A Named Data Networking Based Fast Open Storage System Plugin for XRootD

Cătălin Iordache[1,*], *Susmit* Shannigrahi[3,**], *Yuanhao* Wu[2], *Sichen* Song[4], *Faruk Volkan* Multu[2], *Justas* Balcas[1], *Raimondas* Širvinskas[1], *Sankalpa* Timilsina[3], *Davide* Pesavento[5], *Harvey* Newman[1], *Lixia* Zhang[4], and *Edmund* Yeh[2]

[1]California Institute of Technology, 1200 E California Blvd, Pasadena, CA 91125, United States
[2]Northeastern University, 360 Huntington Ave, Boston, MA 02115, United States
[3]Tennessee Technological University, 1 William L Jones Dr, Cookeville, TN 38505, United States
[4]The University of California, Los Angeles, CA 90095, United States
[5]Research Associate, National Institute of Standards and Technology, 100 Bureau Drive, Gaithersburg, MD 20899, United States

Abstract.
This work presents the design and implementation of an Open Storage System plugin for XRootD, utilizing Named Data Networking (NDN). This represents a significant step in integrating NDN, a prominent future Internet architecture, with the established data management systems within CMS. We show that this integration enables XRootD to access data in a location transparent manner, reducing the complexity of data management and retrieval. Our approach includes the creation of the NDNc software library, which bridges the existing NDN C++ library with the high-performance NDN-DPDK data-forwarding system. This paper outlines the design of the plugin and preliminary results of data transfer tests using both internal and external 100 Gbps testbed.

1 Introduction

The burgeoning complexity of data transfer and access calls for more agile and robust data distribution paradigms. Named Data Networking (NDN) has emerged as a promising architecture that shifts the focus from "where" the data is located to "what" the data is, essentially making the data itself addressable and routable [1]. Within the scope of the Software-Defined Networking Assisted NDN for Data Intensive Experiments (SANDIE) project, we have previously investigated the integration of NDN into data-intensive research applications, illuminating its potential benefits and existing limitations [2] [3].

One of the challenges identified during this exercise was the constrained throughput performance offered by the standard NDN Forwarding Daemon (NFD)[4]. To overcome this barrier, we sought to leverage the high-throughput capabilities of the NDN-DPDK forwarder developed by the National Institute of Standards and Technology (NIST)[5]. However, we also noticed a glaring lack of C++ libraries that can support this high-speed forwarder. To address this gap, we developed a lightweight C++ library purposefully built to bridge the

*e-mail: catalinn.iordache@gmail.com
**e-mail: sshannigrahi@tntech.edu

gap between the ndn-cxx library [6] and the high-speed NDN-DPDK forwarder. By bridging existing limitations and introducing new functionalities, the NDNc library [7] serves as a pivotal step toward realizing the full potential of NDN as a high-throughput, efficient data distribution architecture.

This paper outlines the substantial endeavor undertaken as part of the NDN for Data Intensive Science Experiments (N-DISE) [8] project, which unites researchers from prominent institutions including Caltech, Northeastern University, Tennessee Tech, UCLA, and NIST. Their collaborative mission focuses on the integration of a data distribution system based on NDN for the CMS experiment conducted at the Large Hadron Collider (LHC) [9] that culminated in the demonstration of high-throughput NDN-based transfer of CMS data over a wide area deployment. Using applications developed using NDNc and the NDN-DPDK forwarder, the demonstration achieved an average goodput of 83.2 Gbps and a maximum goodput of 96.9 Gbps over an wide-area deployment.

The paper is structured as follows. In Section 2, we provide a concise overview of NDN and the two fundamental entities central to any NDN data transfer process. Section 3 delves into the design of the NDNc library, elucidating its capabilities and how it seamlessly integrates existing NDN C++ libraries with the high-speed NDN-DPDK forwarder developed by NIST. This section also outlines the architecture of two dedicated applications designed to showcase the library's performance. Moving on to Section 4, we expound upon the architecture of the NDN-based Open Storage System XRootD plugin, elucidating its capabilities and the array of supported functionalities, providing insights into their workings. In Section 5, we unveil the preliminary results achieved by the NDNc applications when deployed over a wide area network. To conclude, we offer an insightful discussion of our future development and integration plans, followed by our overarching conclusions.

