As the name suggests, the most basic variant of the producer/consumer problem involves two agents. They are respectively responsible for generating new work items, and processing them in order to produce some output. Variants may involve a diverse number of actors per category (i.e. multi-producer or multi-consumer); however, we are going to restrict our discussion to the 1/1 case, and we will only briefly consider the multi-consumer case at the end of the section. In the context of our project, we will also assume that the producer expects an output to be sent back for each input. The setting is represented schematically in Figure 6.

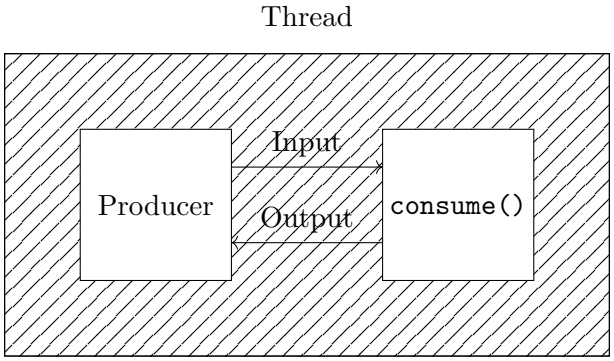

Figure 7: Single-threaded blocking solution of the produced/consumer problem. The function `consume()` embodies the consumer. The producer is fully blocked for the whole duration of the consumption.

2.1.2 Shared-memory approach

We are going to talk in more detail about the practical details of *shared memory* in Section 2.2; for now, we will only consider the implications from outside. In practice, shared memory allows multiple workers to coexist within the same running unit (e.g. a POSIX process). Most importantly, these workers share the same memory, meaning that they can access the same variables [62]. The practical realization of shared-memory workers is heterogeneous and depends on the specific tooling adopted. For example, Java provides support for both *platform threads* (which are bound to OS threads) [110], as well as *virtual threads*, (starting from Java 19) which do not occupy a full OS thread and are intended to be more lightweight thanks to an additional software layer which handles I/O blocking operations [111]. We are going to discuss the topic in more detail in a few paragraphs. For now, we are going to explore several solutions to the *producer/consumer* problem in a shared-memory context.

Single-threaded blocking approach

The simplest solution is to let producers and consumers live in the same thread, meaning that we have effectively only one worker. This is also the most robust approach, since we avoid all problems related to shared-memory (e.g. lock retention, race conditions [56]) which we will highlight in Section 2.2; however, the scalability provided by this method is very limited. The implementation is as simple as a function call, meaning that the producer is responsible for calling a function that does the job expected by the *consumer*. This approach is represented

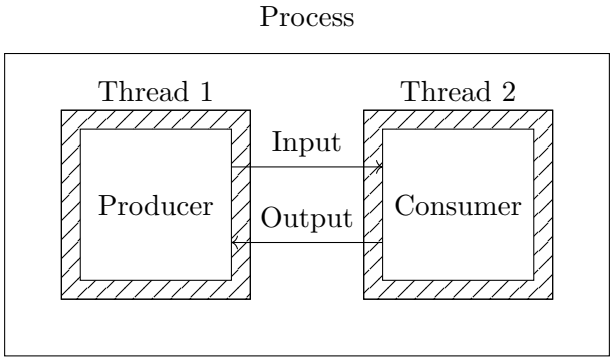

Figure 8: Multi-threaded synchronous solution of the produced/consumer problem. The setting is similar to that presented for the single-threaded case, with the only difference that the consumption happens in a different thread with respect to the production.

schematically in Figure 7.

Multi-threaded synchronous solution

If we introduce an additional thread and let the producer and consumer run on two different threads, we enter the realm of multi-threading. This topic will be treated in Section 2.2, for now we will only state that letting different threads interact is in general difficult and a typical source of bugs [56]. The situation is represented schematically Figure 8. Note that handing over data cannot be implemented by a simple function call as we showed for the blocking case, as function calls run within the calling thread by definition [110, 155]. The usual strategy is using a shared variable (i.e. a variable living within the shared memory) whose role is to keep data until it is needed. Note that such variable must satisfy certain criteria to avoid experiencing multi-threading related problems, which we will discuss in detail Section 2.2; for now, we will simply state that the variable must be *thread-safe*, meaning that it must be "safe" to read and write to it from different threads. Typical data structures employed to this end in Java are synchronized collections, as well as the `Atomic*` family, or standard objects decorated with the `volatile` keyword [110]. Since we expect communication to happen synchronously, the producer shall wait for a reply from the consumer carrying the output of the operation. It cannot do anything else in the meantime, even though it is in theory free to execute code. This kind of schema is usually adopted for robustness reasons, since errors are spotted immediately (i.e. if the consumer does not reply before a certain deadline) and there is no risk of processing data

out-of-order. However, this strategy is rarely adopted in practice. Synchronicity, as opposed to the blocking approach, is rarely a good choice. Moreover, the asynchronous approach which we will present in the next paragraphs can be coupled with some additional techniques to avoid the aforementioned problems. If one is not willing to explore the asynchronous multi-threading realm, due to the added code complexity it brings in, the blocking approach is usually a better choice.

Introduction on asynchronicity

Introducing asynchronicity opens the door to several new approaches to tackle the problem. Producers and consumers are allowed to live in different threads, like in the case that we examined in the previous paragraph. As we anticipated, threads must communicate via *thread-safe* shared variables in order to avoid memory access problems related to multi-threading. An important alternative, which we will also touch on in Section 2.2, is using a *future* (or a *promise* in some languages like JavaScript). From a high-level perspective, a future represents an ongoing operation. When the operation finishes, the thread holding the future may obtain a reference to the result. The developer retains an important degree of freedom in the choice of the variables to assist the multi-threaded communication, and this is usually the trickiest part of the design [56]; indeed, the data structures chosen for sharing data are usually the drivers of the observed performance of multi-threaded software [56,62]. We will start our overview of asynchronous approaches for the producer/consumer problem by discussing a simple design that involves future, focusing in particular on its shortcomings; then, we will present a buffer-based solution, which is quite close to what one could expect to find in most production-ready data-processing software [29, 52, 148].

Future-based asynchronous solution

The first solution that we present is based on the concept of *future* which we have just introduced. A future is typically an asynchronous primitive provided by the standard library of many high-level programming languages; it represents an ongoing computation running on a thread [56]. A future typically provides methods to inspect the state of the computation (e.g. `isDone() in Java` [110]), and get the output value if it is ready. This primitive looks indeed quite promising as a base for our solution to the asynchronous producer/consumer problem: we may require the consumer to provide a wrapping of the ongoing consumption as a future, which will be executed on the consumer thread. The producer is thus free to complete more work while it is waiting for the result of the consumption (e.g. to produce new work items); it shall query the future periodically to verify whether

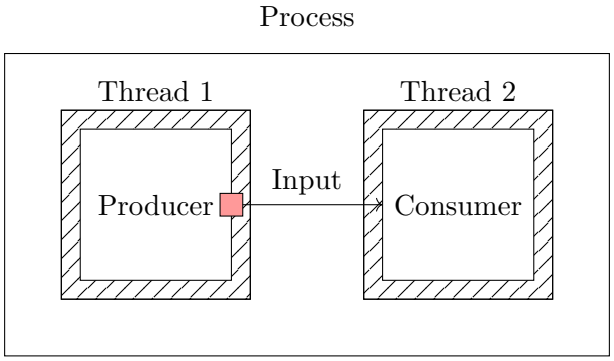

Figure 9: Multi-threaded buffer-based asynchronous solution of the producer/-consumer problem. The red square in the figure represents the future, which is returned immediately by the consumer as soon as the operation starts.

the output is ready, and react accordingly when the consumption is completed. The setup is represented in Figure 9. However, this exposes a rather important problem: the producer may produce new work items while the consumer is busy. How the system should deal with such an eventuality is not obvious: there are many solutions, each having its own shortcomings. Spawning new consumers is for now prohibited, as it uncovers even more problems like retry policies, deduplication, and output ordering. Another possibility is to force the producer to block by calling `get()` on the future which it receives from the consumer, in order to make sure no work items reach the system while it is busy; while this is going to work, it is a clear regression to the synchronous case. In the common case of cheap work item production and low-throughput consumption [87, 148] this system will always be working in synchronous mode, thus annihilating the benefits of asynchronous communication. Such a situation is a clear call for a return to the synchronous case.

Buffer-based solution

A buffering data structure may be introduced between the producer and the consumer, as shown in Figure 10, in order to fix the problems that we highlighted in the future-based solution. Every time the producer finalizes the preparation of a new work item, it shall append it to the buffer; similarly, every time the consumer is ready to take on new work, it will poll the buffer for the oldest unprocessed work item. Therefore, the interface we expect from the buffer should be capable of accepting new elements and extracting the oldest one. We summarize it in Listing 2. In computer science jargon, such a data structure is commonly said to adhere to

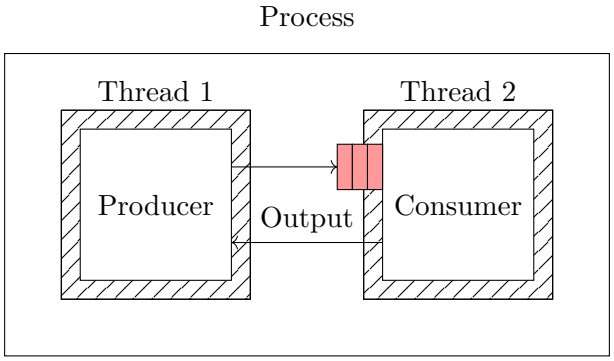

Figure 10: Multi-threaded buffer-based asynchronous solution to the producer/-consumer problem. The consumer received a queue to hold unprocessed items until it can process them.

the FIFO interface (*First In First Out*), and is generally called a *queue* [28]. Most importantly, the queue should be thread-safe; as we said earlier, this means that it must be suitable for usage in a concurrent context, namely it should be reliable in a situation when potentially many actors are trying to access a resource at the same time [67]. Due to the generality of these requirements, implementations are usually part of the standard libraries of most high-level languages. For instance, `java.util.concurrent` provides the `ArrayBlockingQueue` [56], while Python provides a dedicated `queue` module [155]. We discuss with more detail the topic of synchronous queues in Appendix C. One additional desirable feature of these implementations is that they provide features to reject an operation on the queue in certain situations. For instance for some implementations of the queue, if the space in the queue is exhausted, attempting to insert a new object will block the thread until an empty spot becomes available. Similarly, attempting to extract from the queue will block the thread until a new item becomes available [110]. This removes from the developer the duty of designing a policy to poll periodically the queue for free spots or new work to be processed; most importantly, this is also a form of backpressure, as the producer thread will not be allowed to bring in in new work if the consumer is already overwhelmed [87]. This situation is similar to the synchronous regression which we highlighted in the previous paragraph with one important difference: we can fine-tune the queue size so this does not occur too often for our use case. For a robust solution, an additional queue may be used to hold processed items until the producer is ready to take them and forward them to downstream clients. The semantics in this case are identical to the consumer queue.

```
interface Buffer<T> {
    // Append to the tail of the buffer
    void add(T);
    // Retrieve and remove the head of the buffer.
    T pollHead();
}
```

Listing 2: Expected interface for the buffer data structure in Figure 10.

2.1.3 Distributed approach

By *distributed* software we mean an application that is composed of multiple workers running independently. Such workers are typically bare-metal processes or virtual Kubernetes-managed processes [125]. In general, complex software needs workers to cooperate, meaning that we should expect them to exchange data with each other. Depending on the use case, cooperation might involve sharing cache entries via some distributed cache mechanism [24, 45, 165], recurrent queries to fetch data that is only available on a specific server, remote procedure calls (RPC) to functions that are only implemented on a dedicated worker. Distributed code is in general easier to develop due to the fact that we do not need to deal with the problems of multi-threading [62]. Indeed, each worker is expected to have its fully independent memory space, thus removing any need for synchronization or locking. This also means that it is less interesting to treat the in-memory structures that are deputed to hold processing data; indeed, in the present section, we are going to shift the focus on the communication strategy while trying to be agnostic on how data is treated on the workers.

For what concerns the producer/consumer problem, we have already presented the most interesting ideas in Section 2.1.2, and we are going to encounter some of them again in a slightly revisited fashion in the present section. Many of the concepts that we will leverage in the following paragraphs have already been presented as well, thus we are going to conduct a slightly faster discussion in this case.

Blocking approach

The distributed blocking approach is similar to the single-threaded shared-memory approach which we represented schematically in Figure 7. The main difference is the fact that, instead of a simple function call, the producer shall trigger a consumption by exchanging the input data with the worker running the consumer. This might happen via an HTTP call, a remote procedure call, or any other message-passing technique which we will talk about in Section 2.2. The producer

then shall blockingly wait for the remote execution to complete, and for the output to be delivered as the HTTP response body, or as the output of the RPC. This approach offers several advantages over its shared-memory counterpart:

- The workers hosting the producer and the consumer do not share the same memory, thus it is harder to experience out-of-memory crashes, and scaling the architecture is much easier;
- Redundant workers may be spawned, such that in case of a crash of the main consumer or producer worker the expected recovery of the application is much smaller [87];
- Following the microservices concept [103], each worker may use a specific technology which is more suitable for the specific kind of processing it shall take on.

This is indeed the approach employed for the UCAP-Python integration, even before the start of the present work [29]. The last point is the most relevant, since Python workers (the consumers) are expected to execute user-defined Python functions; such a task is rather difficult to accomplish in a JVM-based process, and would require the integration of a third-party library (see Section 1.3 for the details). However, employing Python processes solves the problem in a much more natural and less error-prone way.

Asynchronous approach

The asynchronous distributed approach is again very similar to its shared-memory counterpart, with similar characteristics as the case presented in the last paragraph. From a producer perspective, a *future* (see Section 2.1.2) is usually the natural way to model the execution on the remote consumer worker. Similarly to its shared-memory counterpart, the asynchronous distributed approach may benefit from the interposition of a buffer data structure, both for the producer and the consumer [87]. An important difference is that such a data structure need not be thread-safe, unless multi-threading is employed at some level within a single worker.

An additional possibility that falls under the umbrella of the asynchronous distributed approach is introducing a *broker* (see Section 2.2) between producer and consumer. We will discuss this topic in more detail in a few sections; for now, we will simply state that a broker is a specialized software that receives messages and forwards them where they are needed, providing in addition recovery and replay strategies, as well as extensive monitoring [88]. However, the solution is

not suitable for all use cases, since brokers constitute yet another hop in the architecture.

2.1.4 Additional topics

As we anticipated at the beginning of this section, we will dedicate the coming paragraphs to a discussion on several topics to complete the overview. In particular, we will focus on several ubiquitous patterns in software engineering which are not only related to the context we treat in the present work.

Pipelining

The concept of "pipelining" is pretty much ubiquitous in computer science. One of the most important areas it is applied to is CPU optimization, namely to achieve Instruction Level Parallelism (ILP) [62]. The high-level idea is to let components of the processor act as a pipeline, meaning that they shall all process a work item and pass it to the next piece of the chain. A recent research direction is the application of pipelining in the context of high-performance deep learning [14]. Indeed, it is common to have several components working together (e.g. neural network layers, activation functions), and to establish processing pipelines composed of several steps (e.g. transformers [157]).

We explain the core idea of pipelining with an example: assume that our pipeline is comprised of three tasks in sequence, and that they all take the same time to complete. We start a run of `Task1` on some input, and the other tasks are forced to wait for a task cycle τ. Then, `Task2` is ready to take on the output of `Task1`; however, we can already schedule another run of `Task1` in parallel, with the next work item. Therefore, at 2τ we can start all three tasks, producing the first final result at 3τ. From now on we are able to produce one result per cycle, since each step of the pipeline is busy, meaning that the pipeline is at "regime". In this context, pipelining delivers the optimal task scheduling strategy to maximize throughput. However, real use cases rarely involve tasks all having the same duration, and pipelines are in general much longer. Nonetheless, we can still study how to maximize throughput by scheduling tasks in an optimal order.

Multiplexed communication

Multiplexed communication is in some sense a parallelism paradigm by itself, since it is orthogonal to the other ones that we have presented so far. Multiplexing is the ability of a channel to transfer multiple messages at the same time, in either direction [87]. This means that multi-threaded actors in a distributed

application may seamlessly exchange data without experiencing a bottleneck on the wire. However, it is important to keep track of the utilization of the channel, as overusing it might deteriorate its performance. A typical example of a low-level channel supporting multiplexing is HTTP/2 [32], while gRPC is an example of a high-level communication framework offering built-in support for such feature [61].

Data-centric software

Data-centric software is a peculiar design strategy that focuses on the optimization of data movement. Moving data is indeed quite costly in terms of performance, for what concerns movements on the wire [33], as well as loading in memory [62]. The standard advice is to focus on *data locality*, meaning that operations on a particular portion of a dataset (e.g. a cache line) should be grouped such that the portion is not needed anymore afterward [95]. The "data-centric" paradigm is based on the idea of *data-oriented software*, which proposes an alternative to the ubiquitous adoption of object-oriented programming. The statement is that for some applications it is beneficial to focus on modeling data instead of "smart" components, and to structure software in terms of relations between data types [57]. Business logic should be as simple as possible, and most importantly should be clearly separated from data definition.

These concepts were brought to a different level of importance in data-intensive contexts like deep learning. In the field, it is common to work with datasets that cannot even fit in the memory of standard computers [14]. To work around the issue, algorithms are usually designed to scan the dataset by considering only small portions at each iteration, which is typically known as *minibatching* [14]. As a consequence of the steady increase in size of training datasets that we have witnessed in the last few years, especially considering the recent impact of *large language models* (LLM) [150], it has become critical to make sure data locality is exploited as much as possible [75]. To this end, we have observed the emergence of dedicated tools for the automatic optimization of deep learning algorithms [11,30]. A few optimization examples are reordering pipelines such that data is used in an optimal way, or pre-fetching a piece of a dataset in the background while working on something else [20]. These tools are based on algebraic manipulations which aim at finding dependency graphs among data processing pipelines.

Lazyness

The concept of *lazyness* is not necessarily related to asynchronous communication, but it holds an important role in asynchronous software. We may call an object "lazy" if its creation is delayed until it is actually needed [20]. An object is

```python
import asyncio as aio
import numpy as np

async def fetch_dataset() -> np.array:
    ...

dataset = aio.create_task(fetch_dataset())
# Do something else ...

await dataset
compute_result(dataset)
```

Listing 3: Example usage of lazily initialized variables in Python based on the AsyncIO framework.

"needed" when some code statement uses its value as part of an operation, unless the statement constitutes the initialization of another lazy object. Lazy objects look like standard objects from the outside, while under the hood they are simple references to some running function that is responsible for their initialization. They are in some sense similar to *futures*. The convenience of this kind of object relies on the fact that most variables in software are not necessarily needed immediately, as soon as they are created. Most of the time they serve as parameters, either for the construction of another object or for a function. If one manages to expose the graph of dependencies among them, the actual creation time can be chosen arbitrarily at runtime, thus saving CPU time and memory space. In addition, lazy variables may represent values that the program will need at some point in the future, but not necessarily now, thus allowing for a simple implementation of pre-fetching. A simple example is shown in Listing 3. Lazy objects are typically represented by `Future<T>` instances in Java (where `T` is the type) [110] and "awaitable" objects in Python's AsyncIO [155]. The main shortcoming of lazy initialization is that it is much harder to debug, as it may not be trivial to understand where and why a lazy object is initialized.

2.2 Inter-process communication in practice

After examining inter-process communication from a high level, we now dive one level deeper into the matter by discussing practical ways to exchange information. Since sharing resources among workers is a ubiquitous need among the majority of software systems [87], the market offers a plethora of solutions, each of which is suitable for a specific set of applications. Depending on the coupling level of the workers, we could establish a communication channel via shared memory, HTTP requests, or by querying the content of a "broker" service. In this section,

we are going to examine briefly some of these solutions. This knowledge will be propedeutic to motivate design choices which are critical to achieving the end goal of the work. We are going to keep the discussion on a high level, as the concepts we will present are typically valid for all technologies.

2.2.1 Low-level inter-process communication

In the first part of this section, we are going to examine a few selected low-level approaches for inter-process communication. Contrary to the solutions we will present starting from Section 2.2.2, these approaches traverse a limited set of abstractions. Most of them are typically attainable by leveraging tools from the standard library of most programming languages. Common communication facilities like backpressure, health checks, or load balancing, are not provided out of the box when using low-level approaches; still, these concepts maintain a critical importance in the industry, since they are usually the base for more complex tools. Nonetheless, we decided to omit to treat some powerful low-level approaches like *socket-based* IPC: nowadays, it is very uncommon to use directly the socket interface since modern technologies provide a much safer and intuitive experience [84].

Disk-based communication

The most basic way to exchange data is most probably instructing the worker holding the information to write a file in the persistent storage of the machine, and then accessing it from the second worker. This approach has some important benefits which shall be taken into account:

- It does not require any specific setup or external technology, as most languages provide immediate support for file IO. In complex computing infrastructures like HPC clusters, it's not uncommon to lack superuser permissions, and this often limits the possibility of installing new software. In such environments, simple yet effective solutions are very frequently employed, also because they are frequently quite performant.
- We lack better solutions for massive amounts of data that do not fit in the volatile memory of the machine. In HPC applications it's very common to establish multi-step computing pipelines. The initial step usually produces a huge amount of input information — for instance, large multi-dimensional arrays produced during high-fidelity simulations or physical measurements — to be consumed by subsequent steps in a chain. A common pattern in

such cases is to store the result of each step as one or more files in the persistent storage of the cluster — usually called `scratch` [92] — and to access such file as needed in subsequent steps. This also eases the implementation of *checkpointing*, namely saving partial results of an HPC application in order to restart from the most recent in case of errors [113].

- The code might be easier to design and maintain, since instead of complex idioms with obscure internal behaviors we simply need to deal with a static file.
- Workers exchanging data via the disk are easily debuggable: one could inspect the files they exchange to verify whether they are behaving properly.

Considering the performance aspect, disk-based IPC is known to be quite inefficient due to the limitations of persistent memory. However, we are not required to load the full content of the file in memory: each step could seek the position that it needs and read only a specific number of bytes. Moreover, to speed up read/write operations we could employ a binary format, which we are going to discuss in more detail in Section 2.3.

Aside from performance considerations, there are also some other important limitations to take into account:

- Disk-based IPC becomes unfeasible when the persisted resources are not static. Updating values in the file presents a problem in terms of performance, since writing to the persistence storage is one of the least efficient operations in software [62]; and most importantly in terms of concurrency. We do not know when the resource is safe to access, thus we expose ourselves to processing inconsistent data due to a partially persisted update.
- If the nature of resources to be shared is not well defined or subject to change, it may be difficult to exchange data among workers: the downstream ones expect to know in advance how many bytes they should read, or what to do with the content.

