Benchmarking the Flight Transport Protocol in Different Languages

Jayjeet Chakraborty, Nayan Sanjay Bhatia, Yash Rajesh Chhabria
jayjeetc@ucsc.edu, nbhatia3@ucsc.edu, ychhabri@ucsc.edu

March 8, 2022

1 Introduction

Apache Arrow [15] is a language-agnostic in-memory data format used for moving structured data between different systems without the need for serialization and deserialization. The project aims to be a standard data format that different systems use when exchanging data amongst themselves. It is an open-source project and was released in 2016. Then onwards, the project has grown to become more than an in-memory data format. Over the years, it developed into a framework of data processing components that can be put together to create a data processing system. Several components from the framework is used by modern data processing systems such as Spark [18], Dask [14], Ray [12], and Ballista [6]. One such component of the Arrow framework is Flight [5]. It is a gRPC-based protocol for transferring tabular data over the network very fast. Flight was announced in 2019 and has been adopted by popular data companies like Dremio since then. Similar to Arrow, the Flight framework is also available in different programming languages such as C++, Java (via JNI), Python (via Cython), R and Rust. In this project, we implement a client-server application with Flight in C++, Java, and Python and performed benchmarking experiments to compare the performance of a Flight application written in different programming languages. We report metrics such as latency, CPU usage, and network throughput along with a discussion on the developer experience of using different programming languages to create a modern data transfer application. The contributions of our work are as follows:

- Implementation of a Flight client-server application in C++, Java, and Python. The link to the GitHub repository is given here.
- Performance comparison of the 3 different implementations based on the programming language they were written in. The performance comparison comprises of latency, CPU usage, and network throughput numbers.
- A discussion on the development experience of the 3 different implementations where we report metrics such as lines-of-code and development time breakdown.

2 Motivation

We have been interested in modern storage and data-processing system's lately and we aim to learn more about the state-of-the-art in these areas. The Apache Arrow and Flight framework is a relatively recent development in this area which we are interested in as a team. We decided to leverage this project as an opportunity to learn more about these technologies and hence we decided to build a client-server application in the different languages it supports and benchmark them. We believe this was a bold first step for us in experimenting more with these technologies in the future.

3 Background

3.1 Apache Arrow

Apache Arrow is an in-memory columnar format optimized for efficient analytics operations on modern hardware. It describes a standard data format for exchanging structured data between different systems. Besides being an in-memory format, it is also a collection of different data processing components that allow building parts of a data processing system. Some of the most-used components are: Flight, a gRPC-based data transfer protocol; Feather, A Arrow-based columnar persistent storage format [17]; Gandiva, An LLVM-based expression compiler [4]; Dataset API, An abstraction for realizing datasets over a directory of files. Arrow is also language-independent as it has APIs in several different programming languages such as C++,

Java, Python, Rust, R, JavaScript, and Julia. Several popular data processing systems such as Spark, Dask, Ray, Pandas, Parquet have added support for Arrow data and Arrow data sources. [8]

3.2 Flight

Arrow Flight is a framework used for building applications to exchange Arrow data streams. Flight takes advantages of the bidirectional streaming support of gRPC [16], which allows clients and servers to send and receive data and metadata to each other simultaneously while maintaining high performance. Flight allows for highly efficient data transfer as it: 1) removes the need for deserialization on the client-side 2) It allows for sending data in a parallel streaming fashion 3) And, allows taking advantage of the Arrow columnar format. When a Flight client connects to a Flight server, it first requests a flight descriptor containing all the metadata about the server and the dataset using the `GetFlightInfo` function defined in the server. It then invokes the `DoGet` function also defined in the server, to get a file descriptor to an Arrow data stream which eventually produces Arrow record batches on iterating. The `GetFlightInfo` and `DoGet` are RPC methods that the Flight client invokes. The received Arrow record batches are then materialized into an Arrow table for further processing. The execution flow of a Flight client-server setup is shown in Figure 1.

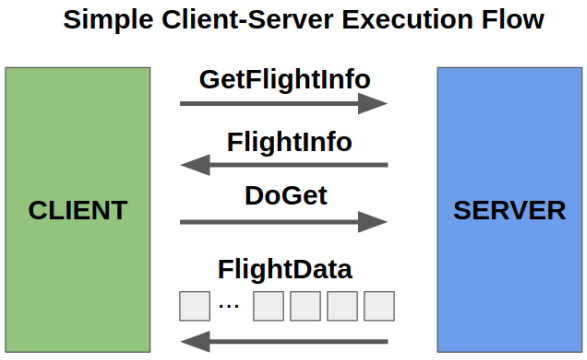

Figure 1: Execution Flow of Arrow Flight [5].