2 Named Data Networking

Named Data Networking [1] represents a significant departure from traditional Internet Protocol (IP) based network communication. Originating from the Information-Centric Networking (ICN) concept [10], NDN transitions from host-centric to data-centric networking. In contrast to conventional IP networks, which prioritize addressing information to determine where to fetch data, NDN prioritizes the content of data packets themselves, using their names as the primary means of retrieval. This fundamental shift is facilitated by two types of packets: Interest Packets and Data Packets. Interest Packets, initiated by consumers or clients, act as requests for specific data and carry the name of the desired data, functioning somewhat similarly to queries in traditional network systems. Data Packets, on the other hand, are the producers' or servers' responses to Interest Packets and contain both the requested data and its name. These packets follow the path of the original Interest Packet to reach the requesting consumer. A unique aspect of NDN is its inherent ability to cache data at the network layer. Intermediate routers within the network can store Data Packets, and when subsequent requests for the same data arise, these routers can directly serve the data without engaging the original data source. This feature inherently enhances data retrieval speeds, optimizes bandwidth consumption, and increases overall system resilience. By prioritizing the content of data packets, NDN provides a content-centric networking approach that holds immense promise for revolutionizing data storage and retrieval systems like XRootD.

3 The NDNc library

To deploy an end-to-end solution for data distribution using NDN as the underlying architecture, we implement both a consumer and a producer application, the two fundamental

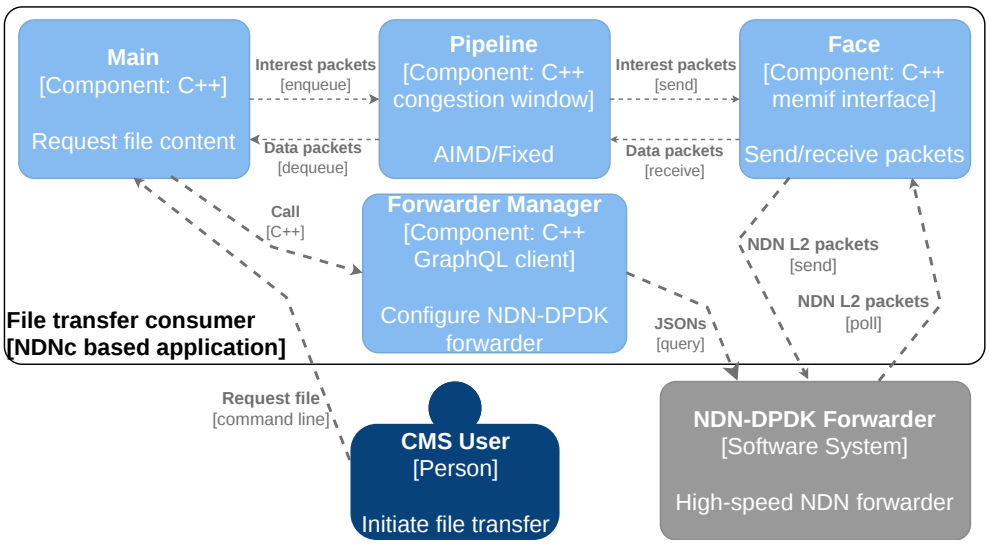

Figure 1: Component diagram for the NDNc file transfer consumer application

entities in any NDN data transfer process. Our initial efforts, conducted within the SANDIE project, highlighted the feasibility of NDN integration but unveiled limitations, notably constrained throughput attributed to the existing NFD forwarder [3]. To achieve exceptional packet forwarding performance, we design our applications to communicate with the high-speed NDN-DPDK forwarder developed by NIST, which has demonstrated leading throughput performance using multi-threaded forwarding [5] [11].

One challenge with this approach is the lack of any NDN C++ library compatible with this new forwarder. Therefore we developed our own lightweight C++ library called NDNc [7] that bridges the ndn-cxx library [6] with the NDN-DPDK forwarder, while adding a small number of common features needed by the consumer and producer applications designed for transferring files or byte ranges from files. The NDNc library offers PIT token [12] support (required for interoperability with NDN-DPDK), a *memif*-based face [13] capable of providing efficient packet transmit and receive functions to and from a locally running forwarder, an NDN packet encoder and decoder, Interest pipelining controlled by either a fixed-window or a congestion-aware AIMD algorithm, and a GraphQL [14] client that can configure the local forwarder by creating and deleting faces and registering Name prefixes [15].