To summarize, disk-based IPC is a powerful tool for some applications, but its applicability domain is usually limited to large-scale computing applications dealing with important amounts of data, for example offline data processing platforms (see Section 1).

Pipe-based communication

A *pipe* is a mono-directional channel of communication between two processes. It can be established between a parent and a child process at the time of the

```
1  $ cat input.txt
2  x=2.0, y=-0.1, z=10.1
3  x=2.0, y=0.0, z=10.5
4  x=2.0, y=0.1, z=10.4
5  x=2.1, y=-0.1, z=4.1
6  x=2.1, y=-0.1, z=4.1
7  x=2.1, y=0.0, z=5.3
8  x=2.1, y=0.1, z=2.2
9  $ cat input.txt | grep "x=2.1" | uniq
10 x=2.1, y=-0.1, z=4.1
11 x=2.1, y=0.0, z=5.3
12 x=2.1, y=0.1, z=2.2
```

Listing 4: Example pipe usage in Bash. The commands in the listing are `cat` (print file content), `grep` (keep only lines containing the specified pattern) and `uniq` (discard duplicate lines).

creation of the latter, and it may remain alive for the whole lifetime of the workers. Pipes have limited size, which varies on the system [84]. After the pipe space is exhausted, it does not accept new data until the next read operation of the downstream client, which frees space in the buffer. This behavior acts also as a rudimentary backpressure mechanism [87].

Pipes are available out of the box in many programming languages, including Python as part of the `multiprocessing` module [155]. In Listing 4 we show a simple example of the usage of pipes in Bash. Each command following a pipe is started in a subprocess, whose standard input is the output of the preceding pipe [84]. Therefore, line 9 in the listing is effectively a multi-process pipeline for data filtering, with pipe-based communication to glue the workers. Considering large input sizes, this simple line of bash will outperform any single-threaded Python or C++ script in terms of throughput [87].

Pipes have quite different applicability with respect to file-based IPC. First of all, they require a much more coupled architecture, since the receiver worker must be started by the sender. Secondly, they are quite harder to use than other solutions due to their size limitation. In general, this IPC strategy is meant for predictable payloads between highly coupled workers.

Shared memory communication

Shared memory IPC, as the name suggests, relies on a shared fragment of memory that is accessible by all parties involved to share information. Many different technologies exist that allow achieving such a setup:

- In some cases the developer receives a piece of raw memory that can be

```
1 int sum = 0;
2 #pragma omp parallel num_threads(10)
3 {
4     #pragma omp critical
5     sum++;
6 }
7 std::cout << sum << std::endl;
```

Listing 5: Multi-threaded summation with OpenMP. The compiler directive at line 2 spawns 10 OpenMP threads which are only active in the section between lines 3 and 6. The directive at line 4 instructs the compiler to prevent multiple threads from accessing the instruction concurrently, thus making the update to the variable `sum` thread-safe. An implicit barrier at the end of the `parallel` section makes sure all threads managed to update the variable exactly once before it is printed to `stdout`.

read and written as needed by multiple workers. An example is the system function `shmget()` in the C programming language [83]. In such cases handling access rights, read/write order, as well as concurrency issues is the responsibility of the developer.

- Other technologies like OpenMP [26] handle the full lifecycle of threads, and provide portable facilities to manage shared resources. Communication is carried out via shared variables that must be accessed in a thread-safe manner. We show an example snippet in Listing 5.
- High-level languages based on object-oriented programming usually define their own concurrency development suite as part of the standard library. For example, Java ships the standard package `java.concurrency.*` [110], while Python provides the `threading` module [155]. They typically provide implementations of common concurrency primitives (see also Appendix B), as well as means to start and control the lifecycle of threads within a process.

Even though shared memory communication is one of the most popular means to exchange information, it has critical shortcomings which must be carefully addressed by the developer. In the next paragraph we briefly discuss some of the most important, outlining also possible solutions.

The perils of shared memory communication

As we anticipated both in Section 2.1 and Section 2.2.1, shared memory presents intrinsic risks and shall be used with care. We have already mentioned the concept

```
1 class SimpleInteger {
2     private int value = 0;
3     public void sum(int i) {
4         value += i;
5     }
6 }
```

Listing 6: Simple Java class which represents an integer. It supports summing an `int` value to the internal field `value`, via the method `sum(int)`.

of *thread safety* without diving into the details for clarity purposes. The present paragraph is devoted to filling this lack.

We will start the discussion by introducing the important concept of *race condition* with an example. The Java snippet in Listing 6 defines a basic class `SimpleInteger` which holds an `int` value. The value may be incremented from outside with a call to the method `sum(int)`. Assuming a multi-threaded environment, an instance of `SimpleInteger` may live in shared memory, thus being at risk of being touched by more than one thread. Such an occurrence would be catastrophic since the class is not thread-safe. In Listing 7 we present the JVM bytecode of the method `sum(int)`. The simple summation at line 4 in the Java snippet is translated to a sequence of six bytecode instructions. Without going into the low-level details, the bytecode instructions between line 3 and line 6 are required to load the parameter value and the class field into the first two positions of the stack memory of the current thread, while the actual sum is computed in line 7, and subsequently assigned to the class variable. The important information to take away is that simple instructions in high-level languages might be translated into several low-level instructions, thus making it impossible to execute them in an *atomic* way. By "atomic" we mean that the state of the program does not change for the whole duration of the operation [62]. Indeed, we now examine

```
1   public void sum(int);
2     Code:
3        0: aload_0
4        1: dup
5        2: getfield      #2                  // Field value:I
6        5: iload_1
7        6: iadd
8        7: putfield      #2                  // Field value:I
9       10: return
```

Listing 7: JVM bytecode for the method `SimpleInteger#sum(int)`, obtained using the tool `javap` [110] from OpenJDK 17.0.9.

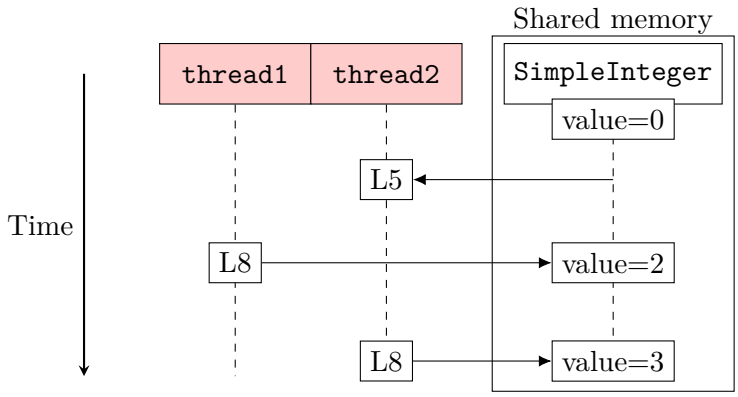

Figure 11: Schema of the race condition in `SimpleInteger`. Line numbers refer to the bytecode in Listing 7.

what could go wrong when threads come into play. We assume having initialized a new instance of `SimpleInteger`, such that the internal `value` is equal to zero. The expected outcome of two calls from two different threads, e.g. `sum(2)` from `thread1` and `sum(3)` from `thread2`, is an instance such that `value=5`. However, if `thread1` manages to update the internal class field after `thread2` executes line 5 of the bytecode snippet, `thread2` will have lost the update. It will then proceed to summing zero and three and then assigning the result to the class field. The final result in this case is three. The workflow is represented schematically in Figure 11. A similar error could happen by switching roles, and in such case we would obtain a final value of two. An example run is shown in Listing 8.

```
1  int count2, count3 = 0;
2  for (int i = 0; i < 1_000_000; i++) {
3      var si = new SimpleInteger();
4      var t1 = new Thread(() -> si.sum(2));
5      var t2 = new Thread(() -> si.sum(3));
6      t1.start(); t2.start(); // Start t1, t2
7      t1.join();  t2.join();  // Wait for completion
8      if (si.value == 2) count2++;
9      if (si.value == 3) count3++;
10 }
11 // Output:
12 count2 = 5 (0.0005%)
13 count3 = 3 (0.0003%)
```

Listing 8: Reproducing the race condition in `SimpleInteger`. It is interesting to notice that the problem occurred only eight times out of one million trials.

As we can see from the output, our program is non-deterministic and might fail at any time, even though a considerable number of runs went through without any issues. Race conditions are essentially ticking bombs that may manifest at any time with completely unexpected values. The careful software engineer must adopt appropriate measures to protect against them, like those which we talk about in Appendix B.

Thread safety amounts to only a portion of the problems posed by shared memory software. Other issues may not cause the program to produce incorrect results; however, they may manifest themselves as inexplicable degraded performance. This is the case of *false sharing*, which is a common problem when *cache coherency* mechanisms are put in place [62]. In simple words, cache coherency aims at making sure that each worker sees all necessary updates on their cached copy (i.e. locally retained) of a shared variable. Typically, a write operation on a shared variable causes all cached copies to be invalidated; this has an important impact on performance since all workers must then go to the main memory to obtain the updated value of the variable. Cache coherency is usually implemented at a *cache line* level, meaning that an update on a specific address of the cache line invalidates the whole set of addresses of the block [27]. This is the essential problem of false sharing: all addresses of the cache line are invalidated and must be read again from the main memory, even though a single address was updated. To mitigate the issue, the developer should make sure that memory writes are not transversal to the cache lines, which may be achieved by carefully organizing the memory usage of each worker [62].

2.2.2 Distributed inter-process communication

We now turn to communication methods which are conceived to exchange data among parties that do not have access to the same memory region. There is virtually no limitation on the membership to this category, therefore the methods which we are going to present in the coming paragraphs are quite general and may be suitable for many applications. Moreover, the fact that memory is not shared translates into a much cleaner code: indeed, the software engineer need not pollute the code with thread lifecycle management or synchronization primitives like those which we mention in Appendix B. Most importantly, the shortcomings that we presented in the previous paragraph are irrelevant in the context of distributed communication, which means an important guarantee of code testability and safety. These advantages come in exchange for an increased communication cost, due to the additional hop on the transport layer, and an increased dependency on third-party libraries, since the general support for distributed communication in standard libraries is typically lacking [110,155]. In the

following paragraphs, we are going to discuss the most important methods from a high-level point of view.

Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP) is the most popular alternative for distributed inter-process communication. In the last few decades, it has become ubiquitous due to the fact that it is the foundation of the World Wild Web [145]. We will not go into the details of HTTP, as we are only interested in how it is used and how it compares with respect to the alternatives. An HTTP request is characterized by the following information:

- Uniform Resource Locator (URL) of the host which shall reply to the HTTP request;
- The HTTP Method, defines the kind of action that is expected from the server. The most common examples are `GET` to retrieve, and `POST` to provide information;
- A request header, namely a set of key/value pairs to share metadata information with the server;
- A body, which is optional and is typically used to push data to the server.

An HTTP response has a similar structure as the request, with the addition of a status code that informs the client about the outcome of the operation. Common statuses are 403 (not authorized) 404 (not found), 500 (internal server errors), and 200 (OK).

HTTP is based on the Transmission Control Protocol (TCP), which provides several important high-level guarantees. Data shall be delivered in order, with guaranteed delivery, and automatic retransmission in case of errors. Also, the data received is checked to ensure that no bit was changed during the transport over the wire [145]. Such a contract made HTTP particularly appealing for all kinds of applications; indeed, nowadays the protocol is not only the base of the World Wide Web, but also the most common communication protocol in the microservices world [36, 87, 103, 149].

Due to the importance of HTTP, many libraries exist to simplify software development for both client-side and server-side operations. For the former, the typical need is to provide an API that integrates well with the rest of the code, meaning that it agrees with the type system (e.g. via Java generics [110]) or it works well with the built-in asynchronous programming model (e.g. with AsyncIO

```python
import requests
url = f"http://{host}:{port}/"
response = requests.get(url)
```

Listing 9: Simple Python HTTP client based on the open source library `requests`.

for Python [155]). A few examples for Java are the built-in `HttpClient` (introduced in Java 11) and Apache HttpComponents' client; for Python, the most popular solution is `requests`, while `aiohttp` provides seamless integration with AsyncIO. We provide a basic example of an HTTP client based on `requests` in Listing 9.

Remote Procedure Calls and RESTful web services

Even though Remote Procedure Call (RPC) and RESTful web services are not concrete tools, but rather specifications and sets of ideas to structure client/server applications, we believe we deserve to be mentioned separately due to their importance. We dedicate a few lines to each of them since they are important concepts that we will also consider hereinafter.

RPCs attempt to model network calls to remote servers as ordinary functions. The idea is to generate *stubs* based on a service definition. The service definition typically defines, for a given server, the set of functions it supports, including their signature. Using automated code generation tools, the client generates (at build time) the code that provides the implementation for the stubs. Thus, remote calls are fully integrated into ordinary code, in particular for what concerns types and exceptions. Callers need not deal with low-level details like HTTP requests/responses, status codes, or JSON deserialization. This is also one of the most important criticisms of RPC: hiding the fact that an apparently simple function call will result in a network call is believed to be a problem for code clarity and safety [87]. One of the most popular RPC engines in the industry is gRPC, developed by Google Inc. [61], which is based on HTTP/2 and Protobuf [59]. gRPC supports several popular languages like Java, Python, C++, Go, and Dart, and provides many useful features like multiplexing and bidirectional asynchronous streaming. We provide an example service definition in Listing 10.

Representational State Transfer (REST) is an alternative approach to structure client/server applications, which is particularly suitable for CRUD software (CReate/Update/Delete). The idea is that each request shall represent an operation of some kind (read, create, update, or delete) on a particular entity. The kind of entity shall be identified by the URL [103]. In addition, the server should not retain any status information about a specific client, meaning that it must

```
message Person {
  string name = 1;
  string surname = 2;
}
message PhoneNumber {
  string countryPrefix = 1;
  string value = 2;
}
service PhoneBook {
  rpc findPhoneNumber (Person) returns (PhoneNumber) {}
}
```

Listing 10: Example gRPC/Protobuf service definition. We defined two message types, `Person` and `PhoneNumber`. Then, within the `PhoneNumber` gRPC service we defined an RPC which returns the phone number given a person's identity.

be stateless, excluding the entities it holds. RESTful web services are typically based on HTTP, since the two share similar properties and requirements [145]. Therefore, REST client calls can be implemented with simple HTTP clients, and a similar statement holds for RESTful web services.

User Datagram Protocol

The User Datagram Protocol (UDP) is an alternative communication protocol to TCP which provides different capabilities. It does not guarantee message delivery, nor it checks for transmission errors, and may not enforce delivery order [145]. However, it does not require establishing a communication channel before data is sent over, which results in an important overall speed up. Moreover, opening communication channels poses an important load on the web service, which is absent when UDP is employed. All of these factors must taken into consideration since this protocol is definitely not suitable for all applications; however, there are some that could massively benefit from its lightness, especially in contexts where missing some messages is not a problem. For instance, UDP is employed at CERN in a distributed tracing library [42]. Due to the massive amount of log messages produced every second, losing a small percentage was deemed acceptable during the design phase.

Remote Direct Memory Access

Generally used communication protocols like TCP and UDP require many steps to acquire a message from a client and present it as an in-memory object to the appropriate running software. The process typically involves the CPU and several memory layers, since the message must be acquired from a socket and temporarily

copied into a buffer, and then from the buffer to the program memory [145]. These steps waste CPU cycles and induce memory congestion for what may look like useless data movements. The idea of Remote Direct Memory Access (RDMA) is to skip these steps and decouple data transfer from CPU. The client and the server shall initiate a channel that links two memory regions in the two hosts. Then, any further data transfer will access the memory of the target directly, without an intermediate buffer or additional memory steps [127].

However, RDMA has limited applicability in mainstream software development. Specific network technologies (like Infiniband) shall be put in place between the hosts to enable its usage. Moreover, the initialization of the channel requires some time to be properly established, which makes RDMA not suitable for ephemeral communication events. The mapped memory region cannot change in size, thus the maximum amount of data exchanged shall be carefully agreed upon by the parties [49]. Considering the tooling availability, there is a clear lack of industry-ready libraries with long-term support and stability guarantees, and most of those available nowadays are being developed by research laboratories for experimental purposes [146]. RDMA was reportedly employed fruitfully in large-scale applications within HPC clusters [161], which are commonly equipped with bleeding-edge network communication technologies due to the humongous volumes of data they orchestrate. As of 2024, the impression is that RDMA is not mature enough to be included in a project with critical stability requirements like UCAP. However, it is definitely an interesting technology to observe carefully for future evolutions.

Broker-based inter-process communication

In addition to the communication strategies that we presented in the previous paragraphs, one approach emerged recently due to the high adoption it is encountering thanks to the recent Kubernetes trend in software engineering [125]. We are witnessing a growing need for stateless software systems, which may be easily restarted, killed, and duplicated as needed by a centralized orchestrator. This need stems from the fact that clusterized deployments require full independence of the software from the location where it is running, as well as all instances to be almost indistinguishable. This means that the whole state should be migrated to an external reliable storage, which may be queried by any running instance of the software.

Such an external storage is usually called *broker*. A broker is an independent application that is responsible for receiving, storing, and forwarding data as needed. This kind of software typically allows to create multiple *topics*, namely streams of information related to different domains. A consumer may subscribe

to a specific topic it is interested in, in order to consume all published messages. At the time of writing, there are many solutions available, like Apache Kafka [88], Redis [24], and RabbitMQ.

The most important shortcoming of broker-based IPC is that it introduces two additional hops [33]. This is an aspect to keep in strong consideration, especially for low-latency systems. For most applications though, the added value of decoupling data producer and data consumer and the benefits of running both in a clusterized environment would typically justify the slightly reduced throughput.

2.3 Data

The concrete materialization of what is actually being exchanged between parties in inter-process communication is a key factor. The choice of the serialization protocol determines an important portion of the communication speed [87, 124]. However, the choice is not always on the software engineer: it may be imposed by an upstream dependency, or by the architecture of some legacy project that is owned by another team. For example, the vast majority of microservices nowadays are designed to be JSON-based RESTful interfaces [103].

In this section, we are going to present several possibilities for data format, as well as their implications in productional environments. We will start with a high-level taxonomy, to provide a simple framework to distinguish available options in well-defined categories; then, we are going to present several alternatives for each of these categories, in order to fix ideas and discuss some concrete use cases. We will also examine some high-level performance comparisons between the options, since performance is of critical importance during the software design process, and is also the main focus of the present work. Finally, we are going to discuss briefly some common best practices to optimize data exchange.

2.3.1 High level taxonomy

Data is information, which may manifest in different forms. Clearly, only a few of these forms are compatible with computers, which can only understand what may be encoded in sequences of bits. Over the last few decades, computer engineers conceived a plethora of ways to represent data, mostly driven by the needs of the most popular trends at the time. However, not all formats stood the test of time; indeed, the oldest format which we talk about in the present document, namely XML, was first introduced in 1996 [124]. In order to introduce order in the overwhelming amount of options to represent data, we will start the discussion by proposing several categories, to which a particular data format may or may not belong.

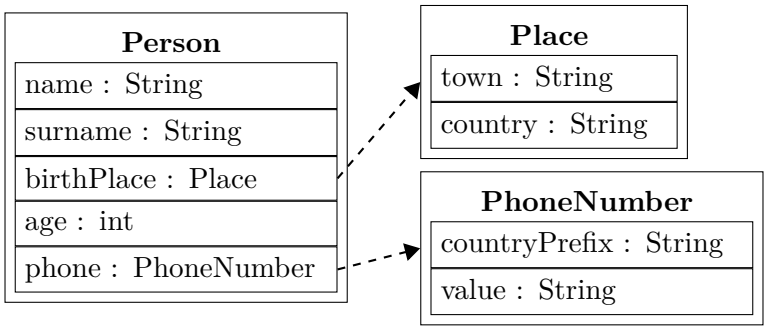

Figure 12: Example structure of an object-oriented class hierarchy. `Person.birthPlace` and `Person.phone` are references (or pointers) respectively to the classes `Place` and `PhoneNumber`.

In-memory data

In-memory data is the most common type of data a software engineer typically deals with. Memory may contain simple values (i.e. what in Java is usually referred to as *primitive* types [110]), or hierarchies of classes whose fields refer to instances of other classes. Moreover, it may contain arrays of types, as well as *collections* (e.g. lists, sets, maps). Depending on the programming languages, the in-memory representation of a particular object varies. For instance, due to the implementation of the CPython garbage collector, the representation of each object contains an additional counter that keeps track of the current incoming references [155]. On the contrary, low-level languages like C and C++ store simple types, as well as arrays and class members, as adjacent values in memory (if we ignore the additional padding added by the compiler in some cases [62]). However, references are a sore point in all programming languages. A reference (or pointer) is essentially a value holding the position of an object in memory. We propose an example hierarchy in Figure 12. The in-memory representation of class hierarchies is not adjacent, since references could point anywhere in the memory managed by the program. Introducing abstractions makes the life of the developer easier when it comes to modeling real-world problems in a programming language; however, the in-memory representation of abstraction hierarchies are typically not trivially serializable, meaning that they cannot be sent on the wire without being "massaged". This is the main purpose of the objects belonging to the next category.

Serialized data

Serialized (or *marshalled*) data is a representation of some in-memory object which can be sent on the wire [87, 124]. Therefore, the importance of this category of objects is critical: serializing in-memory data allows exchanging information with other processes via one of the techniques that we discussed in Section 2.2, provided that the receiver knows how to transform the payload into an in-memory object it can use (*deserialization*, or *unmarshalling*). For this reason, serialization/deserialization tasks are extremely common and very important in software development. Several surveys have found that they comprise about 5/6% of the total computing power of Google's and Meta's data warehouses [82,139]. Depending on the use case, there are many ways to serialize data. We are going to deepen the topic in the next two paragraphs.

Textual data

The first question to pose when selecting a serialization data format is whether the information shall retain the property of being readable by a human while being transferred on the channel. This is a very important point, since it influences the ease of debugging when dealing with distributed systems [94]. Having unreadable sequences of characters flying makes identifying the source of the problem much harder, in particular because the fix is typically required in a short time. Textual serialization formats solve the problem by making sure the data is readable, and most importantly has a schema that makes it easy to fathom the structure of the information. Moreover, the fact that the serialization format is open and easily understandable typically translates into the abundance of third-party tooling in the market. Indeed, all programming languages have multiple parser implementations for the most popular textual data formats, sometimes even as part of the standard library [155]. We will now examine a few examples.

