Framework for Uploading Research data (FUR)

Niels Andreas Tyndeskov Voetmann, kfn536@alumni.ku.dk

Primary supervisor: Kenneth Skovhede, skovhede@nbi.ku.dk,
Co-supervisor: David Gray Marchant, david.marchant@nbi.ku.dk
Co-supervisor: Carl-Johannes Johnsen, cjjohnsen@nbi.ku.dk

March 15, 2021

Abstract

Capturing and storing large amounts of data on systems connected to digital measurement tools can be taxing on local storage capabilities. Therefore, captured data is usually transferred to a remote storage solution that offers centralized access to datasets. The transferred data can be packaged in advanced file formats in order to prepare it for being used efficiently in data analysis. The processes of packaging and uploading data are normally performed separately when data capturing has completed. This leaves room for improvement, as data entries can be packaged and uploaded continuously as they become available, instead of waiting for the entire data capturing process to complete.

This thesis presents the FUR framework to automate uploading, packaging and cleanup of data stored on systems connected to digital measurement tools. The main feature of FUR is continuously uploading data in batches to a HDF5 file stored on a remote storage solution and automatically remove local data once it has been uploaded. FUR uses remote file objects that are opened with the SFTP protocol to manage transferring data.

The results of testing the FUR implementation indicate that it is able to match the uploading speed of standard SFTP uploading methods when uploading typed data to a remote HDF5 file. The results also showed that using the implementation of FUR for uploading binary data to a remote HDF5 file introduces additional overhead, which leaves room for improvement. Finally, the results also indicate that FUR works cross platform and is scalable when used with faster bandwidth connections.

Abbreviations:

- FUR : Framework for Uploading Research data
- I/O : Input/Output
- RTT : Round Trip Time
- HDF5: Hierarchical Data Format 5
- TCP : Transmission Control Protocol
- SSH : Secure Shell
- SFTP : SSH File Transfer Protocol
- FTPS : File Transfer Protocol Secure
- RAM : Random Access Memory

Contents

1 Introduction

Digital measurement tools have a wide range of uses in many industries, but need efficient software and frameworks in order to analyze and automate tasks. Digital measurement tools, such as cameras, are usually connected to an embedded or local system that is able to establish network connections. These systems can manage the process of producing and storing data locally, but need software to help the process of uploading the produced data from the local system to a remote storage solution. As the amount of data produced by digital measurement tools can be massive and taxing to store on local resources, it calls for software to aid automate tasks such as cleanup, packaging and buffering of locally stored data in order to assist the process of uploading it to remote storage solutions.

Cloud storage solutions, such as ERDA[1] offer remote access to centralized storage, powerful processing capabilities and scalability beyond the capabilities of personal computers. This allows for simulations, or data analysis, relying on access to large datasets and performing resource heavy computations to move towards these solutions in order to optimize these processes. With this in mind it is important that new and improved software solutions are developed in order to enable the use of current and future cloud storage solutions. As data analysis performed through resources of cloud storage solutions rely on having given datasets available, it is important to have efficient tools for uploading and storing data in efficient and advanced file formats.

This thesis introduces Framework for Uploading Research data (FUR) that will address the abovementioned problems. It is a framework designed for automating and optimizing the process of uploading data to remote storage solutions in the Hierarchical Data Format version 5 (HDF5) [2] file format.

1.1 Motivation

In preparation of the thesis, the Center for Advanced Bioimaging Denmark (CAB) [3] was contacted to inquire about the potential need of a framework, such as the one presented in this thesis. The department, and general field, rely on high resolution images taken with various filters and lighting using advanced microscopes. The current state of their microscopes is that they are connected to a local University of Copenhagen science computer with no cloud integration, therefore making access

to produced image data rely on physical presence and alternate storage solutions. No current automated solutions were known or used, and manual solutions such as USB's are used to transfer and access data from the locally connected computer. Furthermore, the storage of the computer system connected to the microscopes has to be manually deleted by an office administrator, as the amount of locally stored image data fills up the storage capabilities of the system. The problems of CAB fit the problems that the framework presented in this thesis tries to solve, as both the issue of data portability by automating uploads and cleanup of local files are addressed.

1.2 Pre-project

In preparation for the presented thesis a pre-project [4] was conducted using an industrial camera connected to an Embedded Linux system as the digital measurement tool. The pre-project was conducted in order to gather knowledge and experience with regards to the challenges of efficiently producing, packaging and uploading image data concurrently with data production. The aim of the pre-project was to tailor a solution to the specific Embedded Linux system by trying to optimize all steps of the data flow from the system to a remote storage solution, based on hardware and bandwidth limitations. The presented solution in the pre-project consisted of optimizing compression of the produced image data, using RAM for temporary storage and packaging batches of image data in the HDF5 file format before uploading it. When all the HDF5 files had been uploaded to the remote storage solution, an internal resource from ERDA was specified to perform a job mounting the folder containing the uploaded files and merging all HDF5 files into one main HDF5 file. The architecture of the solution is shown in figure 1.

The solution presented in the pre-project did solve the specific problem for the specific system, but did not present a generalized solution that can avoid the extra step of post processing files after upload (as indicated by the orange box in figure 1). Some paragraphs and figures in this paper are similar or used from the pre-project paper, the relevant parts will have a notation.

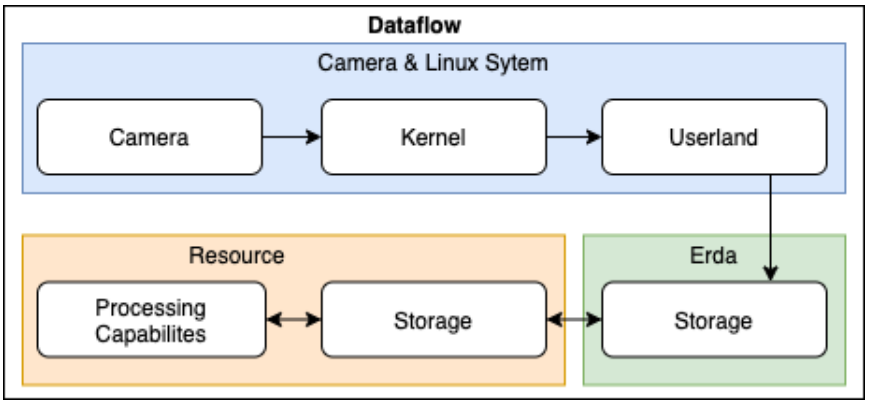

Figure 1: Architecture of Embedded Linux Camera and remote storage solution from pre-project, showing the flow of data. Blue indicates the local Camera and Embedded Linux System, Green indicates ERDA and Yellow indicates a resource within the ERDA network

1.3 Related work

h5s3:
`h5s3` [5] is a driver developed for client-side access to HDF5 files stored on Amazon Simple Storage Service (Amazon S3) [6]. Amazon s3 is a cloud storage solution that offers storing data objects remotely. The driver is compatible to be used in Python applications along with `h5py` [7]. The `h5s3` driver work with HDF5 versions 1.8 and 1.10, which allows for efficiently working with HDF5 files by having them transparently stored in Amazon s3. Features of the `h5s3` driver include fetching subsets of remotely stored HDF5 files, which can help manage distributed computation when working with large remotely stored HDF5 files. The `h5s3` driver has not been updated since January 2018, which indicates that it is no longer maintained and new solutions are needed.

While `h5s3` does deal with interaction with remotely stored HDF5 files, it does not deal with how to efficiently and continuously upload data to a HDF5 file through a remote connection. Furthermore, the solution is not general from a server perspective, as it only works with Amazon s3. As FUR strives to be general and these were some of the reasons why FUR did not expand upon `h5s3`. However, `h5s3` did provide inspiration for the creation of FUR, as it showed that it is possible to create libraries using h5py to perform file interaction with remotely stored HDF5 files.

Filezilla and WinSCP:
Filezilla [8] and WinSCP [9] are both open source cross-platform applications supporting multiple file transfer functionality and protocols such as FTP, FTPS, SFTP, SCP, etc. They also provide services beyond standard file downloads/uploads between clients and servers, including file synchronisation, upload/download automation, and allowing pausing and resuming of file transfers.

The two abovementioned solutions are great for standard file transferring functionality and can provide easy access to the functionality through graphical user interfaces. File synchronisation could be used for continuously uploading a growing HDF file to a server, but the method does include the need of having to store the entire HDF5 file locally. In addition, synchronizing files or folders introduces a lot of overhead, which could result in the method quickly will become slow and insufficient, especially with large amounts of data. This could result in a solution that might be infeasible due to limited storage space of the client system, or result in a solution that is slow and makes bad use of resources. So while they are great applications for standard file transfer functionality, they fall a bit short when it comes to solving the introduced problems.

General Approaches:
There are currently two fundamental approaches to transferring data to a remote storage solution in a single file using an advanced file format, such as the HDF5 file format. The first approach is to package and format the data into a HDF5 file locally either during or after data capturing. When the formatted file is completed on the local system, the HDF5 file is then uploaded. The second approach is to upload data as-is and format it to a single HDF5 file through the use of a script run by internal resources of the remote storage solution. Both approaches have their cons, such as wasting unnecessary time and storage space during the data capturing process, or inefficiently using available resources during data transfer from client to remote storage solution. In addition, the second approach consisting of server-side formatting might not be available through remote connections, as server environments intended for remote access by several users, generally allow limited options for remote file interaction and the possibility of running scripts server-side.