Using NDNc, we have developed a file transfer consumer and producer for benchmarking the performance characteristics of the library. The consumer and producer are deployed using Docker containers [16]. Both entities follow a predefined naming scheme, and use the same name prefix. There are two types of Data Packets that can be requested: one for retrieving file information (or metadata) and the other for retrieving the file contents. Upon receiving an Interest Packet, the producer parses its name in order to extract the file path and the type of data it requests. For file information, the producer calls the POSIX `stat` system call on the file path and then embeds the answer in a common `Metadata` [17] object in NDNc for better encoding and decoding. In the case of Interests requesting content from the file, the producer calls the POSIX `read` system call to obtain the desired range of bytes from the file, as specified by the segment number in the Interest name.

The producer application runs in the network indefinitely, constantly waiting for new requests, while the consumer initiates a file transfer and then terminates. On initialization, the consumer uses the GraphQL client to configure a new interface with the local forwarder and then requests metadata information about a file passed as an input argument via the command line. If the requested file is found, the consumer constructs two worker groups, one for sending Interests and one for receiving Data Packets and assembling the result. The Interest Packets are passed to the *pipeline* that manages the congestion window and then to the local *face* that encodes the Interest to NDN L2 packets and finally sends the packets to the forwarder through the *memif* interface. The data is decoded from L2 and then passed to the receiving worker group. The pipeline takes care of NACK packets and timeouts. Once the file transfer is complete, the consumer destroys its face with the forwarder. The components of the file transfer consumer application are shown in Figure 1.

4 The XRootD NDN based Open Storage System plugin

To ensure a seamless transition for end-users from the existing CMS architecture to an NDN-based system, we have integrated NDN via an Open Storage System (OSS) [18] plugin tailored for the XRootD framework [19]. The OSS plugin facilitates the specialized implementation of a logical file system, wherein the operations of the logical file system are adeptly translated into actions tailored for the inherent storage system. Constructed using a C++ programming interface recommended by XRootD developers, each function in this plugin reflects the calls of the `POSIX` file system. The plugin manifests as a shared library, exporting the `XrdOssGetStorageSystem` symbol which the framework leverages for its loading.

The file transfer application discussed earlier, and the OSS plugin both operate on the same foundational logic in managing and translating `POSIX` file system calls. This logic takes the shape of an NDN consumer encapsulated in the *lib/posix* module of the library. However, their purposes diverge: the file transfer application is crafted to exemplify the optimal performance capabilities of applications built with NDNc, whereas the XRootD plugin is conceived to realize a file system as envisioned by the framework's developers. Both entities nonetheless employ the same producer application which persistently operates, responding to requests within the NDN network. For effective communication, every entity must utilize a consistent NDN Name prefix. In NDNc, the default prefix value is set to "$/\mathrm{ndnc}/\mathrm{xrootd}$". However, the plugin, file transfer consumer application, and the producer can be reconfigured through command-line inputs or configuration files to adopt an alternate prefix.

The NDNc `POSIX` consumer extends its support to integral file system calls for reading files and directories, encompassing `open`, `opendir`, `fstat`, `read`, `readdir`, and `close`. These filesystem calls for specific file paths are converted into Interest Names. The `open` and `fstat` functions are managed identically, with the NDNc API maintaining flexibility in the sequence of their calls. On the receipt of a request for either function, an Interest Packet is created, targeting `Metadata` retrieval from in-network producers. For instance, metadata regarding the file located at */path/to/foo.root* is identified by the Interest Name: "$/\mathrm{ndnc}/\mathrm{xrootd}/\mathrm{path}/\mathrm{to}/\mathrm{foo.root}/32{=}\mathrm{metadata}$". Should the file remain inaccessible to any network producer, an application-level NACK response will be returned and the consumer will take appropriate action either by sending a refreshed version of the initial Interest Packet or by signaling an error to the higher levels of the application. In successful cases, the metadata, encapsulating comprehensive details about files or directories, will be enveloped in a Data Packet echoing the name of the requesting Interest. This `Metadata` [17] object includes essential details like `atime`, `btime`, `ctime`, and `mtime`, pivotal for data packet versioning within the network. A significant `Metadata` component is the maximum payload size, an adjustable attribute on the producer's end, defining the payload limit of a Data packet.