The *eXtensible Markup Language* (XML) was the most popular format in the early stage of the web, thanks to its huge flexibility and the amount of tooling available. However, it slowed down during the last few decades, being surpassed by less verbose data formats in most contexts [105, 141]. It supports schema validation, with the possibility to define namespaces as well as new types. XML is nowadays the standard format for SOAP-based applications; however, it has been largely surpassed by *JavaScript Object Notation* JSON for most of the other web-based use cases like RESTful web services. JSON is easier to interpret, both for humans and computers thanks to its reduced verbosity [105]. Nonetheless, it lacks some features from its older alternative, but it is typically not a problem for standard applications. Both XML and JSON are human-readable, but

they are not the preferred alternative for files that may be edited by hand with a text editor. In such a case, formats like *YAML Ain't Markup Language* (YAML) or *Tom's Obvious Minimal Language* (TOML) shall be preferred. YAML has received attention lately thanks to the Kubernetes trend and is mostly used in configuration files for various services, especially in the field of DevOps (e.g. Ansible, GitLab CI, and Docker). This format provides a much less verbose experience than the previous candidates, and its syntax may remind of that of Python due to its peculiar indentation-based appearance. YAML supports scalars, arrays, and map-based data structures, and it provides a similar typing system as Python, supporting many standard types like dates [10]. Moreover, YAML provides interesting features to render text with special indentation and newlines (the *pipe* "`|`" and *greater than* "`>`" characters) which make it much more pleasant to store and edit quickly text-based data. The same applies to TOML, which is explicitly designed to facilitate human readability and ease of modification, while keeping an easily parsable structure for a computer. TOML has recently been the subject of a PEP (*Python Enhancement Proposal*) to designate it as the standard format for Python project configuration files [23]. Even more remarkably, starting from Python 3.11 the standard library includes a TOML parser [68, 69], thus boosting significantly its adoption.

Binary data

Unlike textual serialization formats, the binary ones have no requirement to be human-readable in the first place. Indeed, they typically consist of a sequence of bytes that require specific tooling to be mapped into a human-readable in-memory object. Binary formats are typically conceived for maximizing the serialization/deserialization speed and minimizing memory occupancy [124]. For this reason, their specifications tend to be rather convoluted [46, 59]. Due to their performance guarantees, binary formats are typically preferred for low-latency applications. Contrary to textual data formats, binary encodings tend to manifest as complex third-party dependencies shipped as considerably large libraries, not necessarily open source. Internal tricks and optimizations are frequently employed to maximize the throughput, [59] therefore it is typically not trivial to write a custom serializer/deserializer for binary formats. The risk of becoming extremely coupled to such external service must be considered before choosing a data encoding, it is fundamental to verify the support of the format on all frameworks and platforms that we aim to target, now or in the future. We will now look at some examples, taking into account only those formats which are backed by libraries mature enough to be considered standard, available for multiple languages, and backed by a development team that guarantees long-term support.

Protocol buffers (Protobuf) is an open-source multi-language framework for binary serialization/deserialization developed by Google, and reportedly used for most of its products [59]. Developers are responsible for writing a *schema file*, which contains the definition of all the object types that may be exchanged. Objects may contain strongly typed fields, with the available types being the standard primitive types, like floating point numbers, integers, and strings, as well as complex ones, like arrays or nested hierarchies of user-defined types. An example Protobuf definition was already presented in Listing 10. The schema file is compiled using a dedicated compiler called *protoc*, which produces the needed glue code to implement the specified schema in a given language, in order to enable using types defined within Protobuf in the code. Protobuf is the standard serialization format employed by gRPC [61], which we have already talked about in Section 2.2. On the other hand, one of the most popular competitors of Protobuf is *Apache Avro* (commonly known simply as Avro). Avro provides similar functionalities, with a slightly different approach. Avro packs the writer schema of the object, which shall be specified as a JSON document, as part of the serialized payload. This translates into a slightly bigger message to be exchanged; however, the benefit of this approach resides in the fact that the writer schema is available at deserialization time, thus making the management of multiple deployments running different versions of the software much safer. Moreover, Avro allows deserializing byte streams to JSON out of the box, thus enabling a much smoother debugging experience with respect to other serialization formats.

2.3.2 Performance comparison

In the next few paragraphs, we are going to lay out a basic benchmark to explore the technical solutions that we presented in Section 2.3.1 from a practical point of view. We are going to start by describing the experimental setup, namely the technologies adopted and the measurement strategy; then, we are going to present briefly the data generated for the experiment; finally, we are going to discuss the results of the benchmark, trying to infer indications for the future development of this work as part of the process.

Experiment setup

The benchmark was implemented in a standard fashion, by recording the elapsed time for serialization and deserialization for a given number of times. The code to reproduce our results is publicly available in a Git repository at https://github.com/fandreuz/python-serialization-benchmark. We provided a simplified strategy to execute the benchmark against an arbitrary Python version

```
>>> docker run --interactive --tty --rm \
        -v .:/home \
        -w /home \
        --env BENCHMARK_DATA_SIZE=$SIZE \    # data size (i.e. array length)
        --env BENCHMARK_ITERATIONS=$COUNT \  # number of measurements
        --env BENCHMARK_TYPE=numeric \       # or 'textual'
        python:$PY_VERSION \
        ./run_benchmark.sh
```

Listing 11: Example command to run the serialization benchmark. The benchmark supports several environment variables to select, for instance, the needed Python version, the size of data, and the number of iterations.

leveraging Docker images. This approach has two main benefits, the first being that it is easier to run the benchmark in isolation, using an independent interpreter version and libraries than those that are locally installed, and the second being that the benchmark may run without any initial effort on any remote server that we may want to probe. An example command is provided in Listing 11. The script `run_benchmark.sh` will also generate the needed files for libraries requiring the compilation of schema files (e.g. Protocol Buffers [59]). Note that the Python version is specified via the environment variable `PY_VERSION`: any of those available in the public Docker registry may be used for the benchmark [100].

Data

Due to the differences between serialization formats that we highlighted in the previous paragraphs, we conceived two different benchmark inputs. Each of them focuses on specific data types, so we can expect to observe meaningful differences in the benchmark results, reflecting the aforementioned theoretical aspects. We aim to identify the most viable candidate for the UCAP-Python use case with an informed approach based on empirical observations.

The first input type is comprised of an `information` field, which contains string and integer fields like an author, a description, and a unique ID; it also contains four numeric 64-bit float arrays representing a scalar field in $\mathbb{R}^3$. In Listing 12 we propose an example instance of a data entry of this kind. The second input type has the same `information` field, but it has three large string fields called `abstract`, `text`, and `appendix`, to model a schematized journal paper entry. The idea of these two experiments is clearly to assess the performance of the serialization libraries in two very different use cases, the first being focused on numeric arrays and the second targeting textual data. The data type may be selected in our benchmark by setting an appropriate value for the environ-

```
{
  "information": {
    "name": "NumericBenchmark",
    "description": "Data for the numeric benchmark",
    "id": 0,
    "author": "fandreuz@cern.ch"
  },
  "x": [0.0, 0.25, 0.5, 0.75, 1.0],
  "y": [-1.0, -0.5, 0.0, 0.5, 1.0],
  "z": [0.0, 0.25, 0.5, 0.75, 1.0],
  "values": [0.5962199732217482, 0.3652215020407171, 0.14453553123599183,
             0.7150423350185419, 0.4757748564058194]
}
```

Listing 12: Example object based on numeric data for the serialization library benchmark.

ment variable `BENCHMARK_TYPE`. The accepted values in the first revision of the benchmarking code are the strings `"numeric"` and `"textual"`. In the coming paragraphs, we are going to provide the results for both data types.

Results

We now look at the results of the benchmark for both the data types depicted in the previous paragraph. Thanks to the flexibility of the code infrastructure and of the execution strategy based on Docker images, we provide results for multiple Python versions. We chose to probe Python 3.9.18 (August 2023) and 3.12.2 (February 2024), since they are respectively one of the most commonly used Python versions, and the newest one currently available to the public. The third-party libraries that were present in the environment during the execution of the benchmarks are the same for both Python versions, and they are listed in Table 4.

All benchmarks have been executed by considering 1000 iterations, considering the 99th percentile value of the measured times for serialization/deserialization. This choice is frequently recommended as a good indicator of the observed performance in most use cases [87]. For both data types, we set `BENCHMARK_DATA_SIZE=100000`, which translates in practice to a data structure containing four arrays holding 100000 64-bit floats in the numeric case, and one containing three strings with 100000 characters each in the textual case. The benchmarks have been run within Docker images pulled from hub.docker.com, on a ThinkPad X1 Carbon Gen 9 laptop.

Respecting our expectations, binary serialization protocols outperform their textual counterparts by several orders of magnitude. This is especially evident

Table 4: Python dependencies in the environment used for the serialization benchmark. The column *Type* refers to the kind of serialization which is performed by the library. `numpy` and `grpcio-tools` are pulled in to be used as utilities.

Package name	Version	Type
avro	1.11.3	Binary
grpcio-tools	1.60.1	×
numpy	1.26.4	×
orjson	3.9.14	Textual
protobuf	4.25.3	Binary
python-rapidjson	1.14	Textual

for numeric-prevalent data types (Figure 13); the difference is slightly less pronounced for text-prevalent data (Figure 14). The performance of *Apache Avro* is comparable to that of *Protocol Buffers* in the first case, but falls behind in the second one. The best performer in both cases is *Pickle*, Python's standard serialization library. However, as we anticipated such protocol cannot be used for language interoperability, since we could not find a Java counterpart to wire the two languages. From this benchmark, we conclude that *Protocol Buffers* is the most versatile serialization protocol, providing official support for both Java and Python.

2.3.3 Manipulating data

It is possible that even using a performant data format, a state-of-the-art toolchain for serialization/deserialization, and an optimized communication pattern for interprocess communication, is not enough to achieve the throughput we aim for. In some cases, this may be due to the scale of data we are treating: indeed, moving data is an extremely time-consuming activity compared to the speed of nowadays CPUs [11, 62, 124]. Processor cycles occur orders of magnitudes faster than memory cycles [27], and this holds for persistent storage as well as volatile. Moreover, exchanging data may be subject to throttling imposed by the transport layer — for instance, gRPC poses a hard limit of 200MB per message [61] — thus making the solution unfeasible for messages exceeding the threshold. The brand-new software we are designing may be bound to a safe zone in which it can operate reliably, but important refactoring and reorganizations will be needed to scale it. Finally, in-flight data (e.g. moving in a network, in a wire) is known to be much more problematic than its in-memory or on-disk counterpart. The first reason for this is that using a network may require the user to pay a cost; the second is that

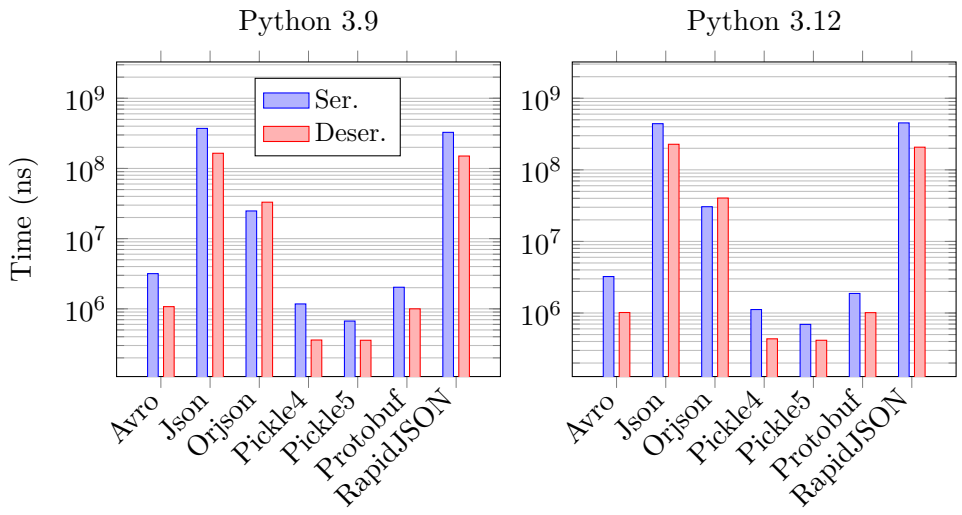

Figure 13: Serialization benchmark results for numeric-prevalent data types. Both serialization (*Ser.*) and deserialization (*Deser.*) are reported. The vertical axis is on a logarithmic scale, and times are expressed in nanoseconds.

moving data across any kind of mean is susceptible to damage (e.g. due to an electromagnetic field) and to being observed by some third-party actor. Having considered all these critical points, in the following paragraphs, we briefly outline several high-level approaches to minimize the impact of communication on the software throughput. These strategies are quite general, therefore they typically apply to a vast majority of mainstream applications.

Data partitioning

The most basic approach to solve message limit issues is to partition our data into multiple packets; then, each packet shall be sent separately on the channel in a blocking fashion [124]. This approach is simple yet effective in most cases, as it yields the desirable side-effect of inherently ordered delivered packets, thus it's trivial to reassemble the original message. Splitting the message into multiple sequences of bytes and reassembling it on the other side of the pipe is typically trivial to achieve, and most importantly scales linearly in terms of throughput. However, we may observe an additional latency due to the repeated ACK awaiting time [89], which could cause unexpected performance degradation if the message is split into a considerable number of segments.

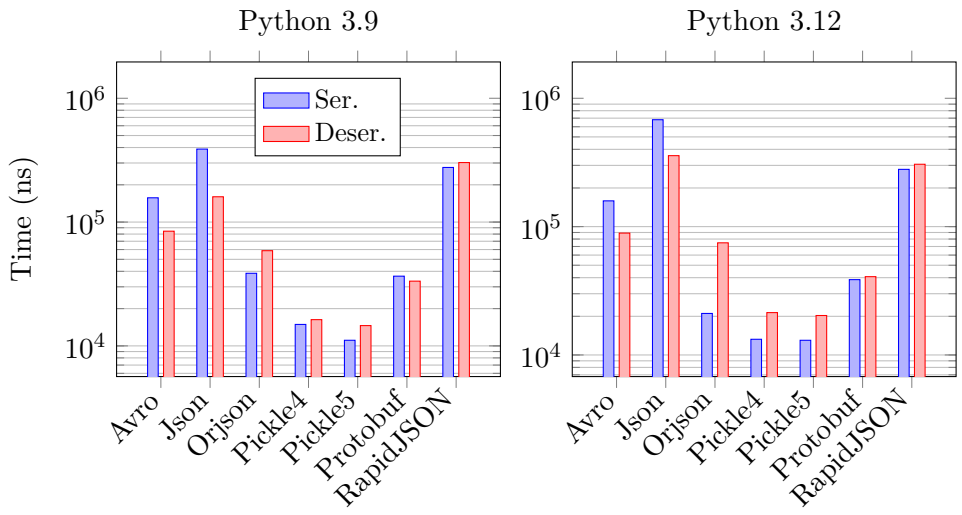

Figure 14: Serialization benchmark results for text-prevalent data types. The same remarks as Figure 14 hold for the plots in this image.

Multiplexed transport

A slight advancement with respect to the approach presented in the previous paragraph, is to leverage a multiplexed transport for the delivery of a massive message. We have already mentioned briefly the idea of multiplexing in Section 2.2. Multiple packets could be sent concurrently and independently, letting the transport layer manage the details. However, such an approach is not always possible: for instance, multiplexing was only introduced as part of HTTP/2, while all HTTP/1-based technologies are left behind [32]. This solution does not suffer from a systematic increased latency due to the ACK mechanics, but the responsibility of in-order delivery is delegated to the transport, which may or may not provide it. For example, gRPC guarantees in-order delivery of asynchronous streaming, unidirectional as well as bidirectional [61]. In the latter case, though, there is no correlation between the delivery time of messages that do not share the same source.

Compressing data

Data compression is a humongous topic by itself, and it is not among the objectives of the present work to provide a complete overview of the matter. It is extensively used in many software products as the basis for performant and reliable inter-

process communication [124]. The high-level idea is to *compress* a sequence of bytes, according to some algorithm which shall reduce the total number of bytes while retaining the original information, and then to *decompress* it on the other end of the pipe using the appropriate inverse algorithm. The wide family of compression algorithms is typically divided into the two following categories:

- *Lossy*: applying these algorithms yields an adequately accurate reconstruction of the original sequence of bytes, usually with a controlled error rate;
- *Lossless*: they yield an identical reconstruction [16]. They typically leverage the fact that all sequences of bytes contain *some* degree of redundancy.

We present a very simple lossless compression algorithm to fix the ideas. Given a sequence of N bits (assuming $N > 1$), such sequence shall be collapsed to a single i (i.e. i repeated only one time) if the sequence of bytes is equal to i repeated N times. In all other cases, namely if the sequence is not a single value repeating N times, the "compressed" form is identical to the original. This compression algorithm yields a $\frac{N-1}{N}$ compression rate in 2 out of 2^N cases. This naive example is also important to remark that all compression algorithms achieve optimal performance only on a limited domain.

3 Language interoperability in UCAP

After the theoretical discussion presented in Section 2, we now focus on the practical optimization of UCAP, leveraging the concepts discussed in the first part of this work. As we anticipated in Section 1, and in particular in Section 1.5, UCAP is considered a mission-critical software in the CERN controls system. For this reason, enabling efficient usage of Python user-defined functions in UCAP to perform transformations on acquired data is an important contribution from a practical point of view, especially considering the steadily growing popularity of the framework, and of Python as well, in the scientific computing ecosystem [43, 73, 138]. Moreover, we believe that the approach and the ideas developed in this work may be of interest for future efforts in similar directions, due to the generality of the requirements and the demand for similar architectures [87].

In the present section, we will first set the objective of the work in practical terms, focusing in particular on the metrics that we intend to improve. This will provide us with a concrete benchmark to showcase our results. In addition, we will also give some insights about the organization of the practical phase of the work, considering important deadlines as well as the number of changes that were introduced in the codebase. Then, we will present our plan to achieve the aforementioned objective, namely the software and architectural optimizations that we decided to operate on UCAP. We will also compare the novel approach with the previous one, highlighting the main differences and the strengths of our proposal. Finally, we will present empirical results to substantiate the validity of our statements, using meaningful metrics extracted from different snapshots of UCAP running in a production-like environment.

3.1 Practical details

In the following paragraphs, as we mentioned earlier, we will define clearly the practical objectives of the work. Moreover, we will also provide some details about the implementation work, namely the amount of effort that was devoted to the practical counterpart of the present document.

3.1.1 Objective definition

The high-level objective of this work in practical terms is to reduce the *latency* of event processing in UCAP, for which we mean the time elapsed between the arrival of an input and the emittance of the corresponding output. This is clearly an extremely relevant metric for end users, who expect their results to be delivered with the least possible amount of latency. UCAP is frequently used to trigger

Table 5: Operations involved in the execution of a Python user-defined function, starting from the reception of the input.

	Location	**Operation**	**Implementation**
I	UCAP node (Java)	Input serialization	Jackson
II	Transport	Serialized input delivery	HTTP/1.1
III	Python	Input deserialization	JSON→`dict`
IV		Transformation execution	UDF
V		Output serialization	`dict`→JSON
VI	Transport	Serialized output delivery	HTTP/1.1
VII	UCAP node (Java)	Output deserialization	Jackson

operations or other computations in a cascade sequence in the control system of the CERN accelerator complex, therefore reacting quickly is critical. So far the general recommendation which was given to the CERN community was to employ Python user-defined functions in UCAP only to leverage specific third-party libraries without equivalent alternatives in Java (e.g. PyTorch or Tensorflow), or to fulfill non-critical use cases that did not require low-latency responses. And this was not dictated by the alleged "inefficiency" of Python itself, but by a limitation of the UCAP framework. This limitation has constrained the spreading of UCAP in the CERN Python community. As we mentioned in Section 1, Python has lately become the de-facto standard in the scientific community, therefore such a limitation ought to be addressed.

In the case of Python user-defined function in UCAP, the latency is influenced by the components listed in Table 5. This is basically the set of operations that are needed to set up the input message to be delivered to the Python transformation process, then used to generate an output, which shall then be packed and sent back to the UCAP node. Note that in the current implementation of UCAP, operations I-VII are performed serially, meaning that each step must wait for all the previous ones to be completed [56]. We observed that the operations mentioned in Table 5 are the main drivers of the latency perceived by the users of the UCAP-Python integration. Therefore, for the sake of simplicity, our definition of latency does not take into account other UCAP components, even though they are on the critical path. Their contribution to the global latency is negligible with respect to the operations listed in the table, and they are driven by different dynamics which are not relevant for the present work. For this reason, all the numeric results in the present section will only take into account the longitudinal cut of UCAP related

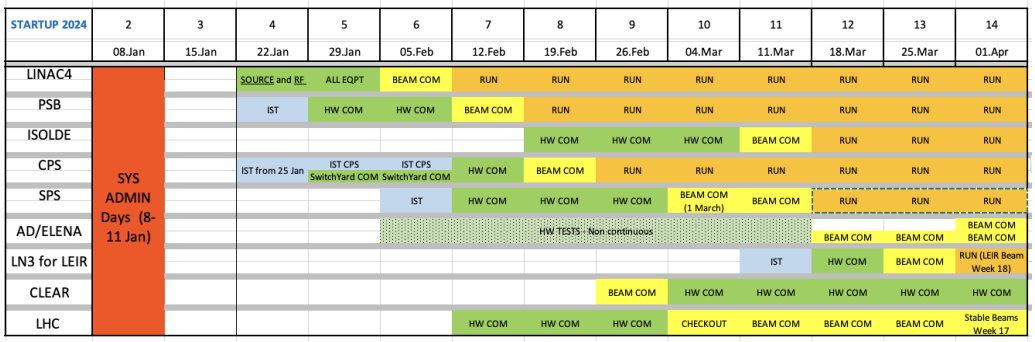

Figure 15: CERN accelerator restart schedule after 2023/2024 year-end technical stop. Critical software for the control system is expected to be fully available and stable since the first operational day of the first accelerator restarted (i.e. LINAC4).

to cross-language interoperability.

3.1.2 Timeline and code changes

The initial design of the new UCAP-Python integration started in July 2023, with a preliminary study of several serialization protocols for large numeric arrays. In such a context multiple alternatives have been tested, like Protobuf, Avro, JSON, and NumPy's proprietary format `npy`. After the study, we conducted a technology review to select the most appropriate third-party libraries to restructure the inter-language communication infrastructure in UCAP. After this phase, we decided to base our approach on Protobuf, gRPC, and NumPy. While the latter is not necessarily related to on-wire communication, it is used as a compatibility layer for bidirectional conversion from byte streams to typed arrays.