1.4 New approach and proposed solution

This thesis presents the Framework for Uploading Research data (FUR) that uses a new approach for uploading and storing data in the HDF5 file format by using remote file objects. The approach is to perform continuous formatting of data during upload to a remote HDF5 file object via the possibilities and functionality of the SSH [10] File Transfer Protocol (SFTP) [11]. The introduced approach will provide the possibility of uploading data to a remote file in the HDF5 format, while running concurrently with data production, and aiming to maximize the speed potential of the data transfer. The idea is to minimize overhead in the implementation of the introduced upload approach in FUR, such that FUR is able to make efficient use of time and resources that are not fully utilized in the two introduced general approaches. FUR will also allow for the possibility to perform cleanup of locally stored data once it has been uploaded. For the implementation of FUR to be a success, it should perform just as well as standard uploading methods, while automating the process of storing data in the HDF5 file format on remote storage solutions.

In order for FUR to be as general as possible, it will aim to be flexible with regard to different data types produced by digital measurement tools. It will also try to be adaptable in regard to hardware specifications, bandwidth connections and other limitations of a system connected to a digital measurement tool. Furthermore, the solution will take into account that some systems might want to have FUR work in a continuous manner, running concurrently with the data production, while other systems want to wait for the data production to complete, before using FUR.

The working space of the framework will be client-side, where it will handle the packaging, buffering, cleanup and upload of data to a remote storage solution. It will be implemented in Python 3, tested on three separate systems with different capabilities, operating systems, bandwidth connections, etc., to ensure that the implementation of FUR works cross-platform and scales well. The framework is designed to work with ERDA [1] and tested with ERDA as a remote storage solution, but the framework aims to be a general solution that can work with other remote storage solutions as long as they support SFTP.

1.5 Outline

The remainder of the thesis is structured as follows:

- Section 2 presents the main properties and functionality of the HDF5 format and its features, which are relevant for the work in this thesis.
- Section 3 briefly covers information regarding the dynamic of clients and servers, file transfers, pipeling, remote file objects and transfer protocols. At the end of the section the design of FUR is presented.
- Section 4 covers the Python implementation of the FUR framework. The subsections focus on covering the implementation and optimization process of the HDF5 appending upload method, which is the main feature of the framework.
- Section 5 introduces the systems used for benchmarking and production of test data.
- Section 6 shows and reflects upon results of upload speeds for uploads performed by the FUR implementation. The benchmarks will give an indication of the performance of the FUR implementation of the appending upload method and compare them to the performance of other uploading methods.
- Section 7 features discussions of the used test setup, and advantages and limitations in the design and implementation of FUR.
- Section 8 discusses future work and possible improvements of the FUR framework.
- Section 9 concludes upon the work in this thesis.

2 The HDF5 file format

This section introduces the HDF5 file format and its features such as groups, datasets and chunked storage. Knowing the properties of the HDF5 file format and its functionalities will be critical to analysing and improving the performance of how differently typed data is stored in the format. Therefore, knowledge from this section will be important in the implementation of the HDF5 appending upload method.

The Hierarchical Data Format version 5 (HDF5)[2] is a file format developed by the non-profit HDF5 group. It is a file format designed for efficiently storing and accessing scientific data and is intended for large scale data problems. It can be used to store many different data types including complex types, such as complex numbers. Storing different data types together in a single file through the use of multiple datasets and groupings is natively supported. The file format is designed with fast I/O performance in mind and has a rich set of features for optimizing this. The format also supports the traditional contiguous storage of data to disk. Some examples of storage optimization and I/O performance features are chunked data storage and creation of datasets with transparent compression using various compression filters such as GZIP [12]. This allows for automatic compression of data on its way to disk and automatic decompression when data is read, when I/O operations are performed with a file in the format using the HDF5 library.

These features and the versatility of the format are some of the reasons as to why the format provides excellent speed for both reading and writing, while minimizing the amount of disk storage used. Furthermore, the HDF5 file formats supports parallel I/O processing through the Message Passing Interface(MPI) [13], a high performance message passing library built for use in parallel or multi-threaded applications. This is a strong feature that allow the HDF5 format to be used for storing large data instances used by simulations and data analysis that rely on MPI. An example of the structure of an HDF5 file can be seen in figure 2.
The HDF5 file format is cross-platform and supported in multiple languages such as C, C++, Python, Fortran through the HDF5 library API [15].

2.1 Groups and Datasets

A group in the HDF5 file format is a container that contains 0 or more groups or datasets. When a HDF5 files is created the root group , denoted by "/", is created

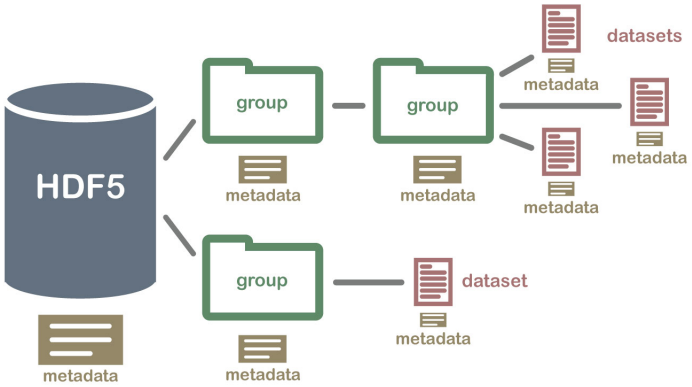

Figure 2: Structure of an example HDF5 file displayed as a tree with its groups and datasets. The grey cylinder depicts both the HDF5 file and the root group. Green folder symbols depict groups and red file symbols depict datasets. Metadata associated with a group or dataset is shown in brown beneath the given group or dataset. Image is taken from www.neonscience.org [14]

and it has the special rule that it may not belong to any other groups. Groups are the backbone of how HDF5 files are structured and they also contain meta data regarding the given group they describe. The structure of groups within the HDF5 format can be a tree structure like in figure 2 or as directed graph with the root group as the specified entry point. Groups in HDF5 are comparable to folders known from the file systems seen in the POSIX family, while datasets are comparable to POSIX files. This is also seen in figure 2 with groups depicted as folders and datasets as files.

Datasets are objects within HDF5 files containing data, along with all relevant information about that data with regards to reading from and writing to it. Datasets are stored in two parts as a header file and a homogeneous array. The header file contains relevant information about how the array is stored on disk. This includes how to interpret the array and other metadata associated with the dataset, such as the dataset object name and dataset dimensionality. As mentioned in the beginning of this section, datasets in HDF5 have a rich set of transparent storage features such as compression filters and chunked storage. These features and other properties of the datasets are specified upon creation of datasets and are unchangeable thereafter, as to ensure the integrity of all stored data throughout the lifespan of a dataset. Com-

pression Filters are stored as a part of the dataset and can not be removed, added or changed after the creation of the dataset. After creation of a HDF5 dataset, the HDF5 library handles locating chunked data and the use of compression filters when data from the dataset is accessed, allowing for applications to read from and write to a dataset without worrying about its underlying features.

2.2 Chunked Storage

Chunked data storage splits the array of a dataset into several equally sized chunks of data, which are stored separately on disk. If dataset chunking is not specified in the creation of a HDF5 dataset, the array of the dataset is stored as row-major contiguously on disk. An example of this method for disk storage can be seen in figure 3, where a 10x10 2d array is stored contiguously. In the left part of the figure, 5 adjacent array indexes from a row are accessed, which does not cause any concerns as they are adjacently stored on disk. In the right part of the figure, a problem of using contiguous storage is showcased. If the dataset is going to be used for accessing adjacent array indexes from a column of the array, the read of data will be inefficient, as this data is not stored adjacent on disk. In order to read these 5 array indexes, 5 rows equalling 50 array indexes need to be read, which is inefficient as only 5 array indexes are needed. The same problem could occur if data was stored column-major and the five adjacent indexes were from a row.

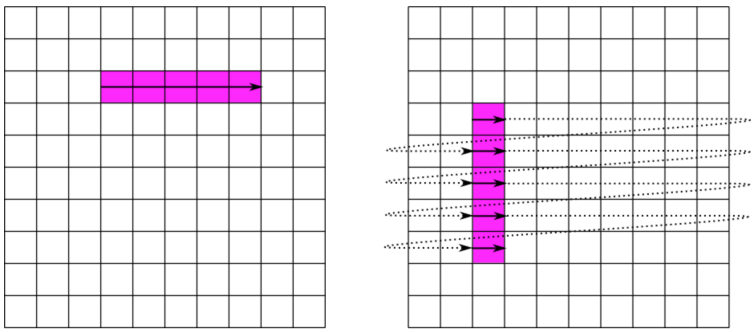

Figure 3: 10x10 array stored row-major and contiguously. Some indexes are highlighted with purple color and overlaying arrows indicate how many total indexes are needed to be read in order to access these highlighted indexes. Image taken from portal.hdfgroup.org [16]

The solution to the problem in figure 3 is shown in figure 4. Here the 10x10 array is stored in chunks of 1x10, resulting in adjacent column indexes in the array being

stored adjacent on disk. The chunking of the dataset then allows for reading 5 adjacent column indexes without having to read unnecessary data from disk.
Chunking of datasets is not limited to splitting the data into whole rows and columns. Chunks can be any shape as long as the shape of the dataset is a multiple of the chunking shape. The array from figure 3 could e.g. be split into 4 chunks of shape 5x5 or 5 chunks of shape 2x10, depending on what makes sense for the purpose of the dataset.

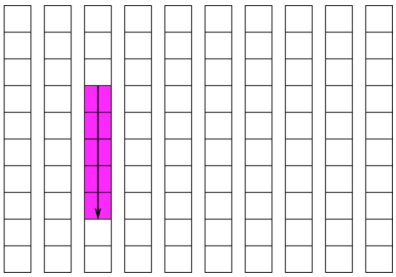

Figure 4: The array from figure 3 stored with chunks of shape 10x1. Some indexes are highlighted with purple color and overlaying arrows indicate how many total indexes are needed to be read in order to access these highlighted indexes. Image taken from portal.hdfgroup.org [16]

Chunking also allows for resizable datasets in HDF5 to exist, as new data can be stored flexibly at available space on disk, as chunks are stored separately on disk. Chunking is automatically enabled for resizable dataset, as a HDF5 file does not control if the disk storage adjacent to the storage used for the dataset array on disk is used for other storage purposes.