4 Implementation

We implemented a Flight client and server as the main component of the project. The client and server communicate over TCP/IP using the gRPC RPC protocol. Flight transports data in the form of modified Protocol Buffers [13], a protocol buffer version with some performance tweaks. On the server-side, we use the Arrow `Dataset` API [2] to read from a directory of Parquet [1] files. The `Dataset` API is another component of the Arrow framework that allows discovering and reading large and deep hierarchies of datasets containing several partitions. During execution, the `Dataset` API reads a datasets in batches of records sequentially and sends back the record batches to the client through a streaming write API provided by Flight. We used Parquet files as our data format as Parquet is the data storage format of choice in most modern data processing systems in the Hadoop ecosystem. On the client-side, the record batches are retrieved from the Flight streaming interface and are decoded to in-memory Arrow format. Figure 2 shows the high-level architecture of our implementation.

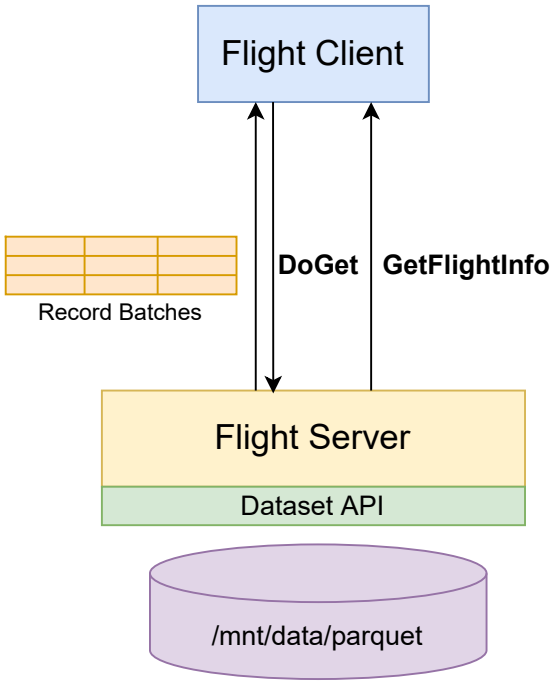

Figure 2: Architecture of a Flight client-server setup.

5 Evaluations

We performed our experiments on CloudLab [9], an NSF-funded bare-metal as a service infrastructure, to evaluate the performance of our implementations. We used the "m510" nodes from CloudLab which have 8-core Intel Xeon-D 2.0GHz processors, 64GB DRAM, 256GB NVMe flash storage, and a 10GbE network interface. The most important aspect of our evaluations were to measure the latency of running a 100% selectivity query where all the rows of the dataset were read in the server and returned to the client. We used a dataset of size approx. 1.5GB containing about 100 16MB Parquet files. The Parquet files were snappy compressed and comprised of data from the infamous NYC taxi dataset. The client and the server ran in different servers and communicated over the private network of 10GbE bandwidth. We ran 15 iterations for each experiment to get somewhat statistically significant results. We ran the experiments in sequential mode to prevent any performance variations due to Python's GIL or any other kind of lock contention due to multi-threading. Also, for all the 3 different implementations, we tried to keep the different parameters such as `batch_size` and `fragment_readahead` consistent, for a fair comparison.

5.1 Performance Comparison

As shown in Figure 3, the C++ and Python implementations performed similarly. This signifies that the Cython [7] wrappers have negligible overhead. Initially, we were not using compiler-optimized libraries for the C++ implementation and that resulted in the C++ version to run slowly. We then raised our issue in the Apache Arrow mailing list and we received guidance on our issue. We then used the compiler-optimized libraries from the Python package of Arrow and were able to get similar results in C++ and Python. This experience reveal that it is really important to have similar optimizations when comparing same applications written in different programming languages.