The high-level objective on which we agreed after the design phase included a release date in early January 2024. This was driven by practical requirements: in December and early January, CERN accelerators are generally not operational due to maintenance work. This period is usually referred to as *year-end technical stop* (YETS) [8]. Within this time window software can be upgraded without causing any disruption to operators and researchers, since there is typically no data to analyze. Indeed, YETS is generally exploited for tests on the machines and software updates [63]. However, starting from January 22nd, the CERN accelerator complex was scheduled to restart gradually, as shown in Figure 15. Thus, no software updates exceeding such a deadline would have been possible, until the subsequent technical stop in September 2024. This was particularly

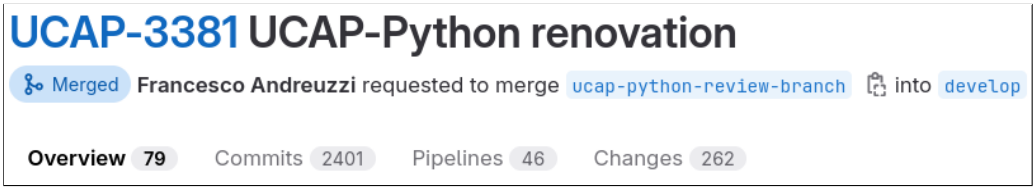

Figure 16: The new implementation of the new UCAP-Python integration was merged into the UCAP project in January 2024. It comprised 262 files changed/added, which makes it one of the most important changes in the UCAP framework since its inception.

compelling for UCAP, since it is considered a critical application for the CERN control system. Due to the importance of the update to enable new use cases to be tackled in UCAP with the assistance of the rich Python ecosystem, it was decided to commit to such a deadline.

The implementation started in September 2023, and it required important changes to the whole codebase. It touched both Java and Python code, for a total of *15.000* lines of code changed or added. Considering the end-to-end implementation work, including code additions as well as testing, the practical phase took about two months. The first working draft of the renovated UCAP-Python integration was ready for the team-wide review in December 2023. In parallel, we started testing the new UCAP-Python integration with several early-adopter users, which confirmed the expected performance improvements, as well as the stability of the new version. After several review iterations to assure quality and robustness, all the changes proposed have been accepted and merged into the main Git branch of the UCAP project (Figure 16). The deadline was fully respected, and the large-scale deployment of the new version started the week before the restart of LINAC4.

3.2 Language interoperability improvements

Following the plan presented at the beginning of Section 3, in the coming paragraphs we will describe in detail the improvements applied to UCAP as part of the present work. We divided the changes in three topics, the first two addressing explicitly the latency problems highlighted in previous sections, and the third improving the monitoring infrastructure in UCAP to enable better measurements at runtime.

```
{
  "parameterName" : "TST.NSOURCE1/BCT1137Aqn"
  "context" : {
    "acqStamp" : "1556539561425238525",
    "cycleName" : "LN4.USER.MD1",
    "cycleStamp" : "155659561150000000",
  },
  "data" : {
    "values" : {
      "intensity" : {
        "doubleArray" : {
          "doubles" : [ 0.570054365214917, -0.5145501148489071,
                        -0.569365030978513, -0.5693650309785132 ]
        }
      },
      "currentLinacSingle" : {
        "double" : -45.336858408856926
      }
    }
  }
}
```

Listing 13: JSON example of UCAP input/output data.

3.2.1 Zero-copy binary message exchange

The first improvement concerns the format of the messages exchanged between the UCAP node (Java) and the Python conversion process. As we briefly mentioned in 1.4 the previous implementation relied on JSON (see Section 2.3) to exchange text-based messages between the two parties involved. Listing 13 displays a simplified example: the object holds two values of type `doubleArray` and `double`, named respectively `intensity` and `currentLinacSingle`. A serialized message of this kind is quite simple to obtain by using standard tools like Python's `json` module, or Java's `Jackson` library. Lists of numeric types are handled internally, as well as maps of key/value pairs. This kind of format provides several benefits, specifically those which we have already mentioned in Section 2.3. As a reminder, this kind of format guarantees ease of interpretation for a human – which is handy at debugging time, especially in a microservices-based environment like UCAP [103] – and extensive tooling for any language in existence. However, as we anticipated in Section 2.3, employing such a format in production may bring in several problems mainly related to its construction and parsing. Indeed, JSON is quite verbose, thus messages may travel significantly slower on the wire, and most importantly a computer needs more time to serialize and deserialize them. The reader may refer to the aforementioned section for more details. These problems manifest in particular when we intend to transmit large arrays, which is one of the most common

```
message Data {
    map<string, GenericArray> values = 1;
}
enum ArrayType {
    BOOLEAN = 0; BYTE   = 1; SHORT  = 2; INT     = 3;
    LONG    = 4; FLOAT  = 5; DOUBLE = 6; STRING  = 7;
}
message GenericArray {
    bytes payload = 1;
    ArrayType type = 2;
    repeated int32 dimensions = 1;
}
```

Listing 14: Protobuf schema definition of UCAP input/output data values.

uses in UCAP and in data processing platforms in general. Generating JSON documents requires extensive usage of string-concatenation, which happens to be rather inefficient in languages with immutable string objects [110, 155]. This is an observation of particular concern since unnecessary copies are often designated as the most critical performance issues in data transfer [112, 136].

This is why we decided to switch to a different format that does not require additional copies, namely a binary representation. Such a decision requires giving up the possibility to visually inspect the serialized messages for debugging purposes; however, we acknowledged that the advantages would largely overtake this renouncement. Among the alternatives that we have considered in Section 2.3, we selected Protobuf due to its extensive documentation and active community. In Listing 14 we show a simplified definition of the body of an equivalent message to that which we have shown in Listing 13. It essentially consists of a mapping from `string` objects (i.e. keys) to `GenericArray`. Each possible value fits into the definition of `GenericArray`, which is defined as shown in Table 6. However, the Protobuf representation shall be considered internal, what is usually called *implementation detail* in software engineering, and should not be exposed to end users. This encapsulation is mainly required to preserve backward com-

Table 6: Definition of the Protobuf type `GenericArray`, which represents any numeric or string array including scalars.

Field name	Type	Definition
`payload`	`bytes`	Content of the array represented as bytes.
`type`	`ArrayType`	Type of the elements of the array.
`dimensions`	`int32[]`	Shape of the array.

Table 7: Triplets of equivalent types in the Protobuf definition, Java and Python. Python types are all defined in terms of NumPy's `dtype`s.

`ArrayType`	Java type	NumPy type
`BOOLEAN`	`bool`	`np.bool`
`BYTE`	`byte`	`np.int8`
`SHORT`	`short`	`np.int16`
`INT`	`int`	`np.int32`
`LONG`	`long`	`np.int64`
`FLOAT`	`float`	`np.float32`
`DOUBLE`	`double`	`np.float64`

patibility of all pre-existing user-defined functions. Therefore, a conversion layer is needed on both sides, Java and Python, to convert to and from the format used on the wire [87]. In Table 7 we listed the equivalent triplets of types in all three parties involved. A similar table is programmatically implemented both in Java and Python to streamline the conversion. We decided not to use Protobuf standard types as values of the field `GenericArray#type` because those values cannot be used like an enum, thus we had to define our own. Moreover, we took the important design decision to represent values as `bytes` fields, rather than `repeated float32`, `repeated int64`, etc., because this would require defining multiple message types, one for each supported value. The decision to separate the content of the array from its type allowed for important simplifications in the Python implementation of the conversion layer, since NumPy allows creating arrays with parametrized types via the `dtype` parameter of the factory function `np.array()` [65].

The conversion layer is the most critical part of the new architecture. With JSON-based messages, the implementation was quite straightforward and involved filling a dictionary of key/value pairs for all the values in the body of the message. The generation of the actual JSON document could then be delegated to the library of choice, as we discussed above. However, the introduction of a binary data format in the pipeline provides an important opportunity for performance improvements. Indeed, most languages and libraries allow obtaining a byte representation from an in-memory array. This means that we could in principle replace the expensive string concatenation of the previous implementation with a constant-time (or linear-time at most) reference assignment. In Table 8 we list all available methods in Java and Python/NumPy for bidirectional conversion between a standard byte array (`byte[]`) and a buffer of bytes which can be assigned

to a Protobuf $\texttt{bytes}_\text{p}$ field, in our case to `GenericArray#payload`. The trailing column states whether the given operation requires a full copy of the object or not. An additional copy operation may be quantified as an additional $O(N)$ contribution to the overall complexity of the operation, N being the length of the array [28]. Copy-less operations, by contrast, contribute as $O(1)$ terms, since they are implemented as a simple reference assignment.

Efficient ND array representation

So far we have considered one-dimensional array structures. However, according to the interface which is was agreed upon with initial UCAP stakeholders, developers of user-defined functions are allowed to employ multi-dimensional arrays. This is mainly dictated by conventions for message data types set by CERN's internal data exchange software [6, 39, 90]. In the previous implementation of the UCAP-Python integration, this additional requirement was implemented by flattening the array (i.e. converting it to a 1D sequence) before serialization, and then restoring the original shape after deserialization to present an appropriate representation of the user. This resulted in two additional $O(N)$ contributions to the overall computational cost for the data exchange. The same process was replicated in reverse, to deliver the output of user-defined functions to UCAP's Java main process [29]. By exchanging byte streams instead, and basing the language-specific serialization/deserialization layers for Java and Python respectively on `java.nio.ByteBuffer` and NumPy arrays, we provide a much more performant implementation. NumPy provides a `reshape()` method, which computes a cheap view on the array [65]. Internally, it consists of simple assignments of indexes and strides, and the underlying buffer is untouched; therefore, `reshape()` is expected to contribute as a $O(1)$ term. By contrast, `ByteBuffer` does not provide a cheap view mechanism, but its collective `get()` method is extremely optimized [110]. Indeed, the developers of the standard library were aware that this method may be a rather CPU-intensive area of the code; thus, it was marked as a strong candidate for JVM just-in-time compilation [44].

Differences between Java and Python implementations

Protobuf is a wide project, and its implementations in all supported languages may be slightly different for what concerns implementation details, even though they are fully equivalent in terms of functionalities. In a few cases this is driven by the environment, like limitations of the language. Looking at Table 8 the reader may spot two important discrepancies:

- Java does not allow copy-less conversion from Protobuf `bytes` to an array

Table 8: Bidirectional conversion between byte arrays (`byte[]`) and Protobuf $\texttt{bytes}_\text{p}$ objects. The subscript "p" in $\texttt{bytes}_\text{p}$ indicates that the object is a Protobuf `bytes` field.

Language	Purpose	Operation	Copy
Java	`byte[]` → $\texttt{bytes}_\text{p}$	`ByteBuffer.wrap(byte[])`	×
	$\texttt{bytes}_\text{p}$ → `byte[]`	`ByteBuffer.get(byte[])`	✓
Python/ NumPy	`byte[]` → `memoryview`	`np.ndarray.data`	×
	`byte[]` → $\texttt{bytes}_\text{p}$	`np.ndarray.tobytes()`	✓
	$\texttt{bytes}_\text{p}$ → `byte[]`	`np.frombuffer(…)`	×

of the primitive type `byte` (or any other Java numeric or string type). In Python, this is implemented using NumPy's `frombuffer()` function, which returns an immutable NumPy array wrapping the buffer. In Java immutable arrays do not exist (unless one looks into the Collections framework, and is willing to accept the prohibitive performance penalties of *boxing* [110]), therefore this could be considered a limitation of the language.

- NumPy's `ndarray` provides two ways to obtain a buffer-like object, namely `np.ndarray.data` (which returns a `memoryview`) and `np.ndarray.tobytes()` (which returns a `bytes` object). The first is copyless and returns a mutable value, while the second makes a copy of the array and returns an immutable object. Unfortunately, the Python implementation of Protobuf does not accept `memoryview` objects due to its mutability, since any of the owners of the reference may still modify the buffer after the assignment before the message is written to the wire, which would be the cause of extremely nasty bugs. For this reason, our serialization layer in Python makes an unnecessary copy of the array. This limitation does not exist in Java, since the user is allowed to circumvent the problem using the function `UnsafeByteOperations.unsafeWrap(ByteBuffer)`. However, we expect that a future version of Protobuf removes this constraint; the discussion on this topic has started already, as we are going to mention in Section 4.

Performance and size benchmark

Having treated the theoretical aspect of the novel serialization method proposed as part of this work, we complete the discussion and the comparison with the formerly employed method with some brief measurements on two meaningful metrics.

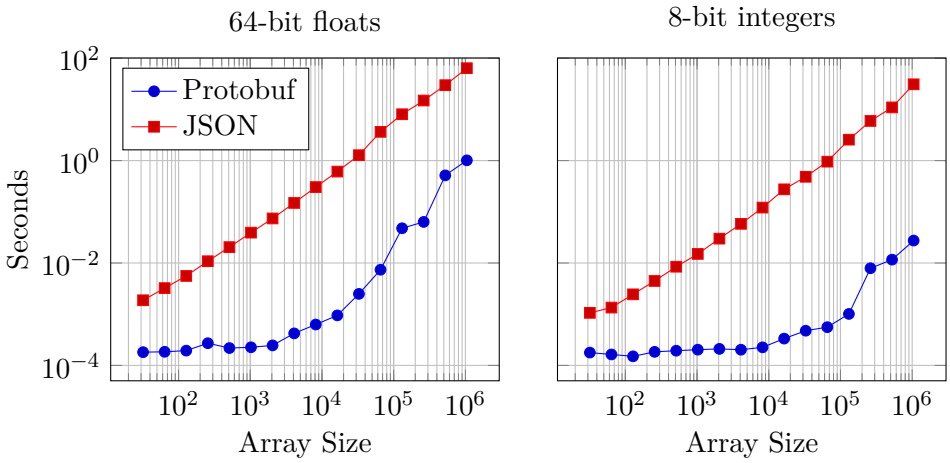

Figure 17: Performance comparison of a message round trip (i.e. serialization/deserialization) implemented with Protobuf and JSON. We considered two data types, for NumPy arrays of varying sizes. Both the horizontal and vertical axes are on a logarithmic scale. In both cases the Protobuf-based approach outperforms the alternative, and small data types (right) are faster to process due to the fewer number of bytes.

These metrics are representative of the performance that will be perceived by end users as part of the global latency per message. The first is the time in seconds it takes to complete a round trip, namely a full pipeline of serialization and deserialization; the second is the in-memory size of the serialized message. The experiments aim to demonstrate the gains of the Protobuf-based approach in isolation; we will report more thorough experiments in a production-like environment in the coming sections. We consider a simplified case, namely the serialization of a single array-like value. This is representative of the most problematic case that shall be handled in UCAP, since other fields that may compose the message (e.g. those listed in Listing 13) are trivial to include with negligible performance degradation. For the sake of simplicity, we implemented the whole round trip in Python, but the same experiment could be conducted on the Java version with comparable results.

The first experiment is conducted as follows: given NumPy arrays of varying size (we employ powers of 2 for simplicity), we measure the time it takes to serialize each of them employing both the algorithm which is currently implemented in UCAP, and that proposed in Section 3.2.1. The results are visualized in Figure 17. The plot on the left of Figure 17 shows the results for arrays of 64-bit floating-

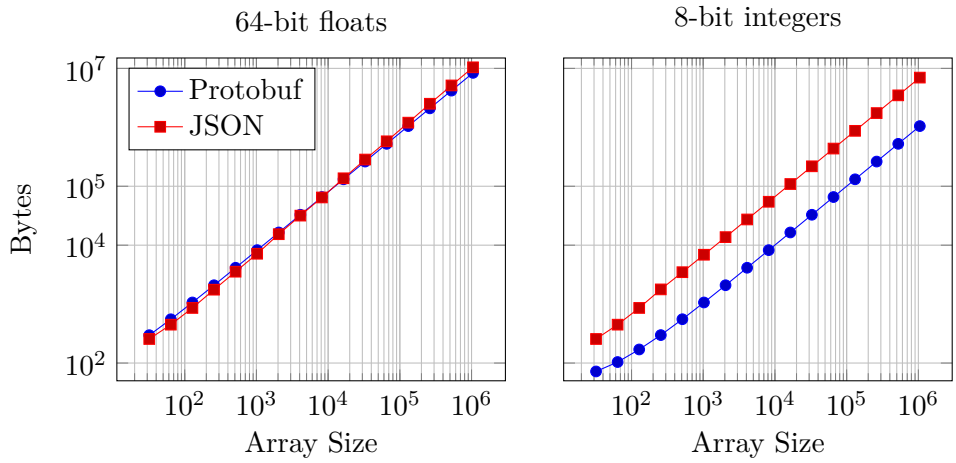

Figure 18: Comparison of the serialized message size produced by both methods considered, with two different data-types, for NumPy arrays of varying sizes. Both the horizontal and vertical axes are on a logarithmic scale. The message size is comparable in the floating-point case (left), while for small data types (right) the Protobuf-based message is smaller by approximately one order of magnitude.

point values, visualized in seconds. The Protobuf-based method outperforms the competitor consistently by approximately two orders of magnitude, which for the largest array considered in the experiment makes a difference of 1 second against 100 seconds. In the plot on the right of Figure 17 we show the results of a benchmark that takes into account a smaller data type, namely an 8-bit integer, keeping the same experimental structure. While the overall growth of both curves is quite similar to the previous experiment, the plot uncovers an important characteristic of the novel serialization/deserialization method proposed for UCAP. Indeed, the Protobuf-based approach is sensitive to the size of the data type. In practice, this means that employing a smaller data type provides a significant performance improvement. This is somehow expected since manipulating fewer bits is supposed to be more performant. However, this behavior did not manifest in the JSON-based approach, due to the fact that the data type information was entirely discarded in the internal representation. In the 8-bit case, we observe that the Protobuf-based approach outperforms the competitor by approximately three orders of magnitude.

In Figure 18 we compare the size of the message produced by both methods in the same context as that which we described for Figure 17. The in-memory size of generated messages is an extremely important metric to evaluate for serialization

protocols, as it is the main driver of the transportation speed within a network. This is something to take definitely into account, especially in the context of scalability [87]. The experiment is conducted as follows: we first serialize each array to Python objects (strings for JSON, and `bytes` for Protobuf); then, we measure its in-memory size with Python's standard `sys.getsizeof`. While measuring the size of a sequence of bytes does not need much discussion, there is more uncertainty in the case of JSON messages. The main question basically concerns the encoding employed to create the string, as this is a critical factor for the message size. Python uses UTF-8 strings by default [155, 160], unless there is a specific need for a different encoding. For instance, the user may enforce a specific one for compatibility with another service. In UCAP, all parties implicitly agree on using UTF-8, therefore we conducted the experiment using such encoding. UTF-8 is a variable-width encoding which enables some optimization by itself, especially if the string does not contain exotic characters. Moreover, all numbers are encoded as strings, thus they only take the exact number of "characters" which they actually need. On the contrary, byte streams require a fixed amount of bits for each item. Therefore we may expect to observe some balance between the two strategies. Indeed, this is the case in the plot on the left side of Figure 18: JSON-based and Protobuf-based messages occupy approximately the same amount of space in memory. One should also take into account that the well-known verbosity of the JSON format does not impact significantly the case under examination, since the number of fields in the object is constant in the plot. Therefore, for large numbers, the size of the message is dominated by the sole encoding of the array, rather than inter-field punctuation. However, the results in the case of cheaper data types like 8-bit integers (right side of Figure 18) testify that using byte-based formats is in general much cheaper. The advantages obtained by leveraging UTF-8 optimizations are negligible with respect to a reduction of an 8x factor of the number of bits for each element.

Extending the approach

The serialization approach presented in this section is easily extendable to other languages. In the current implementation, the method is used to achieve Java-Python communication; however, the code is already in place to implement Java-Java communication. This will be the base for future improvements of UCAP, which we are going to mention also in Section 4. Moreover, support for a new programming language would be effortless to introduce, since the theoretical foundation is already laid out. This is of course conditioned to whether such language is supported by Protobuf or not. As of early 2024, the framework officially supports C++, C#, Dart, Go, Java, Kotlin, and Python [59].

Related work

The approach presented in the last few paragraphs shares similar ideas with the implementation of `pickle` 5th protocol [121]. `pickle` is Python's standard serialization/deserialization module. It is the base for popular Python modules like `multiprocessing` (Python's built-in concurrency library) and Dask (distributed linear algebra), which rely on it to represent in-memory Python objects to be shared between parallel workers [7]. The idea of `pickle 5` is quite similar to that which we have outlined in the previous paragraph, namely passing a reference to the serialization engine a reference – rather than a full copy – pointing towards a view of in-memory data. The same concept has spread in other projects requiring efficient serialization/deserialization of objects supporting the Python buffer protocol [167], like Apache Arrow [120]. We could not use `pickle` as the base for our serialization layer (Python-side) because the pickle format is not compatible with the transport layer that we decided to use (gRPC), and most importantly there is no equivalent deserializer in Java. A peculiarity of `pickle 5` is that serializing a buffer-like object and deserializing it in the same process will provide an object pointing to the same memory location as the original one. This feature is not required in UCAP, since serialization and deserialization will always occur in two different languages. Developers of Dask considered opportunistic compression as part of the serialization depending on the specific use-case [7], which is something we may introduce as well in UCAP in case we observe the emergence of use cases with extremely large-scale datasets.

3.2.2 Pipelined asynchronous UDF execution

As we outlined in Section 1.4, the current implementation of the UCAP-Python integration relies on fully blocking interoperation with a dedicated Python interpreter process that handles the user-defined function it is associated with. In other words, all the operations listed in Table 5 must be completed sequentially before the next input can be processed. This design choice is critical as it enables achieving the following requirements:

- Outputs must be delivered in order, as agreed with end users of UCAP;
- User-defined functions should receive inputs in order, to avoid a non-deterministic state in stateful functions. Ideally, the state should be reproducible uniquely by replaying all past inputs, similar to implementations of the *event sourcing* architectural pattern [87].

The goal of the work presented in this chapter is to enable multiplexed delivery of inputs/outputs by leveraging the HTTP/2 protocol [32], and to parallelize

message exchange with the execution of Python user code, following the ideas presented in Section 2.1.4).

We divided the discussion into three paragraphs to split the topics according to the UCAP domain they belong to. In the first paragraph, we will present the gRPC service definition, which is the central driver of the subsequent language-specific implementations; we will then discuss the changes required in the Java-based side of UCAP, namely that which is responsible for acquiring inputs and orchestrating the execution of user-defined functions; finally, we will briefly present the new implementation of the Python transformation server, according to the gRPC service definition which we mentioned earlier.