Even though chunking provides HDF5 datasets with great features and storage possibilities that can result in improved I/O performance, they can prove inefficient if used incorrectly. The sizes and amount of chunks used for storing data are two measures that can have a big impact on performance if used efficiently or inefficiently. Every chunk in a dataset requires the dataset to store its metadata, resulting in file size growth. If chunk sizes are specified to be too small this might result in file sizes that are unnecessarily large, compared to the actual data that the file stores. Having many chunks also increases the height of the b-tree indexing the dataset, resulting in longer search times. On the other side, if chunk sizes are too big it can also have negative consequences for performance. Whenever a chunk is read, it is read

in its entirety from disk to RAM. In the case of use of compression filters the whole chunk is also needed to allow for decompression, before being available for further processing. This can cause reads to be expensive if the chunk size is too big. In the case that only partial parts of the data is needed, it will make data reads from the dataset very inefficient. Therefore, if I/O performance is important to users of the HDF5 file format, it can prove useful to analyze data and its uses before storing it in the format.

2.3 Indexing and b-trees

In the HDF5 format b-trees [17] are used for indexing chunks of data from a dataset on disk storage. The b-tree data structure is also used throughout the HDF5 format whenever a data structure needs indexing. A b-tree is a self-balancing tree data structure. The data structure is well suited for indexing disk storage, as it can drastically reduce the amount of disk reads when locating data on disk. This is possible since the amount of keys and children in b-tree nodes are limited by the order of the b-tree. This means that a b-tree of order 500, has the possibility to eliminate close to 500 subtrees for each step in its search through indexed disk storage. In comparison, a standard self-balancing binary tree eliminates a single subtree at every step, and the height of it will generally be bigger than a b-tree storing the same information. The standard binary tree might be faster when used for indexing and searching through data that fits into RAM. This is due to b-trees needing to search through the current node, which holds an amount of keys between the order of the tree divided by two and the order, before going to the next node in a search. However in locating data stored on disk, reads from disk will be the bottleneck of the operation. The b-tree data structure can make better use of read block of data, compared to a standard self-balancing binary tree, due to its possibility of keeping multiple keys in each node. The specific uses and implementations of b-trees used in the HDF5 file format can be found in the HDF5 documentation [18].

3 Data transfers and design of framework

This section briefly covers the dynamic of clients and servers, transfer & application protocols, pipelining, remote file objects and general challenges in file transfers. This information will be important when designing the FUR framework and provide background knowledge of tools used in its implementation. At the end of the section the design and thought process behind the FUR framework is explained.

3.1 Client, Server & network protocols

The sender-receiver model is used for describing processes that are communicating with each other, where a sender is defined by being the process initiating the connection. Processes may be running on the same system, but are typically running on separate systems. Communication in this model consists of transferring data between processes. An example is that the user client in the FUR framework acts a sender in this model and the remote storage solution (server) acts a receiver.

In order for a client and server to communicate with each other they need to understand the data packets that are sent between them. This part of the communication is done through application layer protocols that packages and formats data. A connection between client and server is established and handled through a transport layer protocol through the notion of sockets. Transport layer protocols and underlying protocols work in conjunction with application protocols in order to aid data transfers between clients and servers. There are several transport layer protocols, but two of the most popularly used are the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). These two protocols have significant differences, as TCP is a connection-oriented protocol, whereas UDP is connectionless. Being connection-oriented means that both client and server keep a state of the connection and communication is two-way. This allows for the server to give acknowledgements to clients whenever data packets are received serverside. It also means that the client through protocol communication can be alerted when packet loss occurs, allowing for re-transmission of unaccounted packets. This makes connection-oriented protocols useful for file transfers, as we want remotely transferred files to be 100 percent intact when a file transfer is complete. The main drawback of connection-oriented protocols is that the communication does introduce additional overhead in the communication process. The communication of connectionless protocols are one-way from client to server. They can prove useful for applications such as video streaming or multiplayer-gaming, where packet loss is less consequential and unnoticeable if kept minimal. The downside to connectionless protocols are that they do not guarantee that all packets are received serverside. The upside is that they have low overhead, specially when compared to connection-oriented protocols.

The main limitation when building applications that rely on processes communicating across systems is that both the client and server system need to support an

identical application protocol. There are several possibilities for optimizing the use of transfer protocols for efficient data transfers, such as providing local buffers to aid the flow of data transfer or protocol pipelining to minimize protocol overhead. The two protocols discussed and used in FUR are SSH File Transfer Protocol (SFTP) and File Transfer Protocol Secure (FTPS) [19], which both are connection-oriented protocols that use TCP as the underlying transport protocol. Both SFTP and FTPS allow for authentication of user data, verification of server key fingerprint, encryption of application data, etc. Both standard methods for file uploads with SFTP and FTPS require that data intended for transfer with the protocols needs to be stored locally in files.

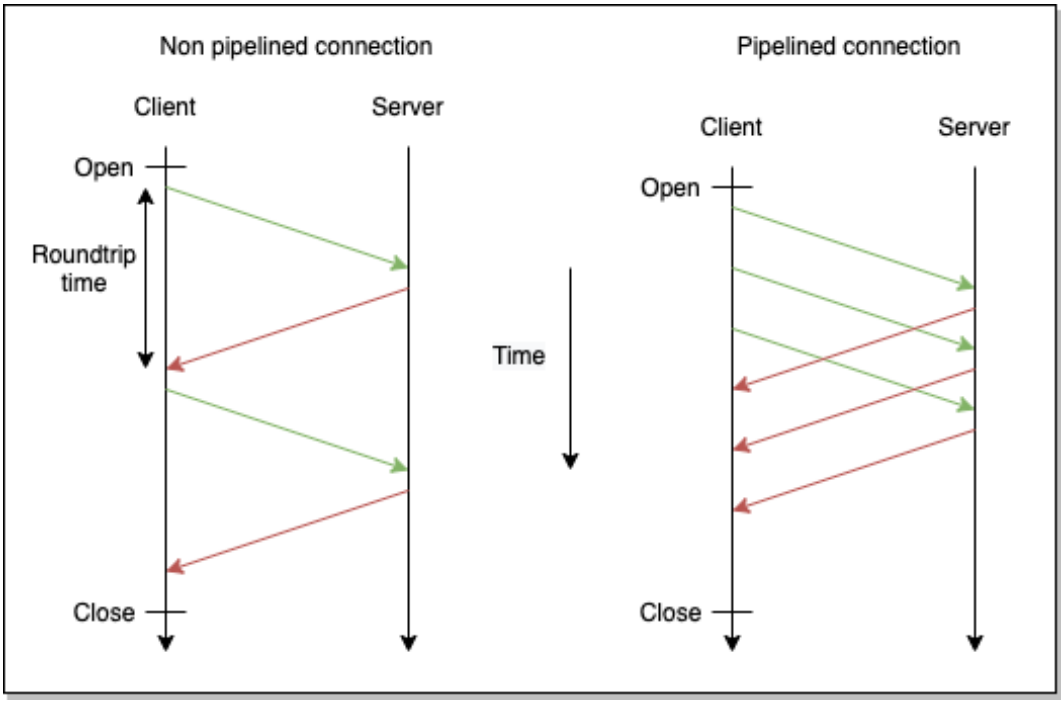

Figure 5: Protocol pipeling in connections. Green arrows indicate data packets from client to sever and red arrows the reverse.

An important factor in communication between client and server is the notion of Round Trip Time (RTT) for connection-oriented protocols. RTT is the time that passes from when a client sends a packet to a server until the client receives acknowledgement of that packet being received from the server. An example of RTT is

depicted in figure 5. A connection-oriented protocol will want to acknowledge that all packets have been received before sending more packets, sometimes waiting for acknowledgements of individual packets or using an acknowledgement for a group of packets. In either case, The RTT can become a significant factor in the rate of file transfers if the transfer protocol is not managed properly. In order to avoid RTT from causing slow transfer rates, it is possible to pipeline a protocol. Figure 5 shows how pipelining works, by sending packets from a client without waiting for acknowledgements from the server, hence improving the performance of the protocol. The use of pipelining introduces some risks in the file transfer, such as handling which packet(s) where lost during a packet loss. This can result in a client application having loaded new data ready for transport, but having to discard the work of loading and preparing that data, in order to solve the data loss issue of the lost packet(s).

TCP is also generally inefficient at uploading smaller files due to it using the slow start [20] algorithm for handling network congestion. The slow start algorithm starts of by sending a minimal amount of data packets, and then exponentially ramps up the amount upon receiving acknowledgements, until one of three situations occur. The first is that data loss occurs, the second is that the Slow Start Threshold (SST) is reached and the third is that the servers receive window limit is reached. When packet loss occurs, TCP saves the amount of packets that were sent during the packet loss. This amount is divided by two and saved as the new SST for the connection. Then the protocol either resets to sending the a minimal number of packets or sending packets equal to the new SST, depending on TCP implementation. If TCP instead has reached the SST, it would start to scale the amount of packets sent linearly, which is also what happens in the case of a TCP connection that has had packet loss reaches the SST.

Lastly, bandwidth is the maximum speed of a given connection either for receiving or sending data packets. In the case of a connection between a client and server, the lowest throughput of the bandwidth connections of server and client is what determines the theoretically achievable transfer rate. In order to fully optimize file transfers, it is the goal of both client and server to optimize the use of protocols and hardware in order to saturate the available bandwidth. The bandwidth is also reliant on many other factors such as network cards, cabling, routers, etc. As these areas are not the focus or meddled with in this thesis, all they get is a mention

acknowledging their importance in determining the rate of data transfers.