During our performance evaluations, we found out the Java implementation to be the slowest one. This observation can be attributed the JVM as it is known for cold starts which slows down JVM-based applications significantly. The Java version of Arrow uses the Java Native Interface [19] [11] or JNI to call classes and methods defined in C++ shared libraries. It is basically a wrapper over the C++ libraries. The JNI interface is known to

have a significant performance overhead as the JVM cannot apply a lot of optimizations to the C++ functions as it does for Java functions. Also, the JNI interface results in a lot of data copying between the C++ and JVM layers, hurting the performance even more. [3]

We also observed the CPU and Network utilization of our system while running the experiments. We found that all of C++, Java, and Python use about 90% of the system's CPU. This shows that neither language has a substantial CPU overhead over the other. We also noted almost similar network throughput with all the 3 language implementations where the network throughput came out to be about 300-400MB/s.

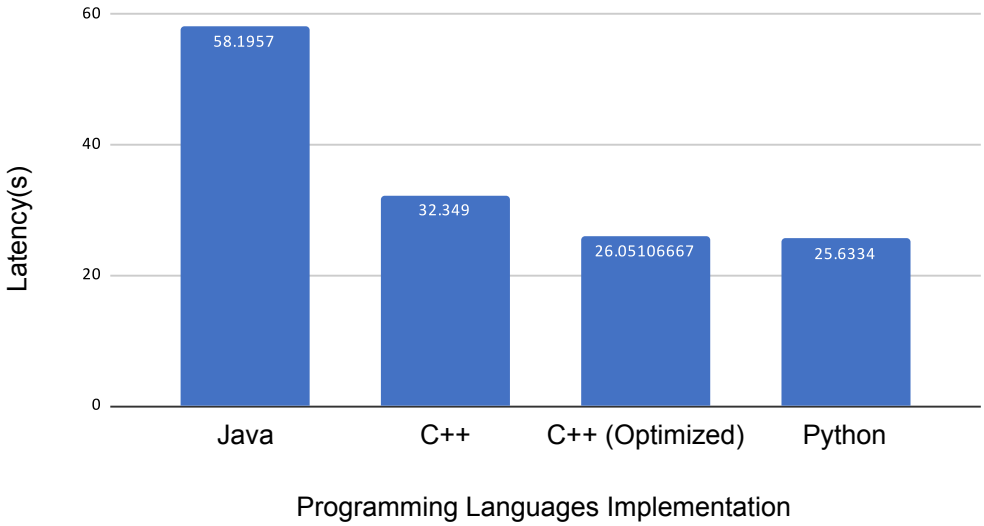

Figure 3: Query latency of reading 100% of the rows in C++, Python, and Java.

5.2 Development Experience

In this section, we discuss the experience we had as developers while using 3 different languages to implement a similar application. All the Pros and Cons mentioned below are purely based upon our experience during the project.

Figure 4 shows the amount of time spent in developing the 3 versions of our Flight client-server setup. We see that Java took the most time due

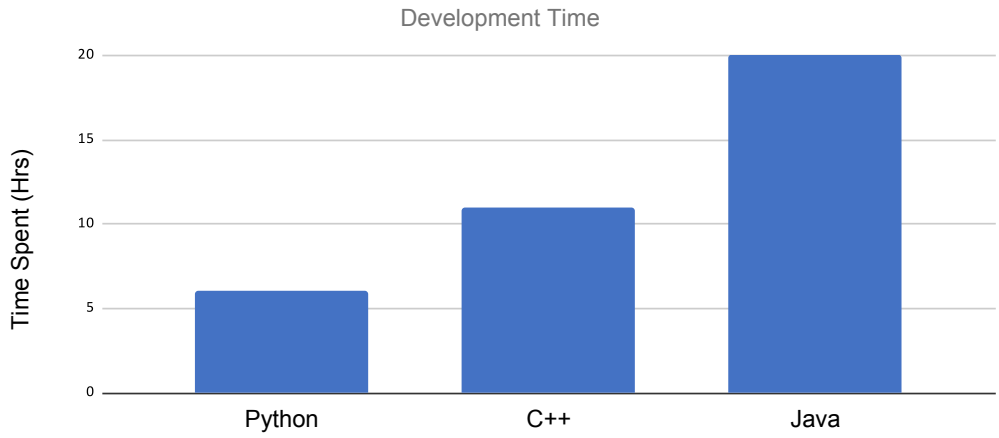

Figure 4: Development time of a Flight client-server in C++, Python, and Java.