Cross-language interoperability protocol

The new implementation of the UCAP-Python integration employs gRPC – a mature open-source framework developed by Google Inc. – for the communication between Java and Python. gRPC sits on HTTP/2 for the transport and on Protobuf for the serialization of messages [61]. We discussed message-passing IPC with more details in Section 2.2.2. For what concerns this section, gRPC replaces the previous fully-blocking communication via blocking HTTP calls. The interaction modality between client and server – which in our case are respectively the UCAP master process and the Python transformation process – is dictated by a `proto` file named *service definition*. The service definition defines all available RPCs for a given service, including the arguments and the returned value. A simplified version of our service definition is displayed in Listing 15. The definition translates in simple words to the following protocol: a UCAP master process may initiate a `convert` RPC by opening a stream of `Data` representing a potentially infinite sequence of input values. The server is expected to pick up inputs from this stream in order and to reply to each of them with the corresponding output of the user-defined function. The two streams are fully independent, in the sense that gRPC does not enforce any relation between any pair of of input and output.

```
1 service ConversionService {
2   rpc convert(stream Data) returns (stream Data);
3   rpc metrics(google.protobuf.Empty) returns(stream MetricsMsg);
4 }
```

Listing 15: Simplified gRPC service definition for UCAP-Python integration. The `convert` RPC is responsible for triggering an execution of the appropriate user-defined function, while `metrics` initiates a request to continuously pull metrics values from the server (see Section 3.2.3

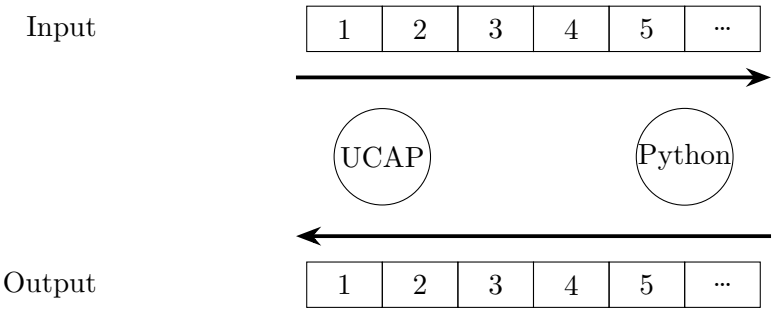

Figure 19: Expected appearance of the UCAP input/output flow, outputs should respect the absolute order imposed by input acquisition. The same behavior should be respected by the new implementation of the UCAP-Python integration.

The only guarantee provided by gRPC bidirectional streaming is that messages on a given stream are delivered in order [61]. It is then up to the client and server to enforce the order of outputs to reflect the order of inputs. The expected situation is described schematically in Figure 19. To summarize, the important paradigm change we introduced in UCAP is that the communication with the Python process is now implemented by two independent streams of data. The way each of these streams is managed is the topic of the next two paragraphs.

Transformations orchestration and in-order results collection

Transformation orchestration is the mechanism of the UCAP master process that extracts an acquired input from the in-memory queue and delivers it to the appropriate Python process for the execution of the user-defined function. The previous implementation was quite straightforward: a given work item could be scheduled for transformation either upon arrival – if the corresponding user-defined function was being idle – or just after processing the previous input. While this approach is quite robust, it leaves evident room for performance improvements. Instead, in the new implementation we allow a configurable number of work items to be *in-flight* at the same time, meaning that they are anywhere in the execution pipeline (input serialization, user-defined function execution or network channel). Since new work items are acquired in a multi-threaded context, an appropriate synchronization mechanism is needed to make sure that no more in-flight work items than the maximum allowed are scheduled for execution at the same time. This is guaranteed in our implementation by the use of `Semaphore`, a common synchronization primitive which is provided by Java's standard concurrency library (for more details see Appendix B) [56, 110]. Each transformation channel receives a

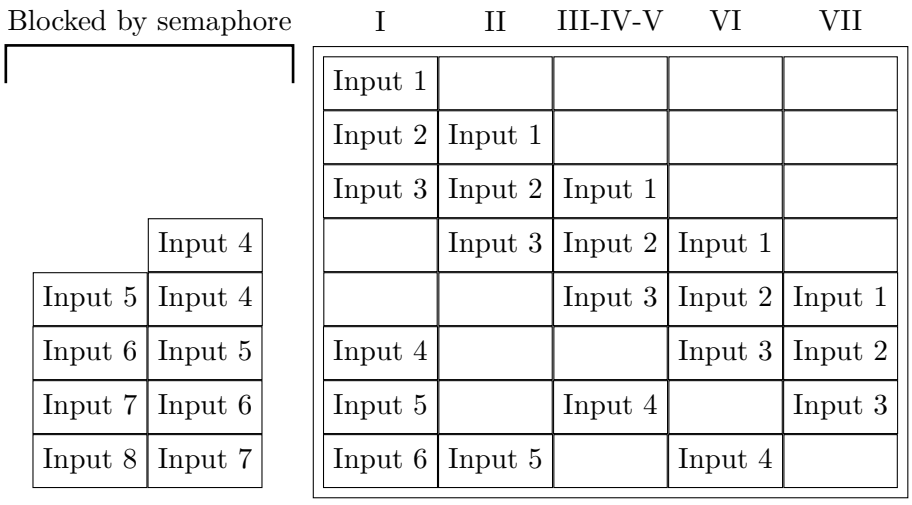

Figure 20: Example diagram of the pipelined execution of UCAP-Python transformations. Each row represents the pipeline configuration in a different instant. To simplify the schema we assume all steps have the same duration. The maximum number of in-flight work items is 3. Steps are taken from Table 5.

dedicated semaphore, which is initialized with a number of permits equal to the maximum number of in-flight work items. It is important to enforce this limit, since as we are going to see in a few paragraphs it is not guaranteed to observe a throughput improvement by allowing more work items to be processed at the same time. Moreover, having many in-flight work items would increase significantly the contention on the multiplexed HTTP channel, which eventually could even damage the throughput [32, 61]. The main reason for introducing the possibility of treating more work items in parallel is the exploitation of pipelining (see Section 2.1.4). Indeed, we may hide the cost of exchanging the next input value on the network behind the cost of running a user-defined function for the previous input, with a consequent increase in the throughput. The situation is described with an example in Figure 20.

Python UDF execution

The implementation of the Python gRPC server is quite simple, as the RPC is executed on a single thread. The method implementing the `convert` RPC receives an iterator which it should iterate on to get the new available inputs coming through the stream. It should then deserialize the input, execute the user-defined function, and finally serialize the output. The value is then injected

Table 9: Assumptions taken during the design of the UCAP threading model, with the corresponding motivation. By "(de)serialization takes the same amount of time" we mean the duration of the operation is the same both on Java and Python.

Domain	Assumption	Motivation
Channel	Communication takes equal time in both directions.	UDFs generally return values with comparable size to the input.
Python/ Java	Serialization takes the same amount of time.	Protobuf is used in both domains.
	Deserialization takes the same amount of time.	Protobuf is used in both domains.
	Serializations takes twice the time taken by deserialization.	Empirical observations.

into the output stream via the `yield` keyword. A simplified version is shown in Listing 16.

Synthetic performance benchmark

In order to predict the possible performance gains achieved by the introduction of the ideas presented in the last few paragraphs, we implemented a simplified model representing UCAP transformation scheduling and cross-language communication. The model is implemented in Java, due to the the extensive usage of threads we made to simulate faithfully the UCAP threading model. Python was not an alternative due to the Global Interpreter Lock (GIL) [155]. Note that the

```
async def convert(
        self,
        request_iterator: AsyncIterator[Data],
        context: grpc.aio.ServicerContext,
    ):
        async for request in request_iterator:
            input_value = deserialize(request)
            output_value = user_defined_function(input_value)
            yield serialize(output_value)
```

Listing 16: Simplified implementation of the gRPC endpoint in UCAP-Python which handles the execution of the user-defined function upon acquisition of an input.

Table 10: Components of the UCAP threading model, with the corresponding tasks to be performed for each run.

Component	Steps
Input producer	Produce a new work item Append the item to the input queue
Input consumer	Extract a work item from the input queue Serialize the work item Send the serialized message via *Communication channel*
Communication channel	Deliver the message Trigger message consumer (*Python executor*/*Output receiver*)
Python executor	Deserialize input Execute UDF Serialize output Send the serialized message via *Communication channel*
Output receiver	Deserialize output Schedule *Input consumer*

model has the sole purpose of predicting the throughput, but it is very general and is not based on any UCAP or CERN-specific information. The model is also based on the simplifying assumptions listed in Table 9, which aim at making it easier to experiment with while keeping it as realistic as possible.

The components of the threading model are listed in Table 10, the reader may notice that these are the same players that we have already mentioned in Table 5. The details about the Java implementation of each component are listed in Table 11. Note that we dedicated only one thread for the Python executor. This is driven by the gRPC threading model [61]: each incoming request is handled by a single thread from its arrival to the serialization of the response. This poses a serious limitation to the parallelization of communication and execution of user-defined functions, as the Python executor becomes effectively a bottleneck. It is generally possible to mitigate this limitation by leveraging pipelining [62], provided that the duration of the bottleneck step is similar to the duration of all the other steps.

In our case, the bottleneck step amounts to 1.5 serializations, plus the user-defined function execution, keeping in mind our simplifying assumptions. We take as a reference an array of 16384 64-bit floating-point items, whose in-memory size amounts to exactly 1 Megabyte. According to the measurements we presented

Table 11: Implementation in Java of the components of the UCAP threading model listed in Table 10.

Component	`java.util.concurrent.Executor`	N. threads
Input producer	`newScheduledThreadPool`	1
Input consumer	`newFixedThreadPool`	Parameter
Communication channel	`newFixedThreadPool`	Parameter
Python executor	`newSingleThreadExecutor`	1
Output receiver	`newCachedThreadPool`	n/a

in Figure 17, a round-trip (serialization and deserialization) takes approximately 0.9 milliseconds, thus we estimate serialization to take 0.6 milliseconds and deserialization to account for the residual 0.3 milliseconds. According to gRPC benchmarks, sending 1 Megabyte back and forth between a client and a server located in the same host takes approximately 6 milliseconds in total [128]; for this reason, we estimate 3 milliseconds for each step on the communication channel. We consider a fixed message arrival interval of 10 milliseconds, and we average the measurements on 20 input/output pairs. The throughput results (in outputs produced per second) for this case are displayed on the left side of Figure 21. It is quite evident that introducing more input parallelism (i.e. the ideas discussed a few paragraphs above) pays off only when stepping from 1 to 2 workers. All further steps do not improve the results by an appreciable amount. This happens because the pipeline includes a long single-threaded step in the Python executor, which for most values considered for the duration of the user-defined function dominates the other steps in terms of execution time, thus damaging the pipeline throughput. The situation is much more balanced if we consider a slow network that takes longer to deliver 1 Megabyte messages, for instance 40 milliseconds. This might be the case for example if the UCAP master process and the Python executor lived in two different data centers at a distance of several hundred kilometers [33]. This second case is depicted on the right side of Figure 21: the benefits of increasing input parallelism are much more noticeable in this setting. In most of the cases considered, spawning 4 or 5 parallel workers to handle inputs enables reducing latency per message by as much as 80%. In general, one should analyze the specific use case and obtain some relevant numbers to infer the appropriate spot in the grid. This is indeed a well-known best practice in the design of distributed systems [87]. Introducing input parallelism increases slightly the code complexity, which in the long term may even affect its evolution rate if the component is perceived as too obscure. If the situation under examination

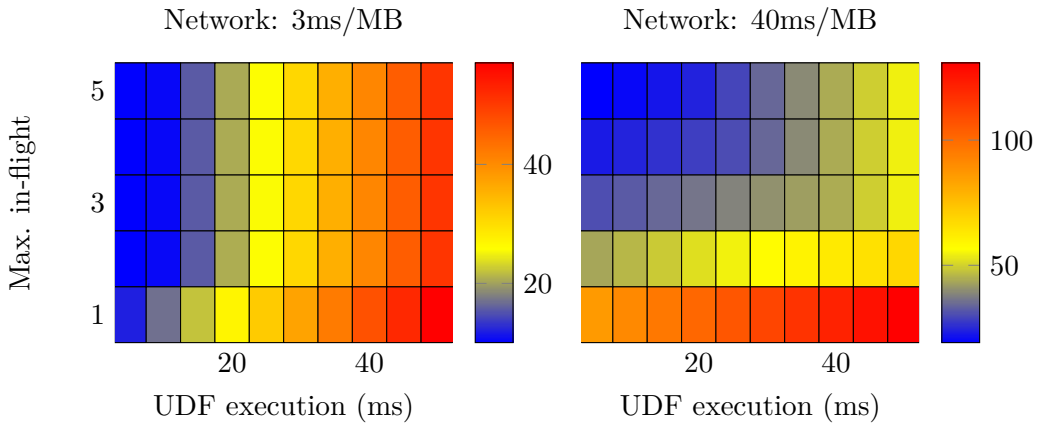

Figure 21: Overall latency per message (time elapsed between input acquisition and output publication), considering input messages of 1 Megabyte emitted every 10 milliseconds, on a channel that takes 3 (left) and 40 (right) milliseconds to send 1 Megabyte (one way) with serialization and deserialization time which amount respectively to 0.6 and 0.3 milliseconds. The duration of the execution of the user-defined function and the maximum number of in-flight work items vary.

was that depicted by the left side of Figure 21, a development team could even reconsider introducing the change in the codebase; the most obvious choice would be to simplify the implementation to always allow only a single level of parallelism. On the contrary, the case depicted on the right side justifies the additional maintenance effort thanks to the improved throughput.

3.2.3 Monitoring infrastructure

Monitoring is only marginal to the work presented in the present document, since it does not directly affect the performance of the software. Nonetheless, it is common knowledge that putting in place a granular metrics coverage is critical in complex software for several reasons [93]. First of all, monitoring may assist in spotting bugs: for example, UCAP internal components expose metrics that enable computing the number of inputs that are lost in the processing pipeline, e.g. because a Python transformation process did not reply after some deadline. Secondly, metrics are useful to discover performance bottlenecks that may affect the overall throughput: for instance, an undersized blocking queue, or an under-resourced thread pool executor. Finally, metrics are often used to verify whether the software is misbehaving, and if a human should be alerted to inspect the problem [153]. We will discuss the topic of observability and its role in enterprise

software with more details in Appendix A. We will also rely on monitoring to evaluate the improvements we have introduced in the UCAP-Python integration. Monitoring is more suitable than synthetic experiments for such purpose, since enterprise software contains generally much more components than those which are the target of the measurement. These components have an important impact at runtime and may disrupt completely the performance that may be promised by canned experiments [87]. For this reason, a realistic production-like set of experiments is needed to estimate the real new performance of a system after an important update.

UCAP is a complex software, and its distributed nature (see Section 1.4) makes monitoring even more critical than it is for standard client-server applications [87, 114]. For this reason, each UCAP node exposes more than 300 basic metrics to enable gathering insights about its runtime behavior [29]. In addition, dedicated metrics are exposed for each input source (i.e. health of the source, number of inputs emitted, timestamp associated with the last emitted value), and each UCAP user-defined function (i.e. average execution time, number of user code errors detected). This infrastructure is extremely valuable and provides a pervasive safety net for discovering production threats as fast as possible. However, the former implementation – which was based on JMX (see Appendix A) – turned out to be extremely costly in terms of runtime resources. A single metrics scrape – namely a query to the software to retrieve all exposed metrics – used to take a few tens of seconds in the worst case. This turned out to be way too long on several occasions, even resulting in false positive alarms for productional UCAP instances. Moreover, each scrape used to impact significantly the software runtime in terms of CPU usage, garbage collector behavior, and heap memory usage.

In the following paragraphs, we will briefly discuss the changes we have operated in UCAP to improve metrics acquisition. Our enhancements enabled more fine-grained metrics updates by making it feasible to scrape metrics several times per minute. This is critical for the following of our work, in order to enable the collection of more precise results during experiments aimed at evaluating the new UCAP-Python architecture. Then, we are going to present several metrics that we believe to be good indicators of the performance of cross-language interoperability in UCAP. In Section 3.3 we are going to present some numbers related to these metrics.

Metrics exposure strategy

As we mentioned in the introduction to Section 3.2.3, UCAP used to expose several metrics before the beginning of our work. These metrics aimed in particular at

monitoring the number of active transformations and incoming events. However, the internal collection and exposure were based on Java Management Extensions (JMX) (see Appendix A). This technical choice, which is quite practical and popular, has several important drawbacks (see also Appendix A). To summarize, JMX makes retrospective analysis with standard tools (e.g. PromQL) much harder to perform, especially when the objective is to aggregate multiple metrics of the same kind. For instance, summing the count of outputs emitted by all Python transformations deployed on a particular UCAP instance was implemented by manually aggregating the relevant intermediate results, rather than following the standard approach of summing tagged contributions in the Prometheus server. This increases the number of metrics exposed, the memory footprint, and most importantly the code complexity. Moreover, each metrics scrape – namely the collection of all metrics in a given instance of the software [93] – is most likely going to require an important amount of time to be completed, thus making the collection of granular metrics at short intervals unfeasible.

Ideally, metrics should be collected at intervals that could be considered "reasonable" according to the change we may expect to observe. In the context of our renovation, we set the goal of a 10 seconds period between each scrape, which is also adopted by software similar to UCAP [163]. In order to solve all the problems mentioned above and achieve our scraping goal, we decided to replace a huge portion of our JMX metrics with meters from the standard framework Micrometer [72]. We focused in particular on the dynamic ones, meaning that they are expected to change frequently. This required replacing all exposed JMX endpoints with Micrometer meters of an adequate type (either gauge, timer, or counter), and reviewing all metrics to identify those which may be dropped in favor of more standard tools (e.g. like aggregation on the metrics server via PromQL).

Heap memory usage

The amount of heap memory used by a Java application is in general the sum of all objects allocated since the JVM start-up, minus those that have already been collected by the garbage collector [44]. Allowing a certain software to operate on high memory usage may be problematic for several reasons. First of all, it may be a symptom of a memory leak, or, even worse, the constantly high memory usage could hide a memory leak, which could manifest as a catastrophic JVM failure at any time [106]. Secondly, requiring an important amount of memory to be allocated for a single application may be constraining from a deployment point of view. This is particularly problematic if the software is deployed on bare metal machines, in a multi-residency setup with other applications: allocating

enough memory for all parties may pose an important problem that requires non-conventional sub-optimal solutions [125]. At the time of writing this document, UCAP is indeed following this pattern (see Section 1.4 for more details).

We observed memory problems quite frequently at runtime with the previous implementation of the UCAP-Python integration. We expect the amount of memory used to be tightly coupled with the size of the messages being treated; however, we frequently experienced situations that escaped this pattern. This was caused by the important amount of intermediate objects allocated during serialization and deserialization, like JSON documents or intermediate string arrays (see also Section 3.2.1). The temporary solution which was frequently adopted, following a discussion with the user to understand the situation, consisted of raising the amount of heap memory that the JVM was allowed to use (via the startup parameters `Xms` and `Xmx` [110]). We expect to observe a much more stable behavior after the improvements presented in this work, and most importantly a lighter requirement for human interaction with operational UCAP instances.

Metrics about heap memory usage are available via simple calls to Java's `Runtime` facility (e.g. `Runtime.totalMemory()`). However, it is quite uncommon to extract these values directly, since many tools exist which provide better insights and aggregation over time. For example, Micrometer provides default registries that collect extensive information about the status of the JVM [72]. Similarly, direct JVM inspection tools (e.g. VisualVM, JProfiler) provide interactive views to monitor the status of the software at runtime, which is less useful to collect and analyze metrics but is extremely valuable to investigate malfunctionings in specific instances.

Queue residency

The duration of the round trip between Java and Python processes in UCAP accounts for an important portion of the latency. However, the waiting time for an input received but not yet processed is at least as important. In a situation of high load on the UCAP node, the input queue for a given transformation may be overloaded with messages to be processed. In such cases, the user will experience for each request a response time which comprises also the processing time for each previously received input not yet processed at the reception instant. This is due to the serial nature of UCAP transformation, which is driven by their statefulness as we discussed in Section 1.4. This design is quite fragile and requires careful optimization of all layers in order to guarantee production efficiency. For example, the occurrence of the so-called *head-of-line blocking*, namely a slow request that blocks the entire queue, might be catastrophic in terms of response time for all inputs waiting to be processed [87]. Analyzing the residency time in the input

queue is important to understand the impact of old inputs on the response time for the new ones.

Response time

In the context of the UCAP-Python integration, the response time is the time it takes to generate the output for a given input. This comprises all the steps listed in Table 5. Response time is the main metric that we aimed to optimize in the present work. Moreover, response time influences queue residency in a cascading effect; thus, any improvement on this front has an important impact on the whole framework.

3.2.4 Discarded designs

During the process of optimizing the UCAP-Python integration, we considered several alternatives. Each design involved varying degrees of architectural changes and different technologies, but not all of them could be accepted as we are going to motivate in a few lines. In the coming paragraphs, we will briefly present the most relevant alternatives in our idea portfolio, discussing in particular the reasoning which made us discard them in the final design.

Disk-based message exchange

The first option that we have considered involved sending all "simple" types (i.e. scalar numeric values, strings, booleans) using the usual JSON format via some variant of the HTTP protocol (either HTTP/1.1 or HTTP/2), and storing array-like types in some local directory shared to both the UCAP node Java process and the Python transformation process. The format should have been shared by both parties involved. The prominent option in this case would have been NumPy's `npy` format [65], due to its extensive documentation and easy portability. A simple serializer/deserializer could have been written in Java to bridge the gap on that side, while standard NumPy could have done the work on the other. This strategy could have provided an opportunity to implement retry strategies in an easy fashion, without relying on internal implementation details from a third-party library.

Disk-based message exchange was ruled out due to several shortcomings. First of all, persisting arrays on disk is quite inefficient with respect to network calls, especially if the two processes live in the same machine or in the same subnet [33]. Second, among the ideas for the future evolution of UCAP, one of those in closest proximity is the removal of the restriction that requires the UCAP

master process and Python conversion processes to live in the same machine. In the current architecture, no shortstoppers are preventing the achievement of such a goal, but writing data to a local directory would have been a step in the wrong direction in this sense. One could write data in an NFS directory, but the performance penalties would have been quite important. To summarize, the problems outnumbered the advantages, which in turn felt easily achievable (without reinventing the wheel) with more standard approaches.

Broker-based message exchange

Another option, which we discussed already in Section 2.2, is to introduce a broker between the node and all its Python transformation processes. This could have simplified message delivery between the event producer and the consumer. Most importantly, it would have reduced coupling between the parties as we discussed previously, thus allowing an independent microservices-like lifecycle. The broker could accumulate messages that arrived while one of the parties is down, and replay them at the first suitable moment. A good candidate technology could have been Redis [24], since it allows keeping all data in memory, which could have been enough considering the ephemeral nature of UCAP inputs and outputs [87]. This alternative was abandoned due to the additional network hop required to communicate with the broker, which was not acceptable for low-latency use cases that UCAP aims to support. Also, this would have complicated the deployment structure of the project, due to the additional number of services to be started and monitored in production.