3.2 Handling remote files via application protocols

Some application protocols, such as SFTP and FTPS, grant users the possibility of creating and deleting remote files, and performing file operations on remote files, through the notion of remote file objects. This allows client applications more possibilities in regards to transferring data in comparison to standard file uploads. Using remote file objects, data packets are still sent and received as per usual, however the packets sent can now contain instructions for file operations, which the server can interpret, along with actual data that needs to be written to a file. This means that all work of loading and preparing data is still done and controlled by the client. Working with remote file objects is identical to working with normal file objects from an application point of view. However, some remote file objects only allow for a limited subset of file operations. Reading and writing to files are I/O bound operations, being dependent on the speed of disk reads and writes instead of CPU speed. This means that in addition to being limited by the speed of disk read and writes on the server, file operations and interaction performed with remote file objects are also limited by the throughput of the used connection. As with local file objects, writing as sequentially as possible to disk will perform best, as it reduces the impact of the overhead introduced by disk access and moving of file pointers. This means that in order to optimize storing received data serverside, it would be most efficient if data is received in sequential fashion. TCP sends data in sequential order, but does not guarantee that data packets are received in a sequential fashion serverside.

3.3 Analysis and Design

The design of the FUR framework will be based on trying to optimize the upload speed between clients and servers, while storing data in the HDF5 file format. It will do so based upon the possibilities and limitations covered by the topics in sections 2 and 3. FUR is designed for being used with ERDA, but is intended to work with remote storage solutions as long as they support SFTP or FTPS.

In order to define the role of FUR it is important to note what it expects from the data it is going to process and upload, as FUR does not aid in production of data. FUR assumes that incoming data is produced by other applications and stored locally in files, in a manner such that the data files are available to an application running the framework. Furthermore, FUR expects the data files to be stored in the

same location and be named in a sequential fashion, based on how they should be ordered in packaging and uploading functionalities of FUR. The reasoning behind this design choice is that scientific data, such as image data or time series, are usually dependent on the order in which data was captured. The last expectations of the incoming data is that the users must specify the total number of data files that are going to be handled for the given use case.

The focus in the design of FUR is to make sure that the overhead of the work performed by the framework has as minimal impact as possible, while optimizing the rate of data transfers. This means that the focus of the design will try minimize time spent not uploading data and optimize the upload speed when uploads are occurring. As bandwidth most likely will act as the major bottleneck of FUR, it is important that FUR tries to utilize the given bandwidth connection as best as possible. This will help FUR stay relevant, as bandwidth connections grow faster.

There are three main cases between the relation of the data production rate and the data transfer rate that FUR has to take into account. The first case is that the frameworks is able to transfer data faster than the rate of data production and the second being the reverse situation. The third option is that all data production has finished before the framework is invoked. It would be optimal if the first case was always true, but the design of the framework has to take some measures if it is not. Therefore, whenever data is packaged in FUR it will be stored in files on disk. Storing packaged data in RAM is also an option, which for local I/O purposes should be faster. However, as systems usually have more disk storage than RAM, and local I/O should not pose as the bottleneck of the framework, using RAM would increase the chances of unwanted side effects. Filling up RAM can have an impact upon other applications running locally, such as a data capturing process running concurrently with FUR.

The framework design provides three ways of continuously uploading data to a remote storage solution. All three uploading methods support both binary and typed data.

1. Uploading data files separately as-is
2. Uploading data batches separately packaged as HDF5 files

3. Uploading an initial empty HDF5 file and perform appending uploads of data batches to this file

FUR will seek to upload data batches whenever they are made available. It will be possible to specify how many data entries a batch should consist of, such that users of the framework can choose the best batch size for their data. Once the batch size has been determined it remains static throughout the entire run. The framework will also allow the possibility of choosing whether to remove local data files after their data has been uploaded to the remote storage solution.

The work performed by every upload method in the framework is designed to be split into two functions, a runner function and an upload function, that share a buffer. The runner functions do the main lifting in the framework and performs a wide range of tasks. The main purpose of a runner function is to have data buffered, such that the the upload function can concentrate on reading buffered data and uploading it. Upload functions handle retrieving packaged data from the buffer, uploading the data and cleaning up intermediate files used for storing packaged data. The reasoning behind splitting up the work into runner and upload functions is that the work in both functions are I/O bound, as they do not perform any CPU heavy operations. This design will allow the two functions to run in separate threads in order to run concurrently. This will hopefully result in the two functions being able to utilize the waiting times of I/O operations of one another. The choice of using a shared buffer between two threads also introduces potential problems, such as race conditions [21]. In order to account for race conditions and synchronization issues a shared lock for accessing the buffer will be used in the both type of functions.

The framework is designed to create an initial connection via SFTP or FTPS and create the remote directories of the specified storage location if needed. It will keep this connection running throughout the lifespan of the runner function in order to avoid the cost associated with creating a connection multiple times. The framework is also designed to allow for packaging of files into batches before being uploaded. This design choice can aid in reducing the impact that RTT and TCP slow start can have on uploading speed, as multiple uploads are condensed into a single continuous upload.

The main differences between the two supported file transfer protocols are their

methods for encryption, authentication and the number of connections used. FTPS runs asynchronously over two channels, a control and data channel, whereas SFTP runs everything over a single channel. This gives FTPS a slight edge speedwise, but not enough to make the protocol choice massively impactful on the data transfer rate, when put up against the bandwidth of the connection. Both protocols are used in FUR to increase the versatility of the framework.

Whenever the transfer of data is the bottleneck of an application, it can prove useful to spend resources compressing the data. Compression is CPU bound, and depending on compression method and and data sizes, it can be taxing for the system performing the encoding. The FUR framework design includes the use of encoding through the HDF5 GZIP filter. When using this filter with remote file objects for appending data to a HDF5 dataset, it will use the resources of the client system to carry out the encoding. This makes for another compelling argument as to why runner and upload functions in FUR should run in two separate threads, as the CPU bound encoding in the upload function hopefully can make use of the CPU in the time that the runner function spends waiting for I/O operations to finish.

4 Implementation

This section will cover the implementation of the FUR framework with a focus on the implementation and optimization of the appending upload method for typed and binary data. The end of the section will introduce two use-case examples. The code for the implementation can be found and downloaded at:
`https://github.com/DraeberNiels/FUR`.

4.1 Language

The framework is implemented in Python 3 and is therefore available as a python module named `fur`. Python 3 was the language chosen for implementation purposes due to several reasons, such as its ease of use and strong support by libraries for data processing and analysis.

4.2 Creating HDF5 files and datasets

The design of the framework requires the creation of local HDF5 files, and these files are created with the `h5py` [7] library. Specifications of the dataset in the created HDF5 files are gathered from input parameters, which are able to specify datatype, chunk shape, the use of a compression filter and size of the dataset. Created datasets

that are used with the appending upload method are defined to have an empty shape and unlimited max shape upon creation. This will allow for the dataset to be resizable and make the HDF5 file as small as possible for the initial file transfer.
The framework supports storing data in HDF5 datasets as integers (8-64 bit), floats (16-64 bit) and as binary data. The binary datatype allows for storage of data that comes in heterogeneous shapes and binary file formats, such as the JPEG file format.

4.3 Creating a connection

Before processing data or starting an additional thread for running an uploading function, the `_runner_*` functions start by establishing a connection to the specified server by using the specified file transfer protocol. The Python libraries `ftplib` [22] and `pysftp` [23] are used for creating FTPS and SFTP connections respectively. After establishing the connection, the directories of the path defined in the input parameters are created recursively if they do not exist already. After this step is completed, `_runner_*` functions spawn the corresponding upload function in a new thread, passing the connection and other needed parameters. Connections are closed when the last batch of data transfer is completed by the given upload function.

4.4 Gathering and buffering files

Gathering and buffering files is the responsibility of the three `_runner_*` functions in the `fur` module, where each functions corresponds to one of the uploading methods. The `_runner_simple` handles the upload method of uploading individual data files as-is, `_runner_hdf5` handles the upload, packaging and cleanup of separate batch sized HDF5 files and the `_runner_hdf5_append` handles appending upload of data batches to a single remote HDF5 file. In the cases of the uploading methods that include HDF5 formatting, the corresponding `runner` functions perform a brief analysis on the first available data file. The analysis extracts measures such as the shape of the data and what type that makes sense to use for storing the data in the HDF5 format. After the initial analysis, the `runner` function starts waiting for the first batch of data to complete. It then processes and buffers the data of the current batch, making it available to the spawned upload function. This step is then repeated until all data batches have been processed and buffered. Then it waits for the upload thread to join back before returning.

Buffers used throughout the framework contain file names. The `_runner_simple` and `_runner_hdf5`, use their buffer to store files that are ready for upload via SFTP

or FTPS. This is also the reason why the two functions both spawn threads running identical upload functions. The `_runner_hdf5_append` function uses its buffer to store filenames of packaged `npy` files for batches of data. As this data needs to be further processed before the appending upload, the function spawns a different upload function in a new thread, which is designed to handle the buffered data and perform the appending upload. All synchronization issues with regards to shared buffers are handled by the `lock` implementation of the Threading module [24]. Both functions wait to acquire the shared lock before accessing data in the buffer and eventually release the lock when the access is done.

4.5 Implementing HDF5 appending upload

The implementation of the HDF5 appending upload method is based on the use of the `h5py`, `pysftp` [23] and `numpy` [25] libraries. The **h5py** library is a python interface to the HDF5 file format and the `pysftp` library allows for creating SFTP connections. The `pysftp` library uses the `Paramiko` [26] library, which is a python implementation of SSH(version 2).