This payload dimension is crucial for file `read` operations. Invocations to this method specify both the file path and the byte range of interest. Depending on the byte range size and the producer's maximum payload size, these calls may be converted into one or multiple Interest Packets. In NDNc, the default size stands at **6144** bytes, with the maximum NDN packet size being **8800** bytes, inclusive of the packet's name, NDN metadata, and signature. Interest Names for byte range requests adhere to a specific format: "/<ndn-name-prefix>/<file-path>/v=<version>/segment=<number>". As an illustration, a request for the initial 64 bytes from a file (version 2) would be structured as: "/ndnc/xrootd/path/to/foo.root/v=2/seg=0". The directory reading logic remains largely unchanged, with the exception that the consumer retrieves full directory contents until the final Data Packet, or `FinalBlockId` [20], is received. This data is then stored persistently, with subsequent directory read calls only requiring iteration to relay the next directory entry to the application's higher levels.

For the OSS XRootD plugin, the file system's realization is based on the C++ interface provided by `XrdOssDF` [21] available in the framework's source code. The files are handled by the *XrdNdnOssFile* and the directories by the *XrdNdnOssDir* part of the *lib/xrdndnoss* [7] module in NDNc. Both classes capitalize on the same consumer instance to maintain a singular congestion window, overseeing all incoming and outgoing traffic from the plugin.

5 Performance Analysis and Results

The file transfer consumer developed for showcasing the performance of the NDNc library was first tested and demonstrated at the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC22) [22] incorporated with the NDN-DPDK forwarder. For the SC22, we deployed a high-bandwidth wide-area network (WAN) testbed connected with VLANs across the United States to evaluate the NDN applications' performance. During SC22, we performed multiple tests with various parameters and topologies as Figure 2 shows. To gain a clearer understanding of the test findings, we will outline a few representative results.

We first tested the performance of the NDNc library for retrieving files at Caltech. On the same host machine, using Docker containers, we've deployed the NDN-DPDK forwarder configured with seven forwarding threads and eighteen NDNc file transfer consumer applications. Each consumer was requesting one file at a time, each file having a size of 1 GB and a CMS name. All the files were cached in advance in DRAM by the forwarder and all consumers were configured identically: using the AIMD congestion control algorithm with an initial window size of 8192 packets. During a five minute test, we achieved an average goodput of 83.2 Gbps and a maximum goodput of 96.9 Gbps with an average delay of 1 ms. This was the first real-world demo of an NDN application achieving a maximum throughput speed close to 100 Gbps.

For the following two tests performed at the conference, we have used the same configuration for both the NDNc file transfer consumer applications as well as the NDN-DPDK forwarder: the consumers were configured to use a fixed congestion window size algorithm with a capacity of 8192 packets, while the forwarders were running six forwarding threads. The transferred files were cached in advance in DRAM by the forwarder and had same size and names used for the first test. Also, the lifetime of Interest Packets was set to 500 ms (exceeding this time before receiving a Data Packet in response would have triggered sending refreshed Interests until a retry limit would have been reached causing an error on the consumer side). The second test scenario was on a two-node linear topology from Caltech server where the clients were running to a Starlight (Chicago) server where the NDN-DPDK *fileserver* application [11] was residing. The two servers were connected via a 100 Gbps

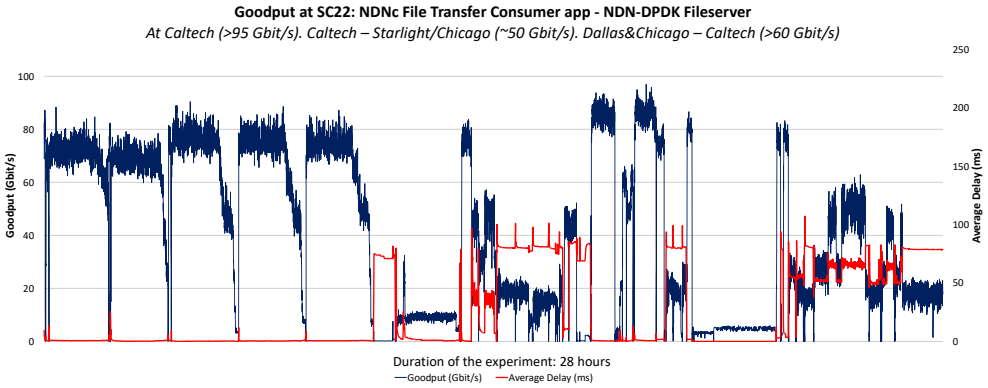

Figure 2: N-DISE results of NDNc goodput over WAN at Supercomputing Conference 2022