Pure Python UCAP distribution

Due to the popularity of UCAP among CERN control system operators, it was proposed several times in the past to develop a minimal copy in pure Python to support the baseline features needed to employ Python user-defined functions. Unofficial pure-Python portings were built for specific portions of UCAP (like event building) to enable in particular easier testing in isolation, but these efforts were never targeting the delivery of a production-ready product. A complete port would generate new problems which may be quite problematic to deal with, in particular from a maintainability point of view. The first point concerns the additional development effort required by the introduction of a brand-new product fully independent from the original one. Any important feature would require twice as much work to reach the production environment. This was already done in the past for several products at CERN, but this proved to be quite cumbersome and difficult to maintain in the long term [90]. In addition, UCAP is built upon

important internal Java libraries (JAPC and RDA3, see Section 1) for which valid alternatives do not exist in Python. Other teams at CERN are providing products that allow calling these libraries from a Python interpreter [43, 102], but they are considered not suitable to be used as the base for highly multithreaded applications like UCAP.

Tool-powered polyglotism

We considered a somewhat hybrid approach between the current infrastructure and that proposed in the previous paragraph. The idea is to run all Java libraries in a JVM, and call functions from within such JVM (for instance to poll the latest input) using some third-party tool like `Py4j` (see also Section 1.3). This is a similar approach to what Apache Spark does [4]. The most prominent problem in this case is that the Python process must have full ownership of the JVM it attaches to, therefore it cannot be shared among multiple Python transformation processes. This would have been quite an important shift from the previous architecture, which would have required massive redeployment campaigns and refactoring of all existing tooling. Moreover, the general feeling about this flavor of language interoperability is that any future version of either the JVM, the JDK, or the Python interpreter could break the whole system, thus leaving the software in an undesirable spot.

LMAX Disruptor

The LMAX disruptor is an open-source high-performance in-memory queue designed for low-latency applications. It leverages several strategies to overcome the prohibitive cost of locking in multithreaded environments, and to minimize general performance issues that are common to all queue implementations [47]. From a practical point of view, it provides an interface that is similar to that of a standard thread-safe queue for producer/consumer problems (see also Appendix C). However, it annihilates the inherent costs of using a queue and is usually the go-to solution for highly parallel short-lived workflows [148]. LMAX Disruptor is already employed at CERN, in an internal library for managing beamline settings [52]. We considered this option thoroughly, as a replacement for the standard Java `ArrayBlockingQueue` which we use to hold input messages before they are processed in transformations, and we decided not to adopt it. We observed that the majority of our latency problems are due to cross-language communication and user-defined function execution. Contention problems in the `ArrayBlockingQueue` are negligible with respect to the other components, thus making the adoption of the LMAX Disruptor almost irrelevant for UCAP in terms

of performance. As a side note, LMAX Disruptor is based on Java's `Unsafe` features [97]; while one should generally not be too concerned – most of the `Unsage` features employed in the Disruptor target more efficient array indexing and variable Compare-And-Swap (CAS) – it is preferable to avoid adding such dependencies to the classpath unless there are strong motivations to do so.

Distributed cache

The introduction of a *distributed caches* is a standard approach to increase the throughput for workloads that require CPU-bound or memory-bound computations triggered by specific inputs on a distributed herd of servers [87]. Indeed, as we discussed in Section 2.2.2, it is common practice in nowadays Software Engineering to deploy redundant instances of an application; this enables fault tolerance in case of unforeseen failures, and load balancing in situations of high inflow. The main downside of such an approach is that any local cache (e.g. an internal `HashMap` for a specific REST controller) is simply not enough to cache the results of heavy workloads, as one would ideally share them among all instances. A distributed cache could be modeled quite generally as a key/value store reachable by multiple servers which are part of the same application, and which retains previously computed pairs to avoid unneeded computations. In the context of a distributed CRUD application, for instance, one could store the results of complex queries to offload the shared database [119]. The market offers several open-source alternatives, like Apache Ignite [165], Redis [24] and memcached [45]. Similar approaches are already used at CERN, due to the importance of low-latency applications for the accelerators control system [94]. However, we considered the following problems:

- User-defined functions are in general stateful, which means that future results depend both on the current input and on the past history. While we could restrict caching to stateless workflows only, there is no general way to determine whether a given user-defined function is stateless or not. We could ask users to specify this characteristic at deployment time, but this would be quite cumbersome to manage and would pose an operational issue if the provided information turns out to be incorrect.

- In order to enable distributed caching, the whole possible input domain should be easily comparable for equality, thus allowing locating quickly the result of previous equivalent computations. While this is extremely efficient for integers and strings, thanks for instance to hash-based algorithms [87], it becomes quite complex when floating-point numbers are involved [58]. Comparing floating-point numbers bit-by-bit is inefficient – especially when

> the input is composed of huge arrays of doubles – and it is most likely not what one would like to do. Floating-point numbers that differ only by a small multiple of the *machine precision* should probably be accounted as equal for caching; otherwise, we would end up with an architecture that generates a huge amount of *cache misses*, thus invalidating any performance gain.

Therefore, we decided not to introduce a distributed cache in UCAP as part of this work. Our workflows present characteristics that make them unsuitable for caching, and the approach has inherent performance and development costs by itself. These additional overheads are not justified by the possible performance gains we expect to observe.

RDMA-based communication

We have presented briefly the key concepts of RDMA in Section 2.2. Indeed, we took the technology under consideration for the new design of the UCAP-Python integration. As a reminder, RDMA allows establishing a persistent channel between two parties, and leveraging it for buffer-less communication avoiding redundant copies [127]. This typically improves end-to-end latency, as well as general CPU utilization due to the reduced number of operations. However, we decided not to proceed further, even though it looked promising from a theoretical point of view. RDMA does not have the flexibility that we needed, in particular since UCAP poses no strict boundary on the size of exchanged messages, and this could deteriorate the throughput in the long term [49]. Moreover, the current third-party tooling for Java, and most importantly for multi-language deployments, is still missing a candidate technology with clear strategies for long-term support [104]. These considerations made us decide against leveraging RDMA for the UCAP-Python integration.

3.3 Results

The computational optimizations, whose theoretical aspects we discussed in the previous paragraphs, have been implemented in UCAP and are an integral part of the present work. We briefly outlined the work organization in Section 3.1. In Section 3.2, for each foundational improvement we discussed, we have also presented a synthetic benchmark to motivate the change. However, these benchmarks are somewhat biased by design, since they do not consider the full environment in which UCAP is submerged. This is particularly important for data-intensive software, as it is easy to misinterpret the results of microbenchmarks, and observe significantly different results in production [87]. In the present section, we will

Table 12: Release of the Python software dependencies in the experimental environment which we set up for Section 3.3. The last two columns tell whether the software is used either in the legacy release of UCAP, or for the new implementation bundling our improvements, or in both.

Software	Version	Released	Legacy	Latest
Python	`3.9.7`	Aug. 2021	✓	✓
grpc-io	`1.59.3`	Nov. 2023	×	✓
protobuf	`4.25.1`	Nov. 2023	×	✓
Flask	`1.1.2`	Apr. 2020	✓	×
Werkzeug	`2.0.3`	Feb. 2022	✓	×
requests	`2.25.0`	Nov. 2020	✓	×

present and analyze the values of several metrics – which we discussed already in Section 3.2.3 – that testify to the quality of the improvements introduced in the present work. This will serve both to certify the quality of the theoretical ideas we presented, and most importantly as a future reference for UCAP users about the expected performance of the Python integration.

Experimental setup

The base UCAP version that we used for the experiments presented in this section is `3.4.9`, released on December 8, 2023. The Python software it relies on is listed in Table 12; the Java software is listed in Table 13. Several third-party libraries may be quite outdated (e.g. Spring Boot and Flask) for stability and compatibility reasons. However, we kept the baseline group of common dependencies on the same version for both the legacy and the new implementation to avoid introducing differences in the environment. The garbage collector that we configured for the JVMs is `G1` [110], which is generally used for all Java software at CERN. Each JVM running a UCAP node was assigned a maximum heap space of 16 Gigabytes, in order to guarantee that no crashes would occur during the experimental phase due to out-of-memory problems. We had to choose a threshold so high because we wanted to test the response of the framework in situations of high pressure with large input values, and the serialization inefficiency of the legacy UCAP-Python implementation caused problems multiple times. The new implementation does not require this amount of memory; however, we wanted to test the two legacy and the new version of UCAP in an equivalent environment.

Table 13: Release of the main Java software dependencies in the experimental environment.

Software	Version	Released	Legacy	Latest
OpenJDK (64-bit)	`11.0.21`	Oct. 2023	✓	✓
Spring Boot family	`2.7.3`	Aug. 2022	✓	✓
jackson-core	`2.14.1`	Nov. 2022	✓	✓
micrometer-core	`1.11.5`	Oct. 2023	✓	✓
japc	`7.15.2`	Dec. 2023	✓	✓
grpc-core	`1.59.1`	Nov. 2023	×	✓
grpc-netty-shaded	`1.59.1`	Nov. 2023	×	✓
protobuf-java	`3.25.1`	Nov. 2023	×	✓
apache-httpclient	`4.5.13`	Oct. 2020	✓	×

Heap memory usage

As we anticipated in Section 3.2.3, we expect to observe important improvements in heap memory usage after the introduction of the enhancements that we discussed in the present work. A comparison between the old and new versions of the UCAP-Python integration is shown in Figure 22. The growth of the memory usage in the new version is much more stable and predictable, which is desirable for productional software [87]. Moreover, after each garbage collection – which is identifiable in the time series as a sudden descent [44] – the JVM can reclaim much more memory. This means the baseline available heap space is more permissive, and JVM crashes due to out-of-memory errors are much less likely to occur. Indeed, during the experimental phase of the work we experienced such crashes multiple times in the old version, and never after introducing our improvements. It is interesting to observe that the peak memory usage is higher in the new UCAP version. Indeed, under the conditions that we set up for the experiment, the JVM hosting the new version occupies approximately 8 Gigabytes, while the legacy version requires no more than 6.5 Gigabytes. However, such information is strongly misleading: generational garbage collectors like G1 place short-lived objects in a fast-paced area that is flushed rather frequently [44]. Temporary objects created during serialization and deserialization are short-lived by definition, therefore they do not appear in our measurements, which are obtained via a sampling at fixed time intervals. This is confirmed by the complete disappearance of JVM crashes due to heap space exhaustion which we observed in the new version.

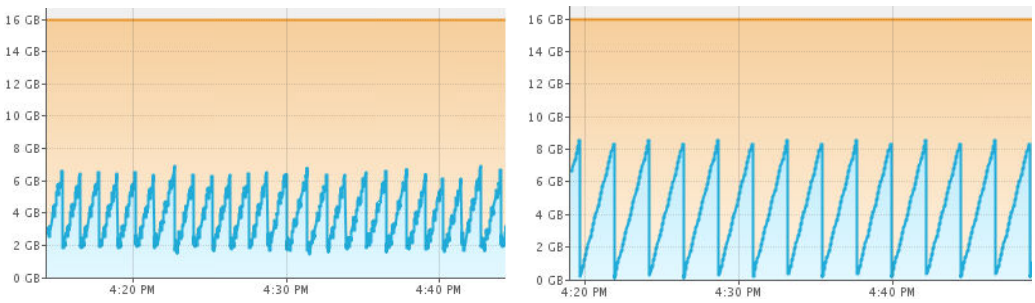

Figure 22: Comparison of heap usage as a time series of two JVMs, one hosting a legacy version of UCAP (left), and one running the improvements presented in this work (right). The orange area represents the available heap memory (16 Gigabytes for both), while the blue line represents memory consumption. The data has been collected using VisualVM, since it enables collecting snapshots more frequently and with less impact on the JVM with respect to other solutions like Prometheus.

Response time

As we highlighted in Section 3.2.3, response time is the most critical metric to optimize since it was the one that received the most compelling complaints in previous versions of UCAP. Moreover, its impact is even more important after the introduction of the input dispatching enhancements which we discussed in Section 3.2.2. We expected to observe important improvements concerning the response time of UCAP-Python transformations after evaluating the encouraging results of the synthetic experiment which we set up in Section 3.2.1. Indeed, we managed to register similar results in the production-like environment, as evidenced by the plots in Figure 23. The performance improvement for large arrays (i.e. more than 10^4 elements) is close to two orders of magnitude for 64-bit integer arrays. Moreover, the enhancements presented in this work open another possibility for performance optimization, since as we have already highlighted earlier using smaller data types results in a much faster response time. The results reported in the figure are the 99.9th percentiles, meaning that 99.99% of the request will typically be served faster than the value reported. This is generally believed to be one of the most suitable metrics to evaluate the performance of online data processing platforms [37,87]. Indeed, slow responses – even if they are a negligible number – present all sorts of problems in the context of a low-latency framework, which would even cause an important performance deterioration in the whole platform. A typical example is Head-of-Line blocking (HOL blocking), which is an undesirable situation when a single slow work item holds back an

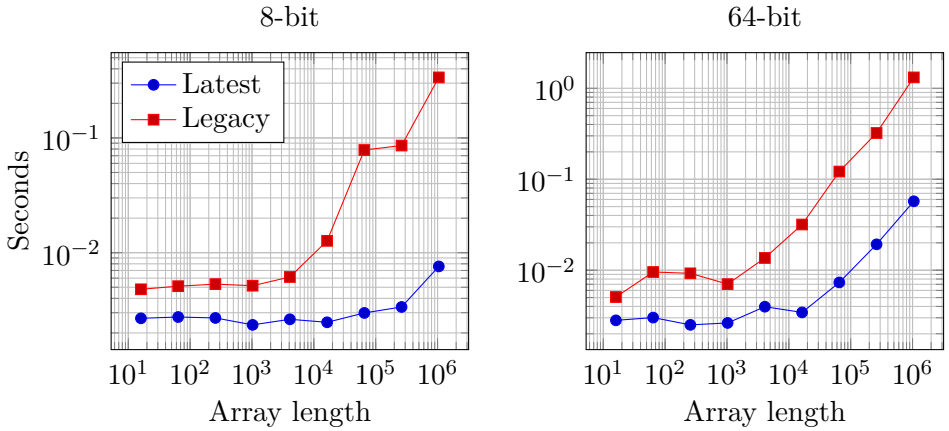

Figure 23: 99.9th percentiles of the response time (i.e. the time delta between the instant when the message is extracted from the input queue and the instant when the output becomes available to the client) of two UCAP nodes running a legacy version of the framework (red) and a new version with our enhancements (blue). The user-defined function sleeps for one millisecond and then returns the same value it received as the input. We considered inputs containing a single 1D array of 8-bit integers (left) and 64-bit integers (right). Both axes are on a logarithmic scale.

important number of inputs that cannot be processed until the previous operation is completed [87].

Queue residency

Our main goal with regard to queue residency was to reduce the influence it has on the response time. Letting input messages queue in a data structure waiting to be processed is dangerous as we argued earlier, since it may lead to JVM crashes due to memory exhaustion in situations of high load on the framework. Moreover, the heap space occupied by the objects referenced by the input queue is effectively wasted: input messages, which may be as large as several Megabytes, are discarded as soon as they are serialized and pushed to the communication channel, and they have no other purpose. Reducing and stabilizing queue residency would fix all these problems. The results of our observations, for inputs containing a 1D array of 64-bits integers, are summarized in Figure 24. The queue residency time is more predictable in the new implementation, and most importantly it has been uniformized by reducing almost all cases to the baseline of 10^{-6} seconds. The

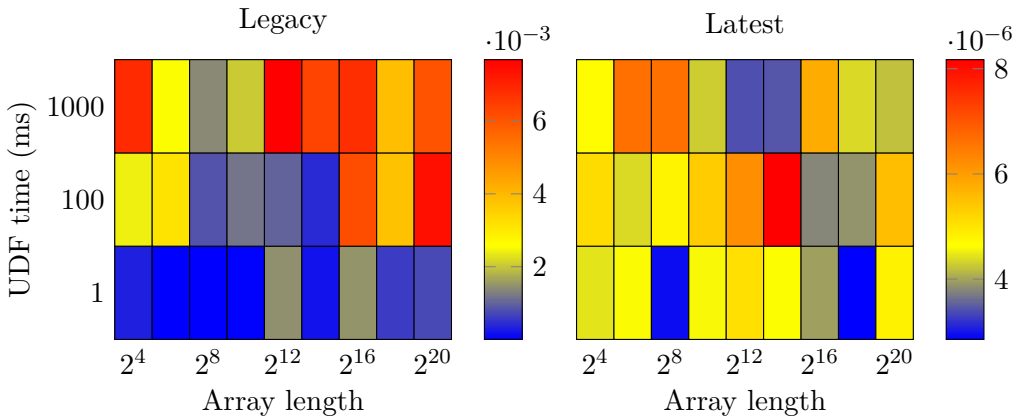

Figure 24: Color plots depicting the mean queue residency for the legacy version of UCAP (left) and after applying our improvements to the UCAP-Python integration (right). We considered user-defined functions having several different execution times (Y axis) to introduce more variability on the response time. The input arrival interval is 1.2 seconds.

maximum queuing time in the legacy version is 10^{-3} seconds, therefore in some cases we observe an improvement of three orders of magnitude. The number of experiments that we could have performed on this topic is extremely wide, as the degrees of freedom at play are quite numerous. We only varied the input size (i.e. the array length) and the execution time of the user-defined function; however, another possible experiment may consist of varying the arrival rate of input messages, or the speed of the communication channel, and observing the reaction of the framework through the metrics that we exposed.

4 Conclusion

This thesis presents an important improvement for critical software in the CERN accelerator context. We leveraged the theoretical aspects of inter-process communication and parallel computing, which we have introduced in Section 2, to achieve a measured speed-up of two orders of magnitude in the response time of UCAP-Python transformations. The main ideas behind such an important improvement were extensively presented in Section 3.2.1 and Section 3.2.2:

- A more refined serialization pipeline based on Protobuf [59] instead of JSON, with a dedicated approach to handle numeric arrays due to their importance in the CERN control system [6];
- A pipelined execution model that aims at hiding the communication time behind the user-defined function execution time, inspired by recent advancements in high-performance deep learning [14].

A speed-up of this scale in an established software like UCAP is quite uncommon and has been totally disruptive with regard to the applicability of the UCAP framework within the CERN experimental community. Most importantly, the changes presented in this work are fully backward compatible, since they only target the internals of UCAP infrastructure.

Facilitated future evolutions

Several planned improvements to the UCAP framework will benefit from the introduction of our changes. These projects have been planned for several years already; however, they have been pushed back in the queue for practical limitations, due to the doubts which have been raised about their feasibility.

The first improvement consists of decoupling the UCAP main JVM process and the Python processes responsible for executing the user-defined functions. The high-level idea is to let them live in different machines, possibly even in different data centers. While this is going to increase communication latency [33], decoupling the processes increases robustness and simplifies the deployment strategy [87]. It will also simplify the transition towards Kubernetes, which nowadays is considered the standard deployment platform in the industry, being also increasingly adopted at CERN for both new and legacy projects [31, 159].

Another improvement facilitated by the present work is the introduction of a Dead-Letter Queue (DLQ) with replaying capabilities in UCAP [87]. Such a facility collects messages that could not be processed either due to a network failure, or an occasional user-code failure. Users then will be able to consult this

queue and replay messages in batch occasionally depending on their needs. In the previous version of the UCAP-Python implementation, appending a batch of old unprocessed inputs to those that are continuously produced by the always-on instrumentation in the CERN control system could result in a severe delay in the delivery of outputs. Our performance improvements made Python transformations match more closely the throughput of Java-based transformations, thus removing any language-specific limitation in the addition of a DLQ.

Finally, a hypothetical future evolution of the UCAP framework consists of splitting the event-building and the data processing parts of UCAP, and placing a broker (i.e. Apache Kafka [88]) in the middle to streamline the communication. This would provide several benefits:

- Event-building could run in a managed environment like Kubernetes, thus enabling lightweight deployment of redundant instances to shield against message loss, and reducing the computational load on the processing nodes;
- Events to be processed could be stored in a dedicated facility (i.e. the broker) instead of an in-memory queue, thus reducing the risk of out-of-memory crashes and providing additional robustness guarantees;
- It would be possible to specify an eviction policy for the broker, which would provide a (limited) history of all processed messages. This would even open the possibility of replaying a rule-based subset of messages.

The main limitation of this approach is the increased latency coming from the additional hops. However, our improved serialization approach is an important step towards the feasibility of this major renovation work. As we highlighted in Section 3.2.1, the usability of the method proposed in the present work is not restricted to the communication between Java and Python, and can be easily extended to other languages, including Java-Java communication.

Expanded utilization scenarios

Improving response time for large arrays allows for a plethora of new use cases to be treated by UCAP. One of the many applications of UCAP revolve around the idea of *Beam Loss Monitoring* (BLM), whose goal is to optimize the "quality" of accelerator beams to maximize the number of collisions in the long run [5, 21]. Extensive monitoring and analysis are required to this end, and UCAP has been employed in this sense to provide an execution framework for scientific Python analysis workflows (e.g. computation of FFTs). The improvements presented as part of this work have already provided significant benefits to these applications,

enabling analysis on large-scale datasets that were untreatable in the old versions due to performance limitations. We also expect Python frameworks based on UCAP in general (e.g. *GeOFF*, see Section 1.4) to benefit significantly from our improvements, due to their intensive usage of numeric data as a central part of their workflow.

Future work

An important work direction for the coming months is optimizing the Python-based side of Protobuf serialization. In the present state, serializing sequences of bytes is inefficient, since it requires an unnecessary copy as we mentioned in Section 3.3. This is a limitation of the Python implementation of the framework, which does not accept buffer-like objects in assignments to `bytes` proto fields, as opposed to the Java implementation which provides this functionality. As an integral part of the present work, we opened a pull request in the official Protocol Buffer repository which implements the feature (Figure 25). The acceptance process takes time and requires review and approval from multiple engineers, thus it may take some time to see that addition in a stable Protobuf release. We plan to follow up on this front as needed in the coming weeks, improving our proposal depending on the requirements of the core development team.