FUR uses a SFTP connection created with `pysftp` to access a file object of a remotely stored HDF5 file. This remote HDF5 file object is then opened and made available for HDF5 file operations with `h5py`. In order to open a HDF5 file object with `pysftp`, whether be it local or remote, the file object must support binary I/O and the file operations `read`, `write`, `seek`, `tell`, `truncate` and `flush`. Remote file objects opened with `ftplib` or other Python 3 FTPS libraries did not support all necessary file operations, hence the only choice found for implementing the HDF5 appending upload method was with SFTP.

When a remote HDF5 file is opened in this fashion, all normal available functionality of the **h5py** library is made available to read and modify the HDF5 file. This allows for transfer of data by appending locally stored data to a dataset of a remote HDF5 file object, hence the name: appending upload. As mentioned in section 2, the array part of datasets are homogeneously typed arrays. This along with `h5py` supporting python array slice notation for HDF5 datasets, makes it very compatible with `numpy` arrays, as they share identical properties. This is also the reason why data batches in FUR for the HDF5 appending upload method are packaged and buffered into `npy` files, such that the data can be read with `numpy` and directly appended to the remote

HDF5 dataset by using Python slice notation.

The main problem of the straightforward implementation of the HDF5 appending upload method is that it is very slow, approximately transferring data at 1/10 of the rate when compared to file uploads using the SFTP `put` method, as shown by the result of Table 1. This shows that even though the implementation of the method is functional, it is very inefficient compared to the SFTP `put` method.

Protocol	Mean Upload time	Standard deviation	Mb/s
SFTP	14.90 s	1.02	53.69
SFTP Append Unoptimized	152.85 s	5.23	3.54

Table 1: Mean upload time in seconds, its standard deviation and the upload speed. Test consisted of 10 upload tests to ERDA with 100 MB dataset of typed image data with SFTP `put` method and unoptimized HDF5 appending upload. The amount of uploads in the test was chosen to reduce the significance of a bad runs dictating the results. The size of the dataset was chosen as it seemed fitting for giving an indication of upload speed, while stile being small enough for a quick test. Tests were performed on the Laptop, which is described in the section 5

4.6 Optimizing append upload remote HDF5

If the presented implementation of the HDF5 appending upload method seeks to be efficient and competitive, it must be able to match the speed of other uploading methods. The first step in investigating bottlenecks of the method was to perform some tests locally. The tests consisted of measuring the speed of appending data to a HDF5 dataset and comparing it to the speed of a sequential write operation to a plain text file with the same amount of data. These tests showed that there was no significant overhead introduced in appending data to a HDF5 dataset that would explain the significant slowdown experienced using remote file objects. The tests did show that storing data in binary format introduced a small amount of extra overhead, but again nothing that could explain the significant slowdown. These tests gave the indication that the slowdown of the appending upload method was not due to the **h5py** library.

The next step in the process was to perform the same tests, but now using remote file objects for the two file types. The results showed that performing sequential writes with the same amount of data to a remote text file was just as slow as the appending upload method. These results indicated that the slowdown could be a product of the `write` function of the SFTP file object. The documentation of the SFTP paramiko implementation, states that standard file uploads methods such as the `put` function uses pipelining for improved speed. It also states that it is possible to enable pipeling for `write` operations done to a SFTP file object. Enabling pipelining does introduce the risk, that the first non-`write` operation, such as close, can result in a `write` error being thrown from that operation instead of being thrown from the `write` operation. Pipeling was enabled for both the remote file objects and tests repeated. This resulted in the speed of sequentially writing to a text file and HDF5 appending upload with typed data matching the upload speed of the SFTP `put` method. However, using pipelining with the appending upload method for binary data resulted in timeout errors.

As the speed of `write` operations had been pipelined, the next step was to dissect the steps of appending to HDF5 datasets, in order to figure out what caused the erroneous appending of binary data. All datasets used with the framework are resizable datasets, which automatically enables chunking. The problem with binary data and chunked storage is that chunks for storing binary data cannot be allocated, as the sizes of each data entry is unpredictable. Therefore, in comparison to homogeneously typed and sized data, the storage method does not know how much storage will be used in total per chunk. This is solved by `h5py` by storing the binary data to disk, and storing two pointers to the start and end addresses of the given binary data entry in an chunk index. These additional storage measures are most likely one of the causes of the extra overhead of the HDF5 appending upload method when using binary data.

The appending upload with h5py is done in two main steps. The first step is to resize the remote HDF5 dataset and the second is to append local data to the newly resized remote HDF5 dataset through standard Python slice notation. The actual `h5py resize` command does not change the byte size of an HDF5 dataset storing binary data, but only the metadata in the HDF5 dataset that is in charge of the dataset size. This means that storage is not allocated upon resizing, which is most

likely in a effort to not have to allocate storage that is not yet in use. Whenever data is appended to the resized HDF5 dataset the storage is then allocated. The errors and timeouts occurred whenever appending was done across multiple chunks or whenever a new chunk in a dataset was being appended to. The work in this thesis did not fully uncover why the errors and timeouts with binary data occurred, so further work is needed to investigate the issue. However, experimentation with HDF5 appending uploads and different functionality during the thesis did uncover how to fix the issue. The issue with the appending upload method with binary data was fixed by pipelining the upload of an entire batch of data except for the last data entry of the batch. Appending the last data entry was done separately with pipelining turned off. Furthermore, the file `flush` operation needs to be performed after the last append, forcing all buffered writes out of the file object write buffer. The solution to the problem indicates that the errors and timeouts were the result of mixing pipeling of the `write` operations of SFTP file objects and the `h5py`/HDF5 method of allocating chunked storage space for datasets, but more investigation of the issue is needed.

4.7 Use-case examples

This subsection gives two examples of how FUR can be used and what the framework can provide to systems connected to digital measurement tools.

4.7.1 CAB microscope

Center for Advanced Bio imaging Copenhagen have several advanced microscopes, which are connected to cameras and local computer systems. Data capturing is managed through capturing software on the local system, and all data is stored locally. The FUR framework can be used to run concurrently to the image capturing process and help automate the upload of the image data to a remote storage solution, thereby centralizing access to the data. The number of images captured per session tends to be within the two-digit range and adjustments to the given microscope is usually performed between the capture of two images. Therefore, it is expected that the FUR uploading method of uploading data files separately as-is will be of use for this type of system. Furthermore, FUR can be helpful in the cleanup of already uploaded local image data, such that the memory of the local systems connected to the microscopes do not have to be manually cleaned.

4.7.2 Industrial camera connected to an embedded system

Another use case could be using FUR on an Industrial camera connected to an embedded system. Embedded system solutions can allow for more specific control of formatting and capturing of image data, but usually have limited memory as they are not meant to be storage units. If image data is captured at a set framerate throughout long periods of time, the framework could run in a separate process to help with continuously uploading batches of the image data and deleting local copies upon completed uploads. This type of scenario is something that the HDF5 appending upload method of FUR is suited for. Furthermore, if the upload speed of FUR is able to match the data production rate, and cleanup of locally stored data is enabled, it means that local storage capabilities no longer become the limiting factor of how much image data can be captured and stored in one session. Use of the FUR framework can reduce the risk of the connected system running out of disk storage and it can make captured image data accessible through the used remote storage solution.

5 Systems

This section briefly introduces the systems used for production of data and testing. It will give a more detailed description of the Embedded Linux system connected to the Industrial Camera and ERDA, and show the specifications of the remaining systems in a table.

5.1 Industrial Camera & Embedded Linux System

This subsection is taken from the preproject [4].
The camera used for producing test data for the results section in this thesis was provided by Qtec. It is a modular industrial camera that runs an Embedded Linux distribution named Poky Qtec 2.2, based on the Yocto Project 2.7 Reference Distro[27]. The camera device supports capturing pixel resolutions up to a height of 1792 and width of 1264 with up to 30 frames er second (FPS) and it has two read buffers. For storage capabilities, it has 3.4 GB RAM and 15 GB disk storage. All interaction between the camera device and Linux system is handled through the Video4Linux2[28] (v4l2) framework. The framework supports real-time video capture on Linux systems and is supported by the Linux kernel. The Embedded Linux system has an AMD G-T56N processor with 1,65 GHz with a total of 2 cores.

Device / Measure	Operating system	Processor	RAM	Bandwith (down/up)
Embedded Linux System	Poky Qtec	AMD G-T56N 1.65 GHz, 2 cores.	2 GB	300/60 Mb/s
Laptop	macOS Mojave	Intel Core i5 2.4 GHz, 2 cores	4 GB	300/60 Mb/s
CAB computer	Windows 10	Intel Core i5-7500 3.40 GHz, 4 cores.	8 GB	1000/1000 Mb/s

Table 2: Specifications of the systems used for testing and producing results in this paper

5.2 Specifications of used systems

Table 2 shows the specifications of the three systems that are used for testing and producing results. The Embedded Linux System and Laptop used the same bandwidth connection, just wired and wireless respectively. Using a wireless connection for testing is suboptimal, as it can introduce a lot of variability in the connection speed. However, as a sufficient cable was not available, it was a choice of testing with a wireless connection or not testing with the Laptop system. Speed tests performed with the wired connection with `speedtest-cli` [29] showed an upload speed of just below 69 Mb/s. The same tests performed with the wireless connection showed an upload speed of just below 68 Mb/s. The CAB computer used a wired fiber connection and speed tests showed an upload speed of around 960 Mb/s.