tagged VLAN link with a round-trip time (RTT) of 45 ms. During this analysis, we managed to achieve an average goodput of 42.1 Gbps and a 49.6 Gbps maximum goodput with 87 ms average delay. For the final demonstration, we have used two client nodes: one settled at the SC22 booth in Dallas and one at Starlight in Chicago, and one producer node at Caltech. All three nodes were connected by 100 Gbps tagged VLAN links with 33 ms RTT between Caltech and Dallas and 45 s RTT between Caltech and Starlight. The cumulative average goodput achieved was 49.6 Gbps, while the maximum value was 62.9 Gbps with 66 ms average delay. The final two tests were the first to showcase these high-throughput speeds over the WAN of any NDN application.

During SC22 we successfully conducted comprehensive testing of the XRootD OSS plugin. For these particular tests, we used the two-node linear topology from Caltech, where the XRootD server was running, to Starlight, where the producer together with the NDN-DPDK forwarder that had all the files pre-cached in advance were residing. The two servers were connected via a 100 Gbps tagged VLAN link. The OSS plugin was configured to use an AIMD congestion control window with an initial size of 32768 packets and 2000 ms lifetime for each expressed Interest packet. During tests that lasted more than one hour, we have used the `xrdcp` [23] command-line tool to copy both files and directories of files in order to test the throughput performance of the plugin. We have managed to achieve an average goodput of around 5 Gbps which is in line with the performance of a single NDNc POSIX consumer. Improvements on this front are discussed in Section 6.

6 Conclusions

This work presents the NDNc library that bridges the existing NDN C++ API developed by the community with the high-performance NDN-DPDK forwarder as well as the integration of the NDN primitives with the legacy XRootD framework through an OSS plugin. The new NDN file system implementation can provide location transparent data access and simplify the XRootD's data location services that are currently implemented through a set of redirectors [24]. Both NDNc and the XRootD plugin are still in the development stages and we are looking forward to further increase their capabilities by:

- Adding multi-threaded support to the memif interface. Currently, the shared memory interface is the only transport type supported in NDNc and applications developed using our library need to have a running NDN-DPDK forwarder on the same host machine. During

the SC22 experiments, we demonstrated great throughput capabilities of our applications and the forwarder, but we needed to run multiple consumers in order to achieve high numbers. The limitation comes from the memif interface implementation, which currently is single-thread. The forwarder shows that it is capable of delivering 100 Gbps throughput, but a single NDNc consumer is limited to around 5 Gbps. By adding multi-threaded support, we would be able to scale the throughput speed of one consumer with the number of threads.

- Adding asynchronous support to the file `read` implementation of the XRootD plugin. The framework offers support for async calls, an option that can be enabled through the configuration files at startup. At Caltech Tier2 [25], XRootD is configured to run async reads, which in theory should bring a bit more performance while lowering the CPU usage, thus we need to add this support to our *lib/posix* module in NDNc.
- Finalizing the work on the OSS plugin by adding support to all file system calls, not only those required to read files and directories.

The XRootD plugin is currently under extensive testing and comparisons with other storage solutions used at CMS are ongoing (i.e., HDFS, Ceph, XCache). These results will be critical in determining our next steps in development. For all analyses done in this paper, we made use of the NDN-DPDK *fileserver* application [11]. Although this served us well, we would like to extend our own producer application developed on NDNc, which would be capable of taking advantage of the file systems where the original data resides.