Another important work item for the UCAP-Python integration evolution in the coming months is the introduction of an internal backpressure mechanism. Ideally, we would send as many inputs as we can on the channel, since gRPC and HTTP/2 are able to handle the load thanks to multiplexing [32,61]; however, swarming the Python process with inputs should be avoided at all costs. Each message requires some work for serialization/deserialization. While this is not guarded by the GIL [61,155], Python processes should be lightweight and almost *functional*, in the sense that it is not ideal to have too much of a hidden state being processed in background threads. This design principle was enforced in particular to improve the ease of debugging. The backpressure facility should provide Python-side metrics to inform the main Java process about how many

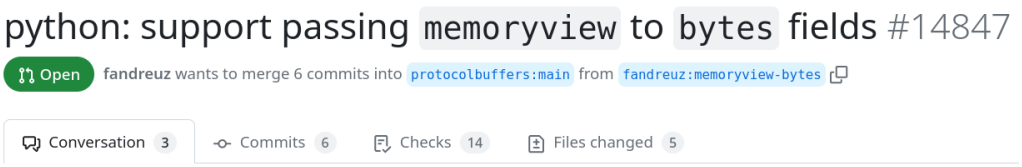

Figure 25: Open pull request in the Protobuf repository to enable direct assignment of `memoryview` objects to `bytes` proto fields.

messages were processed in a given amount of time. Such information should then be used to infer an optimal rate to adopt for sending new inputs on the channel [87].

Additional work will be required to upgrade the software for Python 3.12, which brings in several interesting changes. The introduction of direct bindings to the buffer protocol [167] and the first steps towards a more permissive GIL [134] are interesting new alternatives that we will explore to improve the response time of Python transformations. The release of NumPy 2.0 will be an important occurrence as well, which will require special care and testing [65].

References

[1] G. Aad, T. Abajyan, B. Abbott, J. Abdallah, S. A. Khalek, A. A. Abdelalim, R. Aben, B. Abi, M. Abolins, O. AbouZeid, et al. Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC. *Physics Letters B*, 716(1):1–29, 2012.

[2] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. 2015.

[3] Apache Software Foundation. Apache Lucene. `https://lucene.apache.org/`. Accessed: February 17, 2024.

[4] Apache Software Foundation. Apache Spark. `https://spark.apache.org/`. Accessed: February 17, 2024.

[5] P. Arpaia, G. Azzopardi, F. Blanc, X. Buffat, L. Coyle, E. Fol, F. Giordano, M. Giovannozzi, T. Pieloni, R. Prevete, et al. Beam Measurements and Machine Learning at the CERN Large Hadron Collider. *IEEE Instrumentation & Measurement Magazine*, 24(9):47–58, 2021.

[6] V. Baggiolini. JAPC - The Java API for Parameter Control. In *proc. ICALEPCS'05*, 2005.

[7] A. Banihirwe. Dask serialization. Docs, 2020.

[8] M. Barberan Marin, Y. Muttoni, J.-P. Corso, M. Bernardini, J. Coupard, S. Bartolome-Jimenez, S. Chemli, T. Birtwistle, K. Foraz, S. Grillot, et al. Integration, Configuration and Coordination: from Project to Reality, at CERN. Technical report, 2016.

[9] M. Batty. Digital twins, 2018.

[10] O. Ben-Kiki, C. Evans, and I. döt Net. YAML spec v1.2.2. Spec document, 2021.

[11] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2019.

[12] M. Benedikt, A. Chance, B. Dalena, D. Denisov, M. Giovannozzi, J. Gutleber, R. Losito, M. Mangano, T. Raubenheimer, W. Riegler, et al. Future Circular Hadron Collider FCC-hh: overview and status. *arXiv preprint arXiv:2203.07804*, 2022.

[13] T. J. Berners-Lee. The world-wide web. *Computer networks and ISDN systems*, 25(4-5):454–459, 1992.

[14] M. Besta and T. Hoefler. Parallel and distributed graph neural networks: An in-depth concurrency analysis. *arXiv preprint arXiv:2205.09702*, 2022.

[15] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site reliability engineering: How Google runs production systems.* " O'Reilly Media, Inc.", 2016.

[16] G. E. Blelloch. Introduction to data compression, 2001.

[17] J. Bloch. *Effective java.* Addison-Wesley Professional, 2017.

[18] L. Bromiley and R. Cunningham. Challenges of the FCC-ee civil engineering studies. *Journal of Instrumentation*, 19(02):T02005, 2024.

[19] L. Bulej, V. Horkỳ, M. Tucci, P. Tuma, F. Farquet, D. Leopoldseder, and A. Prokopec. GraalVM Compiler Benchmark Results Dataset (Data Artifact). In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, pages 65–69, 2023.

[20] P. Butcher. Seven Concurrency Models in Seven Weeks: When Threads Unravel. *Seven Concurrency Models in Seven Weeks*, pages 1–296, 2014.

[21] E. Calvo Giraldo, J. Martínez Samblas, C. Zamantzas, J. Kral, E. Effinger, M. Gonzalez Berges, S. Morales Vigo, and B. Salvachúa. JACOW: The Diamond Beam Loss Monitoring System at CERN LHC and SPS. *JACoW IBIC*, 2022:202–206, 2022.

[22] R. Cañamares, P. Castells, and A. Moffat. Offline evaluation options for recommender systems. *Information Retrieval Journal*, 23(4):387–410, 2020.

[23] B. Cannon. Storing project metadata in pyproject.toml. PEP 621, 2020.

[24] J. Carlson. *Redis in action.* Simon and Schuster, 2013.

[25] CERN. home.cern.ch, 2024.

[26] R. Chandra. *Parallel programming in OpenMP.* Morgan kaufmann, 2001.

[27] P. P. Chaudhuri. *Computer organization and design*. PHI Learning Pvt. Ltd., 2008.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[29] L. Cseppentő and M. Büttner. UCAP: A Framework for Accelerator Controls Data Processing at CERN. *JACoW*, pages 230–235, 2022.

[30] J. de Fine Licht, A. Kuster, T. De Matteis, T. Ben-Nun, D. Hofer, and T. Hoefler. StencilFlow: Mapping large stencil programs to distributed spatial computing systems. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 315–326. IEEE, 2021.

[31] J. B. de Martel, R. Gorbonosov, and N. Madysa. Machine Learning Platform: Deploying and Managing Models in the CERN Control System. *JACoW*, pages 50–55, 2022.

[32] H. de Saxcé, I. Oprescu, and Y. Chen. Is HTTP/2 really faster than HTTP/1.1? In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 293–299. IEEE, 2015.

[33] J. Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 1, 2009.

[34] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[35] S. Deghaye and E. Fortescue-Beck. Introduction to the BE-CO Control System. Technical report, 2020.

[36] P. Di Francesco, P. Lago, and I. Malavolta. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150:77–97, 2019.

[37] J. Ding, R. Cao, I. Saravanan, N. Morris, and C. Stewart. Characterizing service level objectives for cloud services: Realities and myths. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 200–206. IEEE, 2019.

[38] A. B. Downey. *The little book of semaphores*. 2005.

[39] A. Dworak, F. Ehm, P. Charrue, and W. Sliwinski. The new CERN controls middleware. In *Journal of Physics: Conference Series*, volume 396, page 012017. IOP Publishing, 2012.

[40] C. Ebert and C. H. C. Duarte. Digital transformation. *IEEE Softw.*, 35(4):16–21, 2018.

[41] R. Eckstein. *Java Enterprise best practices*. " O'Reilly Media, Inc.", 2002.

[42] F. Ehm and A. Dworak. A remote tracing facility for distributed systems. In *Conf. Proc.*, volume 111010, page WEMAU001, 2011.

[43] P. Elson, C. Baldi, and I. Sinkarenko. Introducing Python as a Supported Language for Accelerator Controls at CERN. *JACoW*, pages 236–241, 2022.

[44] B. J. Evans, J. Gough, and C. Newland. *Optimizing Java: practical techniques for improving JVM application performance*. " O'Reilly Media, Inc.", 2018.

[45] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[46] T. A. S. Foundation. Apache Avro specification v1.11.1. Documentation, 2022.

[47] M. Fowler. The LMAX Architecture. 2011.

[48] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. Report from workshop and panel on the status of serverless computing and function-as-a-service (FaaS) in industry and research. In *First International Workshop on Serverless Computing (WoSC)*, 2017.

[49] P. W. Frey and G. Alonso. Minimizing the hidden cost of RDMA. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560. IEEE, 2009.

[50] S. Fuess, O. Gutsche, M. Kirby, R. Kutschke, A. Lyon, A. Norman, G. Perdue, E. Sexton-Kennedy, et al. Fermilab computing at the Intensity Frontier. In *Journal of Physics: Conference Series*, volume 664, page 032012. IOP Publishing, 2015.

[51] J. Fullerton, L. Jensen, and J. Spanggaard. Software renovation of the CERN experimental areas. In *Conf. Proc.*, volume 111010, page MOPMS034, 2011.

[52] M. Gabriel and R. Gorbonosov. Disruptor-Using High Performance, Low Latency Technology in the CERN Control System. 2015.

[53] M. Gaillard. CERN Data Centre passes the 200-petabyte milestone. 2017.

[54] S. García, S. Ramírez-Gallego, J. Luengo, J. M. Benítez, and F. Herrera. Big data preprocessing: methods and prospects. *Big Data Analytics*, 1(1):1–22, 2016.

[55] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. *Proceedings of the VLDB Endowment*, 2(2):1414–1425, 2009.

[56] B. Goetz. *Java concurrency in practice*. Pearson Education, 2006.

[57] B. Goetz. Data Oriented Programming in Java, 2022.

[58] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.

[59] Google Inc. Protocol Buffers. `https://developers.google.com/protocol-buffers/`. Accessed: February 17, 2024.

[60] C. Green, J. Kowalkowski, M. Paterno, M. Fischler, L. Garren, and Q. Lu. The art framework. In *Journal of Physics: Conference Series*, volume 396, page 022020. IOP Publishing, 2012.

[61] grpc. gRPC: A high performance open-source universal RPC framework., 2020.

[62] G. Hager and G. Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[63] K. Hanke, A. Blas, J.-M. Lacroix, D. Aguglia, P. Dahlen, D. Nisbet, J.-L. Grenard, S. Chemli, R. Froeschl, T. Birtwistle, et al. Status and Plans for the Upgrade of the CERN PS Booster. Technical report, 2015.

[64] B. P. Harenslak and J. de Ruiter. *Data Pipelines with Apache Airflow*. Simon and Schuster, 2021.

[65] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[66] A. Hermann, J. Krige, U. Mersits, and D. Pestre. *History of CERN*, volume 3. North-Holland Amsterdam, 1987.

[67] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[68] T. Hukkinen. tomllib — Parse TOML files. Python standard documentation, 2022.

[69] T. Hukkinen and S. Jain. tomllib: Support for Parsing TOML in the Standard Library. PEP 680, 2022.

[70] A. Huschauer, M. Coly, D. Cotte, H. Damerau, M. Delrieux, Y. Dutheil, S. Easton, J. Dumont, M. Fraser, A. Guerrero, et al. Beam performance and operational efficiency at the CERN Proton Synnchrotron. In *14th Int. Particle Accelerator Conf.(IPAC'23), Venice, Italy*, 2023.

[71] G. Iadarola, T. Levens, J. Wozniak, R. De Maria, V. Baggiolini, and J. G. Cobas. A users view on exploiting the control system in MDs. 2019.

[72] V. inc. Micrometer documentation. Technical report, 2017.

[73] T. Index. TIOBE Index—May 2023. *Accessed: Jun*, 2, 2023.

[74] R. Inglés, P. Perek, M. Orlikowski, and A. Napieralski. A simple multithreaded C++ framework for high-performance data acquisition systems. In *2015 22nd International Conference Mixed Design of Integrated Circuits & Systems (MIXDES)*, pages 153–157. IEEE, 2015.

[75] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3:711–732, 2021.

[76] D. Izzo and F. Biscani. darioizzo/audi: Pybind11. *Zenodo*.

[77] D. Jacquet, R. Gorbonosov, G. Kruk, and P. P. Mira. LSA-the high level application software of the LHC and its performance during the first 3 years of operation. these proceedings, 2014.

[78] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards transparent hot-swapping support for producer-consumer components. In *Proceedings of Second International Workshop on Unanticipated Software Evolution (USE 2003)*, 2003.

[79] D. Jemerov and S. Isakova. *Kotlin in action*. Simon and Schuster, 2017.

[80] R. Johnson and J. Vlissides. Design patterns. *Elements of Reusable Object-Oriented Software Addison-Wesley, Reading*, 1995.

[81] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Scientific workflow applications on Amazon EC2. In *2009 5th IEEE international conference on e-science workshops*, pages 59–66. IEEE, 2009.

[82] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 158–169, 2015.

[83] B. W. Kernighan and D. M. Ritchie. The C programming language. 2002.

[84] M. Kerrisk. *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[85] W. Kim. Cloud computing: Today and tomorrow. *J. Object Technol.*, 8(1):65–72, 2009.

[86] M. S. Kirkpatrick. OpenCSF: An Online Interactive Textbook for Computer Systems Fundamentals. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 1105–1105, 2018.

[87] M. Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017.

[88] J. Kreps. Kafka : a Distributed Messaging System for Log Processing. 2011.

[89] J. Kurose and K. Ross. Computer networks: A top down approach featuring the internet, 2010.

[90] J. Lauener, W. Sliwinski, and G. CERN. How to design & implement a modern communication middleware based on ZeroMQ. In *Proc of ICALEPCS*, volume 17, pages 45–51, 2017.

[91] E. Lopienska. The CERN accelerator complex, layout in 2022. Technical report, 2022.

[92] G. MacLachlan, J. Hurlburt, M. Suarez, K. L. Wong, W. Burke, T. Lewis, A. Gallo, J. Flidr, R. Gabiam, J. Nicholas, et al. Building a shared resource HPC center across university schools and institutes: a case study. *arXiv preprint arXiv:2003.13629*, 2020.

[93] C. Majors, L. Fong-Jones, and G. Miranda. *Observability Engineering*. " O'Reilly Media, Inc.", 2022.

[94] T. Marques Oliveira, A. Papageorgiou Koufidis, M. Bräger, S. Halastra, and D. Martin Anido. Distributed Caching at Cloud Scale with Apache Ignite for the C2MON Framework. *JACoW*, pages 307–310, 2022.

[95] R. C. Martin. *The clean coder: a code of conduct for professional programmers*. Pearson Education, 2011.

[96] R. C. Martin. Clean architecture, 2017.

[97] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. *ACM Sigplan Notices*, 50(10):695–710, 2015.

[98] S. Mathew and J. Varia. Overview of amazon web services. *Amazon Whitepapers*, 105:1–22, 2014.

[99] S. N. Mehmood, N. Haron, V. Akhtar, and Y. Javed. Implementation and experimentation of producer-consumer synchronization problem. *International Journal of Computer Applications*, 975(8887):32–37, 2011.

[100] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[101] D. F. Molina Bocanegra. Bindings Performance Analysis across Programming Languagesin a Messaging Platform, 2013.

[102] K. E. Nelson and M. K. Scherer. JPype. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2020.

[103] S. Newman. *Building microservices*. " O'Reilly Media, Inc.", 2021.

[104] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting managed heaps in distributed big data systems. *ACM SIGPLAN Notices*, 53(2):56–69, 2018.

[105] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta. Comparison of JSON and XML data interchange formats: a case study. *Caine*, 9:157–162, 2009.

[106] S. Oaks. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code*. " O'Reilly Media, Inc.", 2014.

[107] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. A study of causes and consequences of client-side JavaScript bugs. *IEEE Transactions on Software Engineering*, 43(2):128–144, 2016.

[108] D. E. O'Leary. Artificial intelligence and big data. *IEEE intelligent systems*, 28(2):96–99, 2013.

[109] K. A. Olive. Supersymmetric dark matter after run I at the LHC: from a TeV to a PeV. *arXiv preprint arXiv:1510.06412*, 2015.

[110] Oracle inc. Java SE 11 Documentation, 2018.

[111] Oracle inc. Java SE 19 Documentation, 2022.

[112] S. K. Palaniappan and P. B. Nagaraja. Efficient data transfer through zero copy. *IBM developerworks*, page 184, 2008.

[113] K. Parasyris, K. Keller, L. Bautista-Gomez, and O. Unsal. Checkpoint restart support for heterogeneous HPC applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 242–251. IEEE, 2020.

[114] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[115] S. Pedroni and N. Rappin. *Jython Essentials: Rapid Scripting in Java*. " O'Reilly Media, Inc.", 2002.

[116] J. M. Perkel et al. Julia: come for the syntax, stay for the speed. *Nature*, 572(7767):141–142, 2019.

[117] M. Peryt, D. Noll, R. Scrivens, and M. Hrabia. The Linac4 Source Autopilot. *JACoW*, pages 665–670, 2022.

[118] A. J. Peters and L. Janyst. Exabyte scale storage at CERN. In *Journal of Physics: Conference Series*, volume 331, page 052015. IOP Publishing, 2011.

[119] A. Petrov. *Database Internals: A deep dive into how distributed data systems work*. O'Reilly Media, 2019.

[120] A. Pitrou. Experimental zero-copy pickling. Pull request, 2018.

[121] A. Pitrou. Pickle protocol 5 with out-of-band data. PEP 574, 2018.

[122] L. B. A. Rabai, B. Cohen, and A. Mili. Programming language use in us academia and industry. *Informatics in Education*, 14(2):143, 2015.

[123] B. Rae, M. Hrabia, V. Baggiolini, D. Banerjee, J. Bernhard, M. Brugger, N. Charitonidis, L. Gatignon, A. Gerbershagen, R. Gorbonosov, et al. Controlling the CERN Experimental Area Beams. *arXiv preprint arXiv:2202.01705*, 2022.

[124] J. Reis and M. Housley. *Fundamentals of Data Engineering: Plan and Build Robust Data Systems*. O'Reilly Media, Incorporated, 2022.

[125] D. Rensin. *Kubernetes*. O'Reilly Media, Incorporated, 2015.

[126] J. W. Rittinghouse, J. F. Ransome, et al. Cloud Computing: Implementation, Management and Security. *Cloud Computing Implementation Management and Security*, 1999.

[127] A. Romanow and S. Bailey. An Overview of RDMA over IP. In *Proceedings of the First International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2003)*, 2003.

[128] L. Safran. gRPC Grafana Benchmarks.

[129] J. E. Sammet. Programming languages: History and future. *Communications of the ACM*, 15(7):601–610, 1972.

[130] W. N. Scherer III, D. Lea, and M. L. Scott. Scalable synchronous queues. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 147–156, 2006.

[131] M. Shahrad, J. Balkind, and D. Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 1063–1075, 2019.

[132] D. Singh and C. K. Reddy. A survey on platforms for big data analytics. *Journal of big data*, 2(1):1–20, 2015.

[133] M. Šipek, D. Muharemagić, B. Mihaljević, and A. Radovan. Enhancing performance of cloud-based software applications with GraalVM and Quarkus. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1746–1751. IEEE, 2020.

[134] E. Snow. A per-interpreter GIL. PEP 684, 2022.

[135] Y. Solihin. *Fundamentals of parallel multicore architecture*. CRC Press, 2015.

[136] J. Song and J. Alves-Foss. Performance review of zero copy techniques. *International Journal of Computer Science and Security (IJCSS)*, 6(4):256, 2012.

[137] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[138] K. Srinath. Python: the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357, 2017.

[139] A. Sriraman and A. Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 733–750, 2020.

[140] B. Stroustrup, A. Koenig, and B. E. Moo. C++. *Encyclopedia of Software Engineering*, 2002.

[141] A. Sumaray and S. K. Makki. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th international conference on ubiquitous information management and communication*, pages 1–6, 2012.

[142] R. Sumbaly, J. Kreps, and S. Shah. The big data ecosystem at linkedin. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1125–1134, 2013.

[143] V. Sundarapandian. *Probability, statistics and queuing theory*. PHI Learning Pvt. Ltd., 2009.

[144] C. Swarna and Z. Ansari. Apache pig: a data flow framework based on Hadoop map-reduce. *International Journal of Engineering Trends and Technology (IJETT)*, 50(5):271–275, 2017.

[145] A. Tanenbaum, D. Wetherall, J. Kurose, and K. Ross. Computer networks title: Computer networking: A top-down approach. *Instructor*, 201901, 2019.

[146] K. Taranov, R. Bruno, G. Alonso, and T. Hoefler. Naos: Serialization-free RDMA networking in Java. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 1–14, 2021.

[147] A. L. Tavares and M. T. Valente. A gentle introduction to OSGi. *ACM SIGSOFT Software Engineering Notes*, 33(5):1–5, 2008.

[148] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. *Technical paper. LMAX*, 206, 2011.

[149] S. Tonse. Scalable microservices at Netflix. challenges and tools of the trade, 2018.

[150] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[151] N. Trofimov, V. Baggiolini, S. Jensen, K. Kostro, F. Di Maio, and A. Risso. Remote Device Access in the new CERN Accelerator Controls middleware. In *ICALEPCS*, volume 1, page 496, 2001.

[152] C. Tudose, C. Odubăşteanu, and S. Radu. Java reflection performance analysis using different Java development. *Advances in Intelligent Control Systems and Computer Science*, pages 439–452, 2013.

[153] J. Turnbull. *Monitoring with Prometheus*. Turnbull Press, 2018.

[154] C. Uden. Exploring implementations for online control & optimisation applications in the operation of a particle accelerator. Technical report, 2023.

[155] G. Van Rossum and F. L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[156] P. Vassiliadis, A. Simitsis, and E. Baikousi. A taxonomy of ETL activities. In *Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP*, pages 25–32, 2009.

[157] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is All You Need. In *Advances in Neural Information Processing Systems*, pages 5998–6008. Curran Associates, Inc., 2017.

[158] V. Vernon. *Implementing domain-driven design*. Addison-Wesley, 2013.

[159] R. Voirin, M. Vanden Eynden, and T. Oulevey. The State of Containerization in CERN Accelerator Controls. *JACoW*, pages 829–834, 2022.

[160] M. von Lowis. Flexible string representation. PEP 393, 2010.

[161] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda. GPU-aware MPI on RDMA-enabled clusters: Design, implementation and evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2595–2605, 2013.

[162] J. Wozniak, V. Baggiolini, D. G. Quintas, and J. Wenninger. Software interlocks system. *Proc. of ICALEPCS07, Knoxville, Tn*, 2007.

[163] J. Wozniak, C. Roderick, and S. R. WEPHA163. NXCALS-Architecture and Challenges of the Next CERN Accelerator Logging Service. *Proc. ICALEPCS'19*, (17):1465–1469, 2019.

[164] J. Xu, J. Yin, H. Zhu, and L. Xiao. Modeling and verifying producer-consumer communication in Kafka using CSP. In *7th Conference on the Engineering of Computer Based Systems*, pages 1–10, 2021.

[165] M. Zheludkov, T. Isachenko, et al. *High Performance in-memory computing with Apache Ignite*. Lulu. com, 2017.

[166] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

[167] J. Zijlstra. Making the buffer protocol accessible in Python. PEP 688, 2022.