5.3 ERDA

Electronic Research Data Archive (ERDA) [1] is a storage and archiving server provided by the University of Copenhagen. Stored files on ERDA can be accessed through a browser interface, SFTP, FTPS, internal clusters and resources within the ERDA setup. ERDA has a fiber connection and therefore should theoretically support upload speeds of up to 1000 Mb/s. However, this connection speed might not always be accessible, as multiple users can access ERDA at any given time, which can have a big impact on connection speed. Therefore, the connection speed to ERDA can vary depending on when tests are performed, since there is no way of telling how many users that are currently connecting to ERDA. There are no hard limits on storage space per user on ERDA, but it is suggested that users do not use more than a couple of terabytes. It will serve as the remote storage solution used

for testing the FUR in this thesis, as it will likely not cause a bottleneck due to its fiber connection, it supports SFTP and it provides plenty of storage space. More information about features and setup of ERDA can be found in the ERDA user guide [30].

6 Results and Benchmarks

This section will present the results and benchmarks of using the FUR framework functionality on different data sizes and types. The tests will be performed with the three systems and ERDA that are all described in the systems section. As the HDF5 appending upload method is the most interesting and complicated uploading method in FUR, this section will focus on showing results of its implementation. At the end of the section a bottleneck analysis of the implementation will be performed based on the showcased results.

6.1 Data used for benchmarks

In order to fairly compare the performance of the framework functionality across systems, identical datasets were used across systems. The used datasets consisted of image data produced by the industrial camera described in the systems section. All datasets consisted of 1800 image frames in RGB (Red Green Blue) [31] format or JPEG (Joint Photographic Experts Group) [32] format. These two data types allow for testing both the typed and binary HDF5 appending upload implementations. The sizes of the HDF5 test files and frames used in the tests do not serve a specific purpose, besides being of a size such that RTT and other factors did not become dominant in the test results.
It was chosen to use prerecorded data instead of generating data, in order to minimize variability across systems by giving the same starting point for all tests performed.

6.2 Different measures of the framework functionality

In order to compare the speed of uploading files with SFTP and FTPS with the upload speed of the HDF5 appending upload method, it is important to explain how the upload speed of the FUR implementation was measured. The upload speed of the HDF5 appending upload functionality includes the time spent uploading the initial empty HDF5 file and the time spent both resizing and appending to the remote HDF5 file.

Besides the actual upload speed of the HDF5 appending upload implementation

of FUR, the introduced overhead of packaging and managing buffered data in the given upload function was also measured, as to not neglect the additional work that is being done to support perform the uploading method. In order to test that the integrity of the test image data was intact after uploads had finished, random frames of the remote HDF5 dataset were displayed and value checked programmatically.

6.3 Test setup

All test results presented throughout this section consisted of timing 10 consecutive runs of uploads with the specified data and system. Before the 10 consecutive uploads were measured, a single warmup run of the given test was performed to ensure that it worked. From the timing of the 10 uploads, metrics such as the mean upload time in seconds, the corresponding standard deviation, and upload speed in Mb/s were calculated. The two measuring units that where used throughout testing were time in seconds and the amount of bytes stored in the final version of the uploaded files. As bandwidth connections generally are measured in Mb/s (Megabit per second), the following equation was used to calculate Mb/s from the test measuring units : (bytes transferred) * 8 / (seconds used uploading) * 10^{-6}.

The tests results from the Embedded Linux camera and Laptop were all performed at midnight, in order to reduce variability in connection speed. The tests performed on the CAB system where done around 12 pm. due to limited accessibility. All uploading tests throughout this section use ERDA as the remote storage solution.

6.4 Baseline tests

In order to establish a baseline for comparison of upload speeds of the various methods across the three test systems, tests of directly uploading a 1016 MB file to ERDA with SFTP and FTPS were performed on all systems. The 1016 MB file size was chosen for this test, as its size should be big enough to give an appropriate picture of upload speeds when uploaded with the test systems and their bandwidth connections. The results of the baseline tests can be seen in table 3.

Even though the contents of Figure 3 are baseline results, they still provide important information and provide grounds for comparisons of results later on in this section. It seems that with the 1016 MB file size, uploads via FTPS on the Laptop and Embedded Linux system were able to reach upload speeds very close to the results of their bandwidth speedtests from section 5.3. As expected, the results

Device / Measure	Protocol	Mean Upload time	Standard deviation	Mb/s
Embedded Linux System	SFTP	253.08 s	1.90	32.12
Laptop	SFTP	131.88 s	3.59	61.64
CAB computer	SFTP	27.88 s	1.51	291.58
Embedded Linux System	FTPS	120.32 s	0.34	67.56
Laptop	FTPS	124.10 s	3.63	65.51
CAB computer	FTPS	*	*	*

Table 3: Baseline test for upload speeds of SFTP and FTPS for uploading a single 1016 MB file to ERDA. The results of uploading the 1016 MB file with FTPS on the CAB computer are missing, due to a mistake in the test script of not closing the FTPS connection after upload of the file had completed.

produced with wired connections show more consistency, which the lower values of the respective standard deviations indicate. Protocol-wise, the speed of file uploads via FTPS is faster than the speed of file uploads via SFTP by a significant amount. Furthermore, the results indicate that SFTP on the Embedded Linux is only able to upload files at approximately 50 percent of the speed when compared to the upload speed of using FTPS on the same system. The same does not seem to be a problem with using SFTP on the Laptop, which uses the exact same bandwidth connection just wireless, as the baseline results produced with the Laptop show only a minor slowdown between the file uploading speeds of FTPS and SFTP. As bandwidth does not seem to be causing the problem, this is likely caused by the SSH installation and could be e.g. an encryption issue. This issue was not further investigated, as it is not the responsibility of FUR to manage SFTP installations, but rather to make use of the provided installations. Even though this has a significant influence on upload speeds of the Embedded Linux Systems throughout this test section, as the appending upload method uses SFTP, it can show how reliant the performance of FUR can be on the provided SSH installation.

The baseline tests performed on the CAB computer show that even though the computer has a bandwidth connection of 1000 Mb/s upload, it is not able to be saturated through file transfers via SFTP or FTPS. This is likely not caused by the choice of protocols, as upload speed is reliant on many factors, such as limitations of the network used on the way to ERDA and ERDA's congestion policy. However,

this is again not a major problem, as this thesis does not look to optimize or create new protocol implementations, but rather make the best use of existing protocols to enable and effectivise HDF5 appending uploads.

6.5 HDF5 append upload tests

6.5.1 Tests with typed data

The results of testing the HDF5 appending upload method using a dataset of 1016 MB typed data consisting of 1800 frames can be seen in figure 4. The reasons for using this size of dataset, besides the reasons mentioned for using the size in the baseline tests, is to give the a basis for comparison with the baseline tests. The chunk sizes used with all typed data tests were approximately 0.3 MB, as small preliminary testing showed that this optimized the transfer rate to ERDA. All tests in this subsection also use a batch size of 400 frames, to also test if the FUR implementation works with batch sizes that are not a multiple of the total dataset size.

Device / Measure	Protocol	Mean Upload time	Standard deviation	Mb/s
Embedded Linux System	SFTP Append	262.48 s	6.62	30.96
Laptop	SFTP Append	131.65 s	1.68	61.75
CAB computer	SFTP Append	25.90 s	0.45	313.84

Table 4: Tests of HDF5 appending upload via SFTP to ERDA with 1.016 GB dataset of image data consisting of 1800 frames in RGB format. Batch size was 400 frames.

Results produced with the CAB computer and Laptop in Table 4 show an increase in upload speed compared to the baseline tests. The consistency of the results on these two systems is also great, as indicated by the small standard deviations. Taking a look at the mean upload time, the difference between the test results on the Laptop and its baseline tests is minimal, indicating a similar performance of the two uploading methods. The mean upload time of the results from the CAB computer shows an improvement in upload speed of around 7 percent, which is also very consistent according to the standard deviation. This is likely caused by fluctuations in the connection speed, as the throughput of the tests are still far from the theoretical limit. The tests from the Embedded Linux system show that the upload speed of

the appending upload is slower when compared to the baseline tests. Looking at the mean upload time, these tests show a significant slowdown of around 8 seconds, which is a decrease of around 3 percent. The standard deviation of these tests is also percentually the biggest, even though the Embedded Linux system used a wired bandwidth connection, which was close to being saturated via uploads with FTPS in the baseline tests. Therefore, the explanation of this variability could again be explained by limitations of the SSH installation.

The differences of speed between the HDF5 appending upload method and baseline results experienced across the three systems are somewhat expected and likely explained by variability in the connections. Besides variability in the connection speed, this difference could also be explained by differences in the uploading methods, such as the `put` function of SFTP checking the file size of an uploaded file after upload completion. Furthermore, the FUR implementation of the appending upload method loads a batch of image data into memory before performing the upload, where the work of doing this is included in the results of using the `put` function of SFTP. Results of using the appending upload method that include the time of loading the image data into memory and deleting the corresponding `npy` file can be seen in table 5.

Device / Measure	Protocol	Mean Upload time	Standard deviation	Mb/s
Embedded Linux System	SFTP Append	266.94 s	7.67	30.45
Laptop	SFTP Append	133.82 s	1.72	60.74
CAB computer	SFTP Append	26.51 s	0.46	306.62

Table 5: Tests of HDF5 appending upload via SFTP to ERDA with 1.016 GB dataset of image data consisting of 1800 frames in RGB format. Batch size was 400 frames. This Table shows results including time for loading data into memory and deleting temporary `npy` files

The results of table 5 show that when the time for loading data into memory and deleting the buffered `npy` file are factored in, the decrease in upload speed is between 3.1 percent, 1.6 percent and 2.3 percent for the results produced on the Embedded

Linux system, Laptop and CAB computer respectively. This makes the tests of the upload speed on both the Embedded Linux system and Laptop slower than the baseline tests, while the results of the CAB computer tests are still 5 percent faster than the baseline tests.

The results of tables 4 and 5 indicate that the HDF5 appending upload method with typed data performs just as well as standard file uploads with the SFTP `put` method when it comes to upload speed. They also indicate that the appending upload method is able to scale well when used with systems with different bandwidth connection speeds, as the HDF5 appending upload implementation shows a similar relation to the speed of the SFTP `put` method across all test systems.

