"Expand: High Performance Storage System for HPC and Big Data Environments" (TED2021-131798B-I00)

High Performance Storage Sytems for HPC and Big Data (Expand)

D 2.1
Final Architecture of Expand

Universidad Carlos III de Madrid

June, 2025

CONTENTS

1. FINAL ARCHITECTURE OF EXPAND

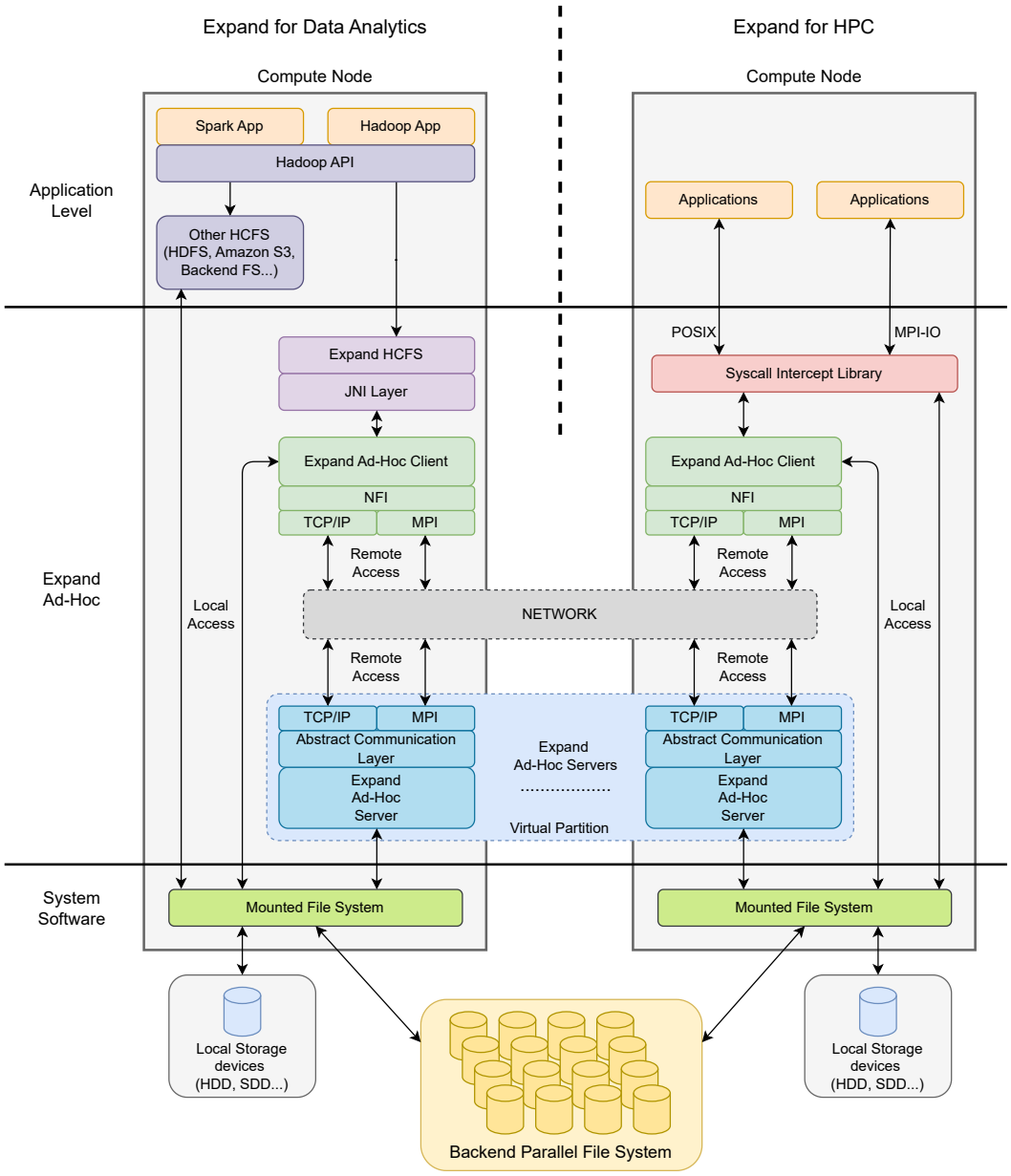


Figure 1.0.1: Expand final architecture design.

2. NEW METADATA MANAGEMENT IN EXPAND

Expand Ad-hoc combines several ad-hoc servers to provide a distributed partition. All files are striped across all servers to facilitate parallel access, with each server storing a subfile of the parallel file. A file in Expand consists of several subfiles, one for each server, with all subfiles fully transparent to Expand users. In Expand Ad-Hoc, the directory tree is replicated in all ad-hoc servers, creating the subfiles on demand. In other words, the subfiles are only created in the necessary ad-hoc servers.