A Observability

As part of the practical improvements implemented as part of this work, we decided to enhance the monitoring infrastructure in UCAP. In addition to the direct benefits of such modifications, we decided to leverage monitoring as a tool to assess the achievement of first-class goals of this work, which we mentioned in Section 3.3. In order to enrich the reader's context on the topic, in the following paragraphs we are going to present several ideas in the field of *observability*, which has lately become an extremely wide topic, especially factoring in the recent trends in the DevOps environment. Then, we will briefly depict the current state of observability in UCAP, which has worked well since its first release, albeit with some evident shortcomings; finally we will propose an alternative monitoring implementation, that is based on standard tools in the industry, which we have fine-tuned for our peculiar architecture.

Monitoring overview

Software *monitoring* – often used interchangeably with *observability* – consists in extracting information about the runtime state of an application [93]. Information is typically materialized as a set of named values called *metrics*, each generated independently by a specific software component to report its current status. For instance, we may want to count the number of incoming requests per minute in an HTTP server, or the average response time of a gRPC endpoint. A trivial way to expose metrics is to print the value of some interesting variables to the standard output. Of course, this approach does not scale well and provides little added value, and the recommended approach is to rely on dedicated third-party tools. Indeed, metrics are extremely valuable when used in conjunction with inspection and visualization tools, and could serve an extremely diverse set of purposes in the lifecycle of a software product:

- Alerting: metrics are frequently used to trigger notifications for engineers responsible for fixing urgent problems in a critical application. For instance, such a situation may happen if the software is malfunctioning (e.g. spike in the count of *5xx* errors) or if it does not respond at all.
- Testing: metrics may be used to implement black-box tests, by computing the difference between an expected reference and a measured value. The implementation of a test of this kind would require important additions to expose the relevant internal value, like a dedicated API endpoint. Such an endpoint would only be needed in the testing environment, which is not a desirable way to proceed [96]. By contrast, high-level tests on metrics

may be implemented as simple alerts (e.g. via Prometheus) in a TEST environment. For example, we established an alarm to verify periodically that the difference between the number of incoming events in UCAP and the number of output events is eventually zero.

- Software utilization: appropriate metrics may be put in place to assess the utilization of certain parts of the software that are candidates for deprecation or removal in future releases [114]. This could help in gaining insights into the impact those changes would have on the users.
- Dashboards are detailed views presenting information about running software. They are extremely useful for gaining insights on the application without digging into logs and metrics, in particular thanks to the possibility of associating interesting changes in the behavior of the software with specific points in time [15]. As an example, Grafana is a very popular tool to create dashboards powered by metrics.

The current popularity of microservices-based applications increased exponentially the number of independent software to be maintained, resulting in a sudden transition from the single monolithic backend to a plethora of lightweight processes [103]. The important advantages of modern software architecture are evident; however, the change in paradigm introduced a plethora of new challenges in the field of Software Reliability [15]. Monitoring is one of the most important weapons in the toolchain of development and quality assurance teams to tackle problems before they become evident to downstream users.

Observability server

We do not have a general definition of *observability server*, since all vendors typically define their own set of specifics and goals for their platform. In the context of this work, we will provide a weak definition by describing the most important features of this category of software products.

An observability server is responsible for taking care of the full lifecycle of metrics, meaning that it should:

1. Discover "interesting" instances which may be publishing metrics;
2. Scrape metrics from each instance;
3. Store the scraped metrics for future inspection in a long-lasting database;
4. Provide a programmatic interface to access data and execute queries.

The first step is typically implemented by declaring a list of *targets* in the server configuration; however, different vendors may provide alternative strategies. For example, the Prometheus Pushgateway allows ephemeral processes to register metrics in a *push* fashion, as opposed to the standard *pull* approach [153]. In the standard setup, the server is responsible for scraping metrics from each instance. Typically, this step is carried out via some TCP-based exchange (e.g. HTTP request, gRPC, …).

The last two points of the list may be subject to important variance in terms of implementation among different products, since each server may define its own storage strategy and interface for inspection. For instance, Prometheus defines its own solution called *PromQL* (Prometheus Querying Language) [153] which may be used in its proprietary web interface, as well as in its web-based API.

The choice of the observability server in general is as important as the instrumentation library: it is the layer that enables extracting added value from code monitoring, thus powering the leap from a disaggregated stream of tagged numbers to actual insights on running applications.

Java Management Extensions

Java Management Extensions (JMX) enable managing and inspecting software running on a JVM [110]. The cornerstone of this technology is the JMX *MBean*, namely a Java object that provides a set of methods that shall be called via JMX to influence the runtime of the application. Typical methods exposed via JMX are:

- Getters for relevant fields that may be of interest to observe the runtime of the application, for instance cache statistics to inspect the current size or the hit/miss ratio;
- Setters of specific properties to influence the status of the application without restarting it;
- Generic methods to enable more complex operations to be carried out on-demand to influence the behavior of the software, like clearing the content of some data structure to free heap memory.

The market offers many alternatives to expose JMX beans, building on the approach provided by the standard library which may feel cumbersome for some use cases. For instance, the Spring framework provides an annotation-based integration that allows exposing specific functions as managed attributes or operations. We provide a real-world example from the UCAP source code in Listing 17.

```java
@Component
@ManagedResource("cern.ucap:name=SecurityProperties,type=Security")
@AllArgsConstructor
public class SecurityPropertiesJmx {
    private final SecurityProperties securityProperties;
    @ManagedAttribute
    public String getMethodProtectionRoles() {
        return String.join(",", securityProperties.getMethodProtectionRoles());
    }
    @ManagedAttribute
    public void setMethodProtectionRoles(String roles) {
        securityProperties.setMethodProtectionRoles(roles);
    }
}
```

Listing 17: JMX bean implemented via Spring annotations. The annotation `@ManagedAttribute` represents an attribute (in this case `methodProtectionRoles` which may be managed (i.e. read and write operations) via JMX.

JMX beans are typically exposed via tools that can attach to the running JVM and display all the available operations and attributes. For example, *VisualVM* is bundled in the JDK, and thus is available to all Java developers out of the box. While JMX is mostly designed for human-based monitoring of running software, it is also possible to use it for automated monitoring by collecting metrics exposed by JMX beans. This is the strategy first adopted by UCAP, which was based on the Prometheus JMX Collector. The metrics collected were exposed on a simple HTTP endpoint, which was leveraged to wire the software with a Prometheus server. This solution is quite easy to set up, requires minimal configuration, and has the important advantage of reducing the number of technologies at play for read-write instrumentation.

However, JMX has quite an important performance impact in the presence of an important number of JMX beans and metrics [41], which was definitely the case for several UCAP instances. Requests on the metrics endpoint trigger a JMX collection which may even take several hundred seconds to complete. This is unacceptable for an online data processing platform, which needs fine-grained metrics to provide an adequately granular view of the behavior of the software. This is the reason why we decided to explore alternatives to replace – at least partially – the JMX-based collection of metrics.

Micrometer

The basic role of an observability client library is to provide a set of objects – typically called *meters* – to represent the current value of important quantities at runtime. Those values shall be exported in a standard format to deliver the measurements to the relevant observability server. Many alternatives exist on the market, but the general recommendation is to favor observability libraries that are agnostic on the choice of the observability server, thus enabling an effortless switch to a different solution in a future moment. However, this might come at the price of giving up some desirable features that may not be generally supported.

Micrometer [72] is an open-source solution maintained by VMware Inc., an important player in the field of cloud computing and virtualization. It defines an extensive set of meters that enable idiomatic and natural code instrumentation. A few examples:

- *Counter* represents a quantity that may only increase by positive steps. Counters are typically used to infer growing rates, for example to check the increase in the number of requests at a specific time in the day.
- *Gauge* represents a quantity that may be set to any value at any time. They are frequently used to deliver information about the internal state of the software at a specific point in time.
- *Timer*, as the name suggests, is a meter that can be used to time the execution duration of specific parts of the code, like short-lived background tasks or user-defined functions.

Meters are identified by a name, which should convey the kind of information they refer to (e.g. `DeviceServer.ValuesPublished`), and a set of *labels* (or tags). Labels are critical for an idiomatic and fruitful usage of code monitoring since they enable aggregating information over different instances. For example, it is typical to introduce many caching objects in an application. We may want to trigger an alert when the hit/miss ratio for any of the caches drops below a certain threshold. To this aim, we could expose a metric that reports the number of `get` operations on any cache of the application, with a label discriminating whether the operation resulted in an *hit* (the cache contains the value) or a *miss*. Each cache should also attach its name to all the relevant metrics attaching. Then, finding the worst-performing cache is as simple as executing the PromQL query in Listing 18. The query executor makes sure to match the hit/miss metric pairs whose other labels are identical, thus making sure that results are not mixed across different caches.

```
min(
    cache_gets_total{result="hit"} /
    cache_gets_total{result="miss"}
)
```

Listing 18: PromQL query which identifies the worst-performing cache in an instrumented application.

B Synchronization primitives

Synchronization primitives are language constructs and special objects that enable orchestrating multithreaded software. Such constructs are designed in a way that allows modeling naturally certain policies to protect resources which shall be accessed concurrently by multiple threads. Their purpose is usually to impose a certain execution policy on a code snippet. For instance, we may want a specific number of threads (and no more) to execute a certain set of instructions at the same time, and to block any other worker until the others have exited the section. Such snippets are commonly called *critical sections* [62]. The main reason for their existence is that we typically try to avoid accessing objects while they are in an inconsistent state [17], which may manifest halfway through complex methods. For example, the simple operation of incrementing an integer is not guaranteed to yield the expected result if a proper synchronization strategy is not put in place [56]. In this section, we will provide some basic definitions and generalities about synchronization primitives, focusing in particular on the JDK ecosystem.

Lock

Locking is the act of preventing concurrent access to a given section, in order to avoid race conditions and undeterministic results. As a consequence, access to the instructions enclosed in the snippet is serialized, thus preventing any concurrency-related problem [135]. Locking is the most basic pattern of synchronization, and it is generally the base for more advanced techniques like semaphores or barriers [38]. The Java programming language provides several lock flavors. The first is embodied by the methods `Object.wait()` and `Object.notify()`. A thread calling `wait()` will block until another thread calls `notify()` on the same object. Any object (except for primitive types) may be used as a "lock" in this sense. Even though locking strategies leveraging this approach are quite simple to put in place, they typically yield quite convoluted code, thus this is not the recommended way to proceed in general [17, 56]. A more idiomatic approach consists in using dedicated lock objects, which in the JDK should be chosen among implementations of the interface `java.util.concurrent.locks.Lock` [110]. A simplified declaration of the `Lock` interface is shown in Listing 19. The Java standard library provides two implementations that should fit most use cases: `ReentrantLock` and `ReentrantReadWriteLock`. The first is a "reentrant" variant of `Lock`: it provides the functionalities listed in the interface, and in addition it makes sure that a thread is able to re-acquire the lock if it has it already. This enables more flexible interaction with the lock and a more natural implementation of complex logic and algorithms. The second is actually a pair of two locks, a *read lock* and a *write lock*

```
interface Lock {
    /**
     * Acquire the lock, or block the thread until the lock becomes available.
     */
    void lock();
    /**
     * Release the lock.
     */
    void unlock();
}
```

Listing 19: Simplified definition of Java's `Lock` interface. Implementations provide additional methods that make the interaction with the object easier and safer.

(usually abbreviated as *RLock* and *WLock*). These locks are used alternatively depending on the nature of the critical section: reading a set of values while preventing intermediate writes from another thread is the typical use-case for *RLock*, while any write operation should be guarded by the *WLock*. These locks have different implementations and semantics [110]. Introducing read/write locks may improve significantly the throughput of concurrent software for certain problems, in particular when read operations outnumber the writes [56]. Another option is to lock a critical section using the `synchronized` block: such a block requires an object to be used as a lock, similarly to `wait()/notify()`, but there's no explicit need for the developer to write any more code. This construct is frequently used because it is quite explicit and easy to spot, and most importantly it minimizes mistakes [17].

Semaphore

A *semaphore* is a generalization of the concept of lock: it is initialized with a number of permits and each thread should acquire one of them before entering the critical section. If no permits are available, the thread will wait until one of the permits is released [135]. Binary semaphores (i.e. semaphores having only one permit) are somewhat equivalent to the locks which we described in the previous paragraph. The usage of multi-permit semaphores is usually restricted to cases when we know in advance the number of workers which may use a shared environment without causing damage. For example, we employed semaphores to restrict the number of threads that may send inputs to Python processes in the context of the UCAP-Python integration (see Section 3.2.2). The JDK ships with an implementation of the semaphore concept in the class `java.util.concurrent.Semaphore`. It provides the standard features of a semaphore, as well as additional methods to simplify the integration with the codebase, e.g. an atomic `try-acquire` to

acquire a permit only if at least one is available without waiting.

Latch and barrier

Latches and *barriers* are quite similar with each other. Both are initialized with an integer value, and will block all threads which try to interact with them. They will only release blocked threads, all at once, when a condition becomes true [135]. For a barrier, the condition consists of the number of blocked threads being equal to the integer values which was passed during the initialization. For a latch, a counter – which is initially set to the value provided to the constructor – should be decremented until it becomes zero. From a practical point of view, barriers are synchronization primitives that synchronize the position of a group of threads, while latches synchronize task completion progresses [56]. Indeed, a thread could "count down" a latch multiple times, but only once for a barrier. In Java latches and barriers are implemented respectively by `CountDownLatch` and `CyclicBarrier`.

Locking limitations

So far we have discussed the building blocks of thread-safe software. These are the most common constructs, and they are typically implemented in the standard libraries of all high-level programming languages with more or less identical semantics. However, the reader should be aware that relying too much on these synchronization primitives could lead to other problems. Indeed, each of the constructs presented in the present section imposes an important performance overhead [56], mostly due to the internal implementation of locks. To mitigate this impact, one could consider several alternatives depending on the use case. A general recommendation is to rely on data structures provided by the standard library [17]. For instance, Java's `ConcurrentHashMap` is a performant thread-safe implementation of the `Map` interface, which provides out-of-the-box striped parallelism across different memory buckets [110]. Another option is restructuring the algorithm to avoid concurrent access to the same resource. While this is not always possible, some computations can be split in order to be fully independent until the very end, which is possibly a simple aggregation of partial results [62]. Finally, one could consider *lock-free* algorithms, which rely on special CPU instructions and data structures to remove the need for locks [148]. An example is the implementation of the JDK's `LongAdder`, which enables highly concurrent summations by splitting all the writes across many memory addresses rather than one, thus reducing memory contention [110].

C Synchronous queues

As we highlighted in Section 2, synchronous queues are critical for robust implementations of concurrent producer/consumer models. For this reason, in the next few paragraphs we are going to spend some words on the typical functionalities we should look for in such data structures. Then, we are going to present several production-ready open-source implementations to give some pointers towards nowadays industry standards.

The Java queue interface

The standard Java `Queue<T>` interface, where `T` represents the type of the objects in the queue, extends the interface `Collection<T>`, thus inheriting all its declared methods like `size()` and `isEmpty()`. In addition, the interface declares two triplets of methods to insert, extract, and view the next element:

- `add(T)/remove()/element()` throw an exception if the operation is incompatible with the current state of the queue (i.e. empty or full);
- `offer(T)/poll()/peek()` return a sentinel value instead (i.e. `null` or `false`).

The interface `BlockingQueue<T>` extends this definition by introducing blocking methods with an optional timeout.

Boundedness

Using an unbounded queue seems a good approach in principle, since then one could retain all unprocessed inputs until they are needed. However, this is a catastrophic idea that should be avoided in production software. It is quite rare to have a perfect balance between producer and consumer: ignoring the transient state of the system, this translates into queues that will asymptotically be empty or growing indefinitely. While the first case is not problematic, the latter has the potential to crash the JVM altogether.

The system may be described naturally using the formalism of *Queue theory*, in particular in the `M/M/1` case. For simplicity, we assume an exponentially distributed arrival time and exponentially distributed service time. `M/M/1` are stochastic systems with a discrete state space coinciding with $\mathbb{N}$, where the state $\{i\}$ is the case "i elements are waiting to be processed in the queue" [143]. It can be shown that the queue is only stable (i.e. the expected number of elements does not diverge to infinity) when $\lambda < \mu$, where λ is the arrival rate and μ is the server

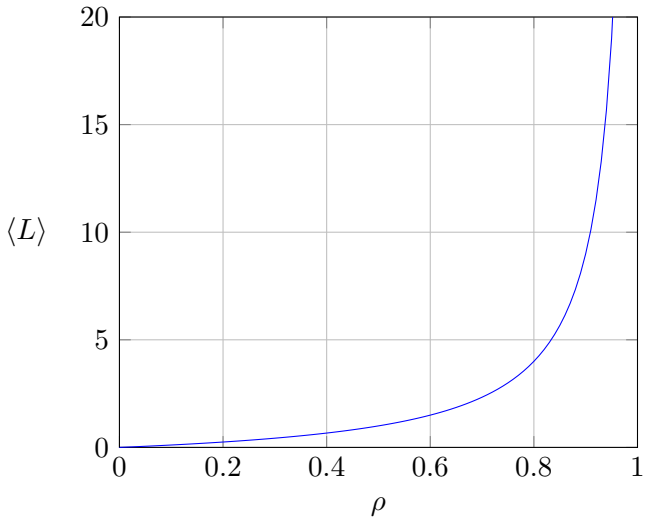

Figure 26: Expected queue size $\langle L \rangle$ for several values of $\rho = \lambda/\mu$ in the inverval $[0, 1)$. The expected size diverges as ρ approaches one.

processing rate. This can be inferred from the fact that the probability that the system contains i items, including both those waiting and those being serviced, is:

$$L_i = (1 - \rho)\rho^i,$$

where $\rho = \lambda/\mu$. Consequently, the expected state $\langle L \rangle$ of the system can be computed as follows:

$$\langle L \rangle = \frac{\rho}{1 - \rho}, \tag{1}$$

and thus the average time spent in the system is $\langle L \rangle / \lambda$, or equivalently $\frac{1}{\mu - \lambda}$. In Figure 26 we plot the value of $\langle L \rangle$ computed with Equation (1) to get an idea of the evolution of the expected crowdedness of the queue as the arrival time approaches the server processing time from the right. The asymptotic growth which $\langle L \rangle$ experiences as ρ approaches the unity is problematic for production software, which cannot sustain the indefinite growth of a data structure that will increase both heap memory occupancy and waiting time for newly acquired work items.

The situation is even worse considering that the arrival time in online data processing platforms is a random variable with a generally strong variance [87]. Moreover, the servicing time is mostly comprised of two factors:

- Inter-process communication time, which in turn is strongly influenced by the selected communication strategy, the data format, the serialization/deserialization tool, and most importantly the size of the input.
- Processing time, which depends on which kind of task shall be performed For example, in UCAP this is yet another source of variability, as the task consists in the execution of a user-defined function, which may be a simple sum, as well as the computation of a DFT.

Each term is driven by variables that are not under the developer control, thus undermining the confidence of any prediction on the state of the queue in the long term. The general recommendation is to avoid the risk of keeping an unstable queue in production [87]: an indefinitely growing queue translates into unbounded waiting time, and most importantly a constant risk of exhausting the heap memory in the JVM [148]. In such a situation all items in the queue would be discarded without any chance to recover them, unless disaster recovery strategies were put in place [15].

Observability

Critical data structures should ideally be monitorable to some extent [93]. We discussed already some ideas in the context of observability in Appendix A, thus the reader should refer to the relevant paragraphs for a brief overview of the topic.

We list several examples of relevant metrics which may be of interest to inspect the runtime state of the queue:

- The current number of values in the queue;
- Mean, median, n-th quantile of the residency in the queue (the *soujourn time*);
- Number of times a producer was blocked (queue full);
- Number of times a consumer was blocked (queue empty).

It requires significant effort to extract these metrics from a running queue, most importantly due to the high number of asynchronous primitives used in the code. For example, the implementation of the method `size()` of `ArrayBlockingQueue` locks the whole queue, which may be unacceptable for low-latency applications, especially considering that the frequency of metrics scrapes should not influence the performance of the software. The situation changes drastically if metrics are acquired on a best-effort basis, which means for instance that there is no requirement to make sure they all refer simultaneously to the same point in time. Such

a weakened monitoring setup is usually enough to provide critical information to observe the queue without damaging its performance.

Implementations

As we mentioned earlier, several production-ready synchronous queue implementations exist on the market for all high-level languages, both embedded in the standard library and as third-party dependencies. The Java package `java.util.concurrent` provides three main implementations [110]:

- `SynchronousQueue` is a bounded queue of size one, which always blocks both insertion and extraction threads until the other end is ready. This data structure may be useful to streamline the implementation of the *naive* solution to the producer/consumer problem which we described at the beginning of Section 2.1.2.
- `ArrayBlockingQueue` is a bounded queue backed by an array. The size shall be specified as a parameter to the constructor, and cannot change for the whole lifetime of the data structure. The backing array is initialized during the construction of the queue, to reduce housekeeping costs when items are added or removed. However, `ArrayBlockingQueue` uses a single lock for both ends of the queue, which is a common compromise between performance and code simplicity. This is fine if we expect insertions and extractions to occur with low frequency, or with a strong imbalance (e.g. slow consumer and fast producer, or vice versa) [130]. In addition, `ArrayBlockingQueue` optionally supports thread *fairness*, which means that threads that have been waiting for the longest time will be served earlier.
- `LinkedBlockingQueue` is backed by a linked list [28], as opposed to `ArrayBlockingQueue`. It does not support fairness, and most importantly it is powered by two separate locks for the head and the tail of the queue. This reduces significantly the problems we highlighted above, and we may prefer this implementation if extraction and insertion occur with balanced frequency. However, we shall keep into account the additional housekeeping costs caused by the frequent object creation and deletion which are typical of linked lists [44].

Python's `queue` module provides several implementations of the queue interface as well [155]. The most important ones are:

- `SimpleQueue`, which implements a Pythonic version of the `BlockingQueue` which we described above;

- `Queue`, which provides a similar interface as `SimpleQueue`, with the addition of some convenient methods to track the progress of processing tasks spawned by each work item.

In general, it is recommended to conduct an extensive benchmark providing the expected values for insertion and extraction frequency, to assess which implementation is most suitable for a specific use-case. The queues presented in this overview are quite general-purpose and may serve just fine in most situations. However, low-latency applications may benefit from more specialized implementations provided by third-party vendors. A common example is LMAX's *Disruptor* [47, 148]. This data structure achieves impressive throughput – the referenced paper contains a comparison against `ArrayBlockingQueue` – thanks to efficient memory usage, clever usage of the JVM Just-in-Time compilation, and dedicated implementations of data structures (e.g. to avoid primitive boxing [47]). This data structure has spread widely due to its strong performance guarantees, including the CESAR project at CERN [52, 123].