6.5.2 Test with increasing sizes of typed data

In Figure 6 the results of performing tests of the HDF5 appending upload method with datasets of size of 254 MB, 508 MB and 1016 MB are shown. These sizes of the datasets were based on the 1016 MB dataset used in the prior tests. This was done to investigate how the upload speed of the HDF5 appending method behaves with scaling data sizes. Batch sizes were kept as 400 frames for the upload tests of all datasets, which results in different byte sizes of data batches. The upload speed results include the time for loading data into memory and removing the corresponding `npy` files. The tests were performed on the CAB computer as it was deemed the most resourceful system and had stability with using a wired connection.

The boxplot in Figure 6 shows that the tests with 508 MB dataset were fluctuating just over 100 Mb/s between the minimun and maximum and around 60 Mb/s in the interquartile range. These are some big fluctuations that are likely generated by variability in the connection, which could be caused by several issues as discussed earlier. The tests of the other datasizes were much more consistent, as their more condensed boxplots show. A potential trend that shows is that the increase in dataset size and hence increase in byte size of batches, results in an increase in the Mean upload time. However, more datapoints with greater consistency are needed to determine if this is a trend or merely caused by something such as bandwidth variability.

With that said, it is expected that an increase in byte size of batches will improve the upload speed, as the overhead of handling batches will be less significant and

Boxplot of HDF5 appending upload speed with scaling dataset sizes

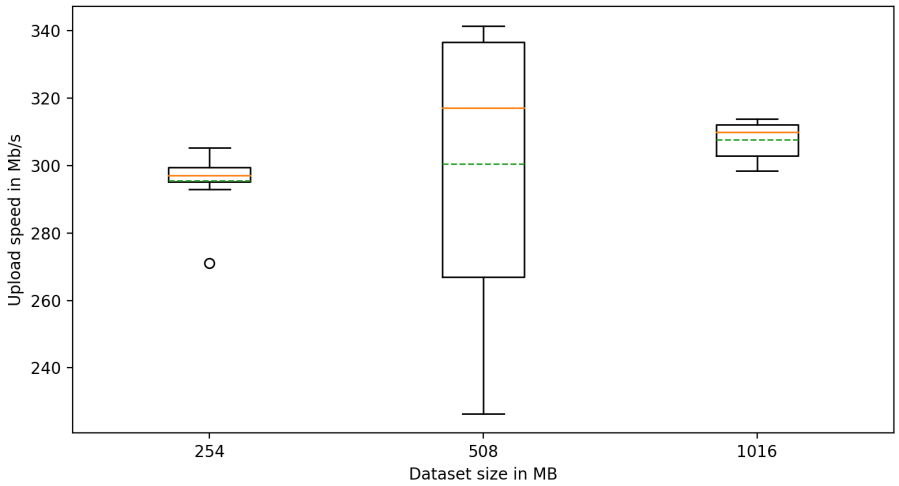

Figure 6: Boxplot of HDF5 appending upload via SFTP to ERDA with datasets of 254 MB, 508 MB and 1.016 GB of image data in RGB format from CAB computer. Batch size for all tests was 400 frames. The orange lines depict the medians of the test results and the dotted green lines the means.

a larger amount of data is uploaded continuously. Even though the tests with the 254 MB and 508 MB datasets transported one fourth and half of the bytes of the 1016 MB dataset respectively, results with all three datasizes show very similar mean upload times. This lightly indicates that the implementation of the HDF5 appending upload method when used with batches of smaller byte size, is able to perform just as well as the SFTP `put` method when it comes to upload speed. In light of this, it is expected that if byte sizes of batches get too small it can have a major negative impact on upload speeds. It would be interesting to investigate how the upload speed behaves for batches with even smaller byte sizes. This calls for further testing of the FUR implementation to be done.

6.5.3 Tests with binary data

The results of tests using binary data with the HDF5 appending upload method can be seen in figure 6. These tests were conducted using a dataset of 333.07 MB image data consisting of 1800 frames in JPEG format. This dataset size was deemed fitting to get a decent measure of upload speeds. Results from Figure 6 with a 254 MB

dataset indicated that upload speeds of tests using one fourth the size of dataset in the baseline tests, were able to achieve similar upload speeds to the baseline tests. Hence, using a dataset that is one third of the size should not prove too problematic, but using a binary dataset with a size of 1016 MB would have been optimal. The chunk size for these tests was set to be equal to the batch size and the timing of the results included time for loading data into memory and deleting temporary `npy` files.

Device / Measure	Protocol	Mean Upload time	Standard deviation	Mb/s
Embedded Linux System	SFTP Append-Bin	115.02 s	0.35	24.28
Laptop	SFTP Append-Bin	60.12 s	1.44	46.46
CAB computer	SFTP Append-Bin	12.34 s	0.31	226.35

Table 6: Tests of HDF5 appending upload via SFTP to ERDA with dataset of 333.07 MB image data consisting of 1800 frames in JPEG format. This Table shows results including time for loading data into memory and deleting temporary `npy` files

The results of figure 6 show that performing appending uploads with binary data to the HDF5 file format is slower than both the baseline and typed data results. The decrease in upload speed is likely caused by the introduced overhead of two factors. The first factor is that storing binary data in chunks in the HDF5 file format is more complicated due to the uncertainties of the heterogeneous data sizes, as explained in earlier sections. The second factor is the additional work in turning of pipelining for appending the last element of every batch and executing a file `flush` afterwards. It is noteworthy that upload speeds are all consistently 22-25 percent slower across all systems when compared to the baseline results. This indicates that the HDF5 appending upload method does introduce a slowdown compared to the `put` method, but does so consistently at around 25 percent.

6.6 Test of other framework functionality

The functionality of the two other upload methods of FUR were also tested with all systems to ensure that they worked. These tests were purely to test for functionality, as both uploading methods simply consist of uploading files via SFTP or FTPS, making them not too interesting. As for the upload method of uploading data files

separately as-is, it is expected it will provide the lowest upload speed of the three upload methods, as it has to perform the overhead associated with a file transfer operation for every single data entry. As the size of data files grow towards 1016 MB, it is anticipated that the upload speed of the method will begin to resemble the results from the baseline tests.

The upload method of packaging data batches in HDF5 files and uploading them separately is expected to perform upload speeds very similar to the upload speed of the HDF5 appending upload method, when both methods use SFTP, identical data and batch sizes. It it anticipated that it will be a bit slower, as extra overhead is introduced with transferring multiple files instead of just one. The advantage of this upload method is that it is able to use FTPS, so this could possibly cause this upload method to be faster than the HDF5 appending upload method.
Tests of using the GZIP compression filter with datasets for the HDF5 appending upload method was also performed with all systems, but these tests were again only performed to ensure that the feature worked. A reason for not going into detail about encoding performance, is that encoding speeds are very system dependant, as encoding is performed client side before data is transferred. Furthermore, the use GZIP compression filters in FUR is simply using the `h5py` functionality, so no actual implementation of compression filters has been made in this thesis. Therefore testing and analyzing the use of compression filters would have close to nothing to do with the FUR implementation but all to do with the `h5py` library.

6.7 Bottleneck analysis of HDF5 appending upload method

The results from this section indicate that the overhanging bottleneck of the FUR implementation of the HDF5 appending upload method was the speed of the bandwidth. This means that the implementation of the framework achieved its goal of not acting as a bottleneck in the upload of data. Uploading binary data with the HDF5 appending upload method did make the upload speed approximately 25 percent slower, but results showed that this was consistent across all test systems, which all used different bandwidth connections. This indicates that the speed of the bandwidth connection was still the main bottleneck, as the upload speed kept increasing with faster bandwith connections.

Generally, using SFTP for file transfers did introduce a minor slowdown in uploading speed when compared to uploads done via FTPS, but using SFTP did not act as a

significant bottleneck on the Laptop or CAB computer. However, the results of the tests performed with the Embedded Linux system showed that FUR can be reliant on SFTP installations. The upload speeds performed by the Embedded Linux system with SFTP only managed approximately 50 percent upload speed when compared to file uploads with FTPS on the same system. When this was combined with storing binary data in the HDF5 format, it acted as a major bottleneck, slowing down the upload speed approximately 65 percent compared to baseline tests performed with FTPS. Further testing is needed to show if the upload speed of the system with SFTP would improve with a faster bandwidth connection.

7 Discussion

This section critiques the test setup and discusses the advantages and limitations of the design and implementation of FUR.

7.1 Critique of test setup

The tests and results provided in section 6 do provide some decent indications of the performance of the implementation of the HDF5 appending upload method in FUR. However, there are things that could be improved in the test setup in order to better support the indications of the results.

The biggest point of critique is the low amount of repetitions in the test setup. As mentioned, time and access was limited with the CAB computer resulting in a low number of test repetitions, and tests on the other two systems were conducted in a consistent manner. Ideally, more repetitions of the tests performed on the two other test systems could have been done, in order to achieve a stronger foundation for concluding upon the produced results.

Another point where the test setup could improve is to test a much wider range of dataset and batch sizes. As mentioned in the the test section, it would be interesting to see how smaller datasets and batch sizes affect the upload speed and at what size point the upload speed becomes significantly slower. While it did seem reasonable using a 1016 MB file for testing upload speeds, as the baseline results were very close to the speed tests of the given bandwith connections, this file size is still considered very small in the world of data exchange. Therefore, it would also be interesting to perform tests with much bigger file sizes.