References

[1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, Named Data Networking (ACM SIGCOMM Computer Communication Review), **vol. 44**, no. 3, pp. 66–73, 2014

[2] E. Yeh, H. Newman, and C. Papadopoulos, SANDIE: SDN Assisted NDN for Data Intensive Experiments, 2017, NSF Award 1659403, https://www.nsf.gov/awardsearch/showAward?AWD_ID=1659403

[3] C. Iordache, R. Liu, J. Balcas, R. Sirvinskas, Y. Wu, C. Fan, S. Shannigrahi, H. Newman & E. Yeh Named Data Networking based File Access for XRootD. *EPJ Web Conf.*. **245** pp. 04018 (2020), https://doi.org/10.1051/epjconf/202024504018

[4] Named Data Networking Project, T. NFD: Named Data Networking Forwarding Daemon. (2023), https://docs.named-data.net/NFD/current/

[5] J. Shi, D. Pesavento & L. Benmohamed NDN-DPDK: NDN forwarding at 100 Gbps on commodity hardware. *Proceedings Of The 7th ACM Conference On Information-Centric Networking*. pp. 30-40 (2020)

[6] Named Data Networking Project, T. ndn-cxx: NDN C++ library with eXperimental eXtensions. (2023), https://docs.named-data.net/ndn-cxx/current/

[7] N-DISE Project, T. NDNc: A Lightweight Integration of ndn-cxx with NDN-DPDK to Achieve High-Throughput Performance in Scientific Applications. (2023), https://github.com/cmscaltech/sandie-ndn/tree/master/NDNc

[8] Y. Wu, F. Mutlu, Y. Liu, E. Yeh, R. Liu, C. Iordache, J. Balcas, H. Newman, R. Sirvinskas, M. Lo, S. Song, J. Cong, L. Zhang, S. Timilsina, S. Shannigrahi, C. Fan, D. Pesavento, J. Shi & L. Benmohamed N-DISE: NDN-Based Data Distribution for Large-Scale Data-Intensive Science. *Proceedings Of The 9th ACM Conference On Information-Centric Networking*. pp. 103-113 (2022), https://doi.org/10.1145/3517212.3558087

[9] L. Evans and P. Bryant, *LHC Machine*, Journal of Instrumentation, **vol. 3**, p. S08001, 2008

[10] D. Kutscher, S. Eum, K. Pentikousis, I. Psaras, D. Corujo, D. Saucez, T. Schmidt & M. Wählisch Information-Centric Networking (ICN) Research Challenges. (RFC Editor,2016,7), https://www.rfc-editor.org/info/rfc7927

[11] NIST, T. NDN-DPDK: High-Speed Named Data Networking Forwarder. (2023), https://github.com/usnistgov/ndn-dpdk

[12] Named Data Networking Project, T. PIT Token. (2023), https://redmine.named-data.net/projects/nfd/wiki/NDNLPv2#PIT-Token

[13] (FD.io), T. Shared Memory Packet Interface (memif) Library. (2023), https://s3-docs.fd.io/vpp/22.06/interfacing/libmemif/index.html

[14] The GraphQL Foundation, GraphQL: A query language for your API. (2023), https://graphql.org

[15] Named Data Networking Project, T. Name. (2023), https://docs.named-data.net/NDN-packet-spec/0.3/name.html

[16] Docker, T. Docker: Accelerated Container Application Development. (2023), https://www.docker.com

[17] S. Mastorakis, P. Gusev, A. Afanasyev & L. Zhang Real-Time Data Retrieval in Named Data Networking. *2018 1st IEEE International Conference On Hot Information-Centric Networking (HotICN)*. pp. 61-66 (2018)

[18] XRootD Project, T. Open File System & Open Storage System Configuration Reference. (2023), https://xrootd.slac.stanford.edu/doc/dev56/ofs_config.htm

[19] A. Dorigo, P. Elmer, F. Furano, and A. Hanushevsky, XROOTD - A highly scalable architecture for data access, WSEAS Transactions on Computers, **vol. 1**, no. 4.3, pp. 348–353, 2005

[20] Named Data Networking Project, T. FinalBlockId. (2023), https://docs.named-data.net/NDN-packet-spec/0.3/data.html#finalblockid

[21] XRootD Project, T. XrdOssDF Class Reference. (2023), https://xrootd.slac.stanford.edu/doc/doxygen/current/html/classXrdOssDF.html

[22] N-DISE Project, T. SC22 Network Research Exhibition. (2023), https://ndise.net/2022/11/19/sc22-demonstration/

[23] xrdcopy. (2023), https://xrootd.slac.stanford.edu/doc/man/xrdcp.1.html

[24] Using Xrootd Service (AAA) for Remote Data Access. (2023), https://twiki.cern.ch/twiki/bin/view/CMSPublic/WorkBookXrootdService#ReDirector

[25] CMS Caltech Tier2. (2023), https://tier2.hep.caltech.edu