Regarding metadata management, Expand Ad-Hoc does not have exclusive metadata servers. Instead, the ad-hoc servers also manage the metadata, which is stored in a little reserved space at the start of the subfiles. Each subfile of an Expand file has a small header at the beginning of the subfile. This header stores the file's metadata. This metadata includes the following information: block size, the first server (which identifies the server where the first block of the file resides), the replication factor, and the file distribution pattern used.

All the subfiles have that reserved space for the metadata, but only the server called the *master* of that file have the data stored in it. The master of each file can be obtained by using a hash function in the file name, which returns one server among all the ad-hoc servers in the partition used from the current configuration For example, as it can be seen in Figure 2.0.1, the master of the file named "f1" is the ad-hoc server 1, so the metadata is stored in that server.

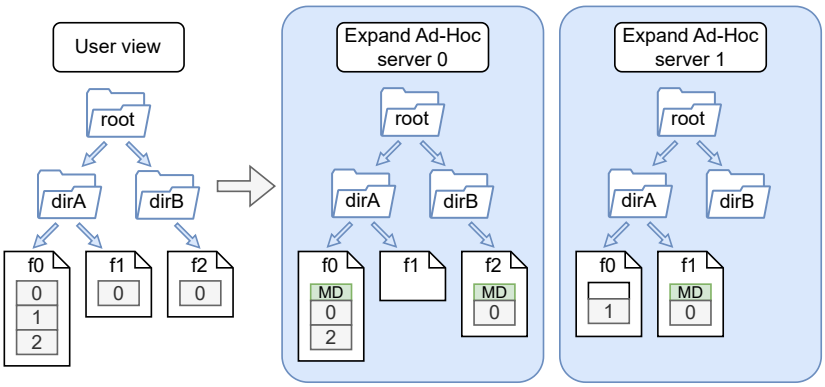


Figure 2.0.1: Expand Ad-Hoc data and metadata management.

For a more convenient and efficient way to find files, when a file is created, it is also created in the master server of its directory. This second file is created because when the user wants to use the `readdir` operation, we redirect that operation to the master server of that directory, which will have all the files. For example, in Figure 2.0.1, the master of the directory named "dirA" is the Expand Ad-Hoc server 0, so the file "f1" is created in that server but does not contain data. An example of the creation of a file on demand can be observed in Figure 2 for the file "f2", which is not created on the Expand Ad-Hoc server 1.

In Expand Ad-hoc, files are divided into blocks, with a block size that the user can specify, and distributed across the ad-hoc servers. Expand Ad-hoc uses a *mapping function* to obtain the subfile offset and the server where a given file offset is stored. Let $Serv$ be the set of servers of a partition in Expand, and Off, with $Off \subset \mathbb{N}$, the set of file offsets, the mapping function can be defined as:

$$f_{mapping} : Off \rightarrow Serv \times Off$$
$$f_{mapping}(off) = (serv_i, off_i)$$

Where off represents the offset within a file, $serv_i$ is the server where it is stored, and off_i is the offset within the subfile stored in $serv_i$.

Figure 2.0.1 shows a file distribution using a round-robin pattern (the only one currently implemented in Expand), starting in the master server of the file. In this case, the mapping function can be conceptualized as a mapping with computational complexity of $O(1)$. It can be seen in Algorithm 1. In this algorithm, *nserv* represents the number of servers in the partition, *b_size* the block size used and $first\_node$ the master server of that file.

Algorithm 1 : Algorithm for the mapping function for a round-robin pattern.

function A_MAPPING(off)
  $block = off/b\_size$
  $b\_line = block/nserv$
  $serv = (block + first\_node) \mod nserv$
  $serv\_off = b\_line \cdot b\_size + (off \mod b\_size)$
  return $(serv, serv\_off)$
end function

3. EXPAND FOR HPC APPLICATIONS

This chapter will explain the Expand[1] file system's final design and implementation for HPC applications. To this aim, we have completely redesigned the initial Expand file system, using MPI for communications between clients and servers.

Figure 3.0.1 shows the Expand components diagram with all elements. As may be seen, all of them could be integrated into a virtual partition without any problem by using on the client side the connectors included in Expand for each kind of server.