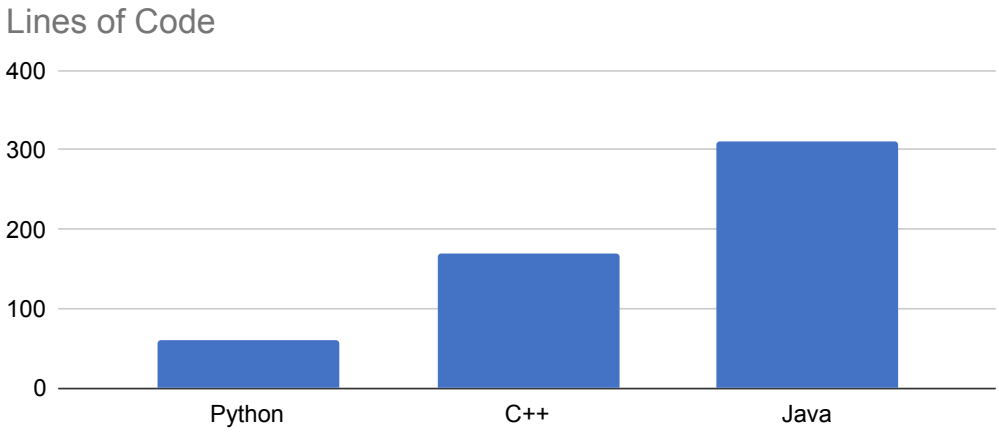

Figure 5: Lines of code in a Flight client-server setup in C++, Python, and Java.

the heavy debugging effort that went into it. Python took the least time as expected due to its user-friendliness. C++ development did not take much time, but we were having some issues with the compiling and linking of C++, which ended up taking a little bit long. The share of the total time that went into the different phases of development is shown in Figure 6.

In Figure 5, we show the lines of code that each implementation took. As expected, the Java implementation took the most lines of code followed by C++ and Python.

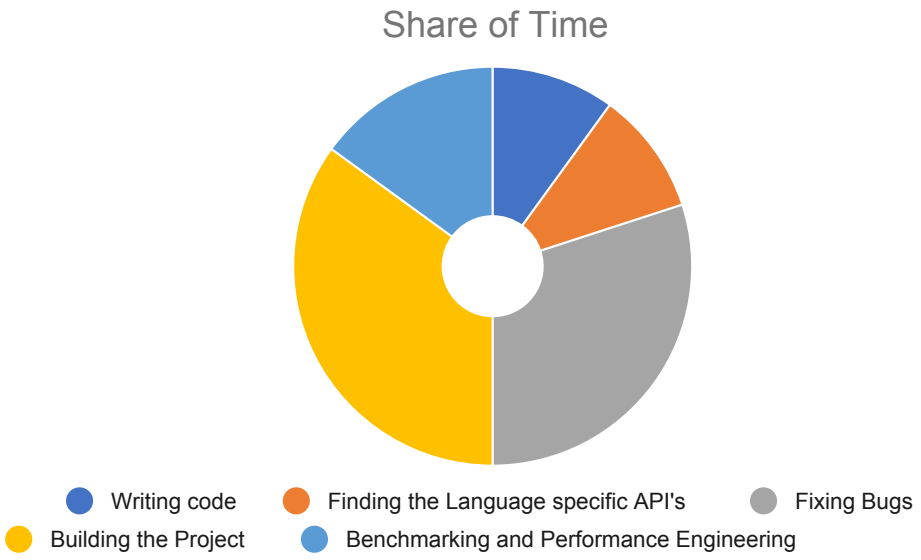

Figure 6: Development time breakdown of a Flight client-server setup.

5.2.1 C++

Pros:

- The C++ library had a high-velocity community around, leading to most developed APIs, great documentation, and good community support. This was mostly because the core of Arrow is written in C++.
- Writing in C++ also helped us learn a lot of intricacies of the language such as command-line arguments, use of smart pointers, and importing and linking with external libraries.

Cons:

- Writing C++ in an efficient manner is actually quite hard. So, if not very necessarily required, it's better not to use C++ and rather adopt some easy-to-use programming language such as Python.
- Debugging C++ code is also very hard due to it's poor stacktrace support. Especially, in the case of errors such as Segmentation Fault where it is required to use debuggers such as `gdb` and `valgrind` to fix the issues.
- Compiling and Linking with 3rd-party libraries is also not straightforward in C++ due to it's lack of user-friendly packaging support.

5.2.2 Python

Pros:

- Python libraries are always in high-demand due to their demand in Data Science/ML/AI applications. For that same reason, the community responsible for the Arrow Python libraries were very active and had created great documentation.
- The sleek library import system of Python makes it easy to use 3rd party libraries in Python code without any compiling/linking hassle.
- Code written in Python usually has less lines of code than C++ and Java. This helps save time when prototyping and evaluating ideas and that's the reason why Python is a popular scripting language.