7.2 Advantages

As the results section showed, the implementation of HDF5 appending upload method in FUR proved to be able to match the upload speed of the SFTP `put` method. This is an advantage in itself, as the HDF5 appending upload method can run concurrently with data production to store produced data to a remote HDF5 file continuously. This can result in saving time and resources by making efficient use of the time during data capturing and by avoiding the need for formatting data to the HDF5 format on the remote storage solution. By not making use of post processing methods for merging uploaded data to the HDF5 format, such as done in the preproject [4], the need for installing any software to handle processing on the remote storage solution is removed. These advantages become amplified when used together with other features of the FUR framework, such as performing cleanup of local data once it has been uploaded.

If the FUR framework is run concurrently with data production, it is able to extend the possible amount of captured data that can be stored, as FUR can perform continuous uploads and cleanup of the locally produced data. Furthermore, if cleanup of local data is enabled and the upload speed of the chosen uploading method from FUR is faster than the rate of data production, the amount of data that can be stored will be limited by the disk storage available on the remote storage solution, instead of being limited by the storage capabilities of the local system. This can prove very usable when used with systems such as embedded systems connected to digital measurement tools, as the memory and disk storage usually are very limited resources of such systems.

It is also an important advantage that the test results indicate that the FUR implementation scales with faster bandwidth connections. This along with the FUR framework being able to work with remote storage solutions as long as they support SFTP, makes for a general and seemingly efficient framework.

7.3 Limitations

The implementation and design of the FUR framework does not claim to be perfect and therefore does come with its limitations that users should be made aware of. One such limitation is that running functionality of FUR on a system concurrently to the data capturing process can cause buffering, packaging and uploading of data to im-

pact the data capturing process, if the given system does not have enough resources available for running the framework efficiently in a separate thread or process. The processing impacts of packaging and buffering never presented any bottlenecks, so their impacts were not investigated. This is definitely a shortcoming and should be investigated in future work, as this processing can have a big impact on low power systems.

The current implementation of the HDF5 appending upload method will at some point during runtime store a copy of data entries locally when it is packaged into `npy` files. As of the current implementation, FUR packages a batch of data as soon as it is available and then deletes the packaged batch after uploading it has completed. This means that if cleanup of local data is not enabled to remove the initial data files, and if the rate of data production and packaging are much higher than the upload speed, this can cause FUR to put a lot of unnecessary stress on local storage capabilities. Possible solutions to this problem are addressed in section 8.

The FUR framework also has its limitations when it comes to incoming data and how it can stored in the HDF5 format. The HDF5 appending upload implementation only works for chunked and resizable HDF5 datasets. This means that if contiguous storage in a HDF5 dataset is wanted FUR does not provide a solution for this. As explained in section 3, the FUR framework is designed to work with sequentially named data, which might not fit all types of data capturing. A possible solution for solving this could be a feature that sorts files by their creation timestamp. However, such a feature could also introduce some problems if ordering of the data is important. An example being that if there is no guarantee that captured files from a digital measurement tool are timestamped sequentially on the connected local system, it could result in storing data out of order.

It is recommended that users of FUR accommodate themselves with chunking of HDF5 datasets, as chunks have an impact on file size and speed of I/O operations. The optimal chunk size for uses in data analysis might not be the same as the optimal chunk size for minimizing the size of the HDF5 file. Therefore, users should analyze and base the chunk sizes of datasets on whether file size or I/O speed for data analysis it the most import factor, before using FUR.

8 Future work

This section discusses potential improvements of FUR.

8.1 Extending functionality of appending upload

As of now, the FUR implementation of the HDF5 appending upload method only allows for creating a new HDF5 file locally with a single empty dataset, which then is stored remotely and then appended to. A feature that could be introduced is to allow for appending uploads to already existing remote HDF5 files, either by creating a new dataset or possibly extending an existing dataset. It is expected that the upload speed of this feature would match the upload speed of the current HDF5 appending upload implementation, as the main difference is would be the need and location of creating a new dataset. This feature is at the top of the list for future development of the FUR implementation.

8.2 Optimize packaging and buffering in `_runner` functions

The process of buffering and packaging data has potential for optimization. This has not been a major focus point in the current implementation of FUR, since local I/O operations generally are much faster than bandwidth upload speed. However, if bandwidth connections and file transfer protocols get to a point where local I/O can potentially become a bottleneck, or FUR is used with a low power system, it will be important to have this part of the FUR framework optimized.

An optimization idea could be that packaging of data into buffered files is started as soon as any given data entries from a batch are available, instead of waiting until the last file of a given batch becomes available. Gradually scaling the batch size based on data production and upload rates could also be introduced, instead of keeping batch sizes static.

Another idea to optimize the use of local storage capabilities could be to perform a continuous analysis of the upload speed versus the rate of data production. This analysis could help in guiding buffering and packaging, such that in situations where data cleanup is not enabled, the local system does not end up storing packaged data unnecessarily if its not needed immediately for upload. Alternatively, FUR could allow for the option of doing both packaging and uploading of the current data batch in a single function, but warn users that this will likely reduce the speed of uploading data wit FUR.

8.3 Investigation of HDF5 appending upload with binary data

As mentioned in section 4, a further investigation is needed to pinpoint what causes the need for temporarily turning off pipelining and executing a file `flush`, when using binary data with the HDF5 appending upload method. Solving this this issue could give an indication of whether transferring binary typed data can be just as efficient as using typed data with the FUR framework.

9 Conclusion

There is a need for efficient software to aid local systems connected to digital measurement tools in uploading data to remote storage solutions. In order to address this need this thesis presented the FUR framework.

The FUR framework is designed to help automate and effectivize packaging and uploading data in the HDF5 file format to a remote storage solution, while also providing functionality for cleanup of local data. With a suitable hardware and network setup, the FUR implementation allows for remotely stored HDF5 files, which have been created with the framework, to exceed local storage possibilities of systems connected to digital measurement tools.

A functioning version of FUR was implemented in Python 3. The details of the FUR implementation were presented, along with challenges in optimizing the HDF5 appending upload method. The FUR implementation and its functionality was tested across three separate systems with different operating systems, hardware specifications and network setups to help ensure that the framework works as a general solution. The results of the performed tests work well to give preliminary indications of the performance of the FUR implementation. However, more testing is needed with a wider range of data sizes and higher number of repetitions in order to better support the indications of the produced results.

The produced results indicate that the FUR implementation of the HDF5 appending upload method with typed data is able to match uploading speeds of standard file uploading methods. They also indicate that using binary data with the HDF5 appending upload method causes an slowdown approximate to 25 percent in upload speed. Results were produced with systems using different bandwidth connections, which indicated that the upload speed of the HDF5 appending upload method scales with faster bandwidth connections. Therefore, FUR seems to be a well functioning solution that can help local systems connected to digital measurement tools with storing data on remote storage solutions.

References

[1] ERDA. https://erda.dk/, 2021.

[2] The HDF Group. The HDF5® Library & File Format. 2021.

[3] Center for Advanced Bioimaging Denmark. https://cab.ku.dk/, 2021.

[4] Niels Voetmann. Cloud Storage Solution for Industrial Camera. 2020.

[5] h5s3 documentation. https://h5s3.github.io/h5s3/index.html, 2017.

[6] Amazon s3. https://aws.amazon.com/s3/, 2021.

[7] Andrew Colette and contributors. h5py. https://www.h5py.org/, 2021.

[8] Filezilla. https://filezilla-project.org/, 2021.

[9] WinSCP. https://winscp.net/eng/download.php, 2021.

[10] T. Ylonen. SSH - secure login connections over the internet. In *6th USENIX Security Symposium (USENIX Security 96)*, San Jose, CA, July 1996. USENIX Association.

[11] T. Ylonen, and S. Lehtinen. SSH File Transfer Protocol, draft-ietf-secsh-filexfer-02.txt. https://tools.ietf.org/html/draft-ietf-secsh-filexfer-02, 2001.

[12] P. Deutsch. RFC1952: GZIP File Format Specification Version 4.3, 1996.

[13] Message P Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.

[14] Leah Wasser. Hierarchical Data Formats - What is HDF5? https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5, 2020.

[15] The HDF Group. HDF5: API Specification Reference Manual. https://support.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html, 2017.

[16] The HDF Group. Chunking in HDF5. https://support.hdfgroup.org/HDF5/doc/H5.format.html#Btrees, 2021.

[17] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. 1970.

[18] The HDF Group. III.A. Disk Format: Level 1A - B-trees and B-tree Nodes. https://support.hdfgroup.org/HDF5/doc/H5.format.html#Btrees, 2021.

[19] M. Gien. A file transfer protocol (ftp). *Comput. Networks*, 2:312–319, 1978.

[20] Allman, M., Paxson, V., and E. Blanton. TCP Congestion Control. 2009.

[21] Robert Netzer and Barton Miller. What are Race Conditions? - Some Issues and Formalizations. ACM letters on programming languages and systems. 1992.

[22] ftplib — FTP protocol client. https://docs.python.org/3/library/ftplib.html, 2021.

[23] pysftp. https://pypi.org/project/pysftp/, 2009.

[24] "threading — Thread-based parallelism". https://docs.python.org/3/library/threading.html, 2021.

[25] Numpy. https://numpy.org/, 2021.

[26] Paramiko. http://www.paramiko.org, 2021.

[27] Source Repositories. Git.yoctoproject.org., 2021.

[28] Part I - Video for Linux API. https://linuxtv.org/downloads/v4l-dvb-apis-new/userspace-api/v4l/v4l2.html, 2021.

[29] Ookla. Speedtest® CLI. https://www.speedtest.net/da/apps/cli, 2021.

[30] UCPH ERDA team. ERDA user guide. https://erda.dk/public/ucph-erda-user-guide.pdf, 2021.

[31] SGI. RGB. https://fileinfo.com/extension/rgb, 2021.

[32] Joint Photographic Experts Group. JPEG. https://fileinfo.com/extension/jpeg, 2021.