Cons: While doing our project, we didn't really face any challenges with the Python implementation.

5.2.3 Java

Pros:

- Java has great IDEs such as IntelliJ that makes developing Java an easy and intuitive experience.

- Since Java runs in the Java Virtual Machine (JVM), there isn't generally any platform specific issues in Java as in other languages such as C/C++.
- Similar to Python, Java also has a huge library of utilities which makes developing complex applications relatively easy and does not require importing a lot of 3rd-party libraries which could easily become a software engineering hassle.

Cons:

- Coding in Java is very verbose and sometimes leads to complex and difficult to read code.
- Java packaging systems `maven` is decades old and often leads to transient errors when run outside of an IDE. Also, the package repositories of Java are sometimes outdated and broken.

6 Future Work

As future work, we aim to compare the performance of Arrow's native serialization/deserialization based data transfer with Arrow Flight. This would help us understand if how much of an performance improvement does Arrow Flight provide over native Arrow data transfer. We would also like to augment our study of the client-server performance in different programming languages with a Flame Graph [10] for each language implementation. Such root cause analysis will help us better understand the reasons behind the performance difference observation. Rust is a language which is gaining immense reputation in the system's research community. Since, both Arrow and Flight has libraries in Rust, we would like to implement a Rust-based Flight client and server, benchmark it, and compare with the results of the C++, Java, and Python implementations.

7 Conclusion

Arrow Flight is a modern gRPC-based data transport protocol developed to efficiently transfer Arrow datasets over the network. It was developed as an improvement over traditional JDBC/ODBC protocols which are more

suited for OLTP-style data access. The Flight protocol aims to optimize OLAP-style access over the network where large column chunks are transferred at once. In this work, we leverage the Flight framework to implement client-server applications in 3 different programming languages to transfer Arrow data over the wire. We measure metrics such as query latency, CPU utilization, and network throughput. We find that the C++ and Python implementations perform similarly while the Java implementation is quite slow because of the JVM and JNI overheads. The fact that C++ and Python implementations are equally performant signifies that the Cython wrappers which the Python version of Arrow uses, does not have a significant overhead. Besides quantitative evaluations, we also discuss the development experience that we went through when implementing the client-server in the different languages. We experienced Python to be the most developer/user friendly, followed by C++ and Java.

References

[1] Apache parquet. https://parquet.apache.org/.

[2] Arrow dataset api. https://arrow.apache.org/docs/python/dataset.html.

[3] Understand the overhead of jni. http://golangcloud.blogspot.com/2012/05/understand-overhead-of-jni.html, 2012.

[4] Gandiva: A llvm-based analytical expression compiler for apache arrow. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/, 2018.

[5] Introducing apache arrow flight: A framework for fast data transport. https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/, 2019.

[6] Ballista: A distributed scheduler for apache arrow. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/, 2021.

[7] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2010.

[8] Jayjeet Chakraborty, Ivo Jimenez, Sebastiaan Alvarez Rodriguez, Alexandru Uta, Jeff LeFevre, and Carlos Maltzahn. Skyhook: Towards an arrow-native storage system. *CCGrid*, 2022.

[9] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[10] Brendan Gregg. The flame graph: This visualization of software execution is a new necessity for performance profiling and debugging. *Queue*, 14(2):91–110, 2016.

[11] Sheng Liang. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.

[12] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association.

[13] Sran Popić, Dražen Pezer, Bojan Mrazovac, and Nikola Teslić. Performance evaluation of using protocol buffers in the internet of things communication. In *2016 International Conference on Smart Systems and Technologies (SST)*, pages 261–265. IEEE, 2016.

[14] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*, volume 126. Citeseer, 2015.

[15] Arrow Development Team. Apache arrow. https://arrow.apache.org, 10 2018.

[16] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. Grpc: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review*, 27(3):75–86, 1993.

[17] Hadley Wickham. Feather: A fast on-disk format for data frames for r and python, powered by apache arrow. https://www.rstudio.com/blog/feather/.

[18] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[19] Hongze Zhang. Arrow-7808: [java][dataset] implement dataset java api by jni to c++. https://github.com/zhztheplayer/arrow-1/tree/ARROW-7808. Accessed: 2020-09-14.