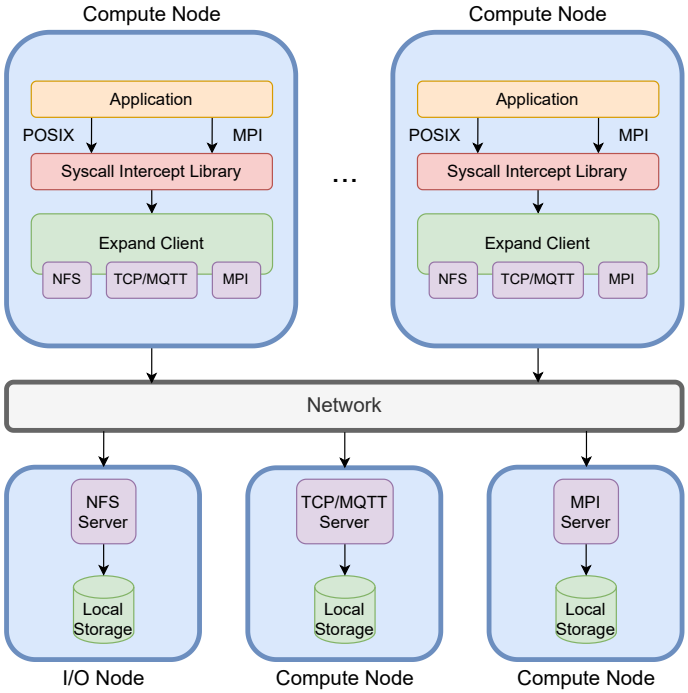


Figure 3.0.1: General Expand components diagram.

In the following sections, we explain the design and operation of Expand for HPC applications. We also show how they can be used as an ad-hoc file system in HPC environments to alleviate potential bottlenecks in the I/O systems created by data-intensive applications running on supercomputers.

3.1. The design of Expand for HPC applications

Expand file system for HPC applications architecture is based on an ad-hoc deployment, where data servers that run on different compute nodes communicate with each other using the MPI standard (see Figure 3.1.1). We decided to use the MPI standard for communications in this version of Expand because, since its appearance in 1994, it has been heavily used in HPC. As a result, it is available in all supercomputers and is highly optimized for

[1] `https://xpn-arcos.github.io/`

the different network technologies, following the MPI authors' goals: performance, scalability, and portability. In addition, MPI has already been used in distributed storage systems [1], and current MPI implementations have been upgraded to support modern high-performance communication technologies such as libfabric, and UCX, among others [2]–[4]. This allows MPI to remain a good choice for distributed storage systems.

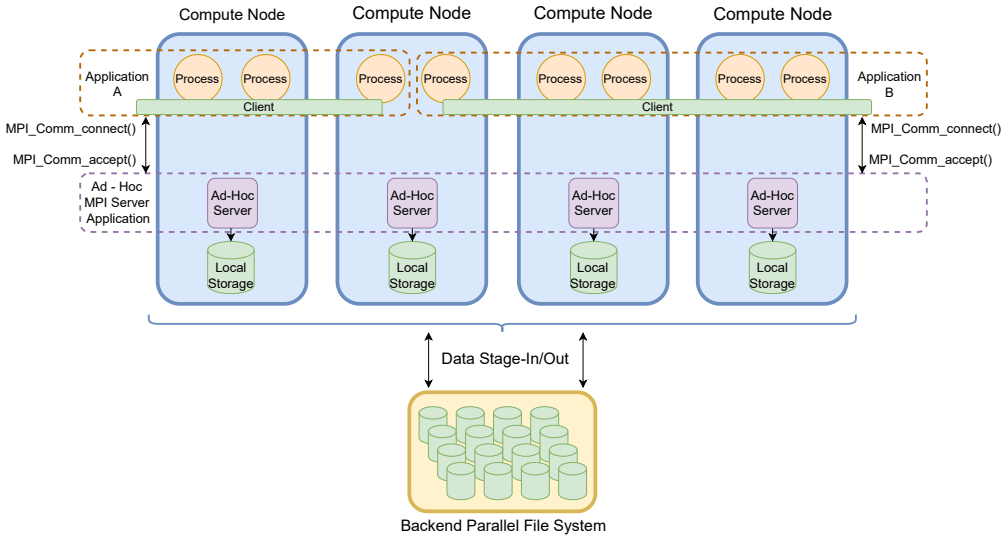


Figure 3.1.1: Architecture of the Expand File System.

As Expand is based on a client-server architecture, first, the Expand servers must be run as an MPI application on the set of compute nodes allocated to the application. If several application processes are run in one node, only one server is created in this node. Next, the application using the ad-hoc file system is deployed across all nodes with the Expand client. The way of deploying both elements using MPI communications is shown in Figure 3.1.2.

On the one hand, Expand Ad-Hoc servers must first initialize MPI using `MPI_Init` and open an MPI port using `MPI_Open_port`. Secondly, upon client request, each ad-hoc server has to send the MPI port opened in the previous phase to the clients using the server's control socket. Thirdly, once these two steps are completed, the servers wait until the clients connect to them using `MPI_Comm_accept` with `MPI_COMM_SELF`. When the connection is established, the clients and servers will use a point-to-point connection to send (`MPI_Send`) and receive (`MPI_Recv`) the requests and the necessary data to perform the different operations on the file system. When the client wants to end the work session, the servers will disconnect from the clients using `MPI_Comm_disconnect`. Finally, the servers will close the port using `MPI_Close_port` and finalize MPI using `MPI_Finalize`.

On the other hand, Expand Ad-Hoc clients, first, also have to initialize MPI using `MPI_Init`. Second, the client will receive the MPI port from the server to be connected to, communicating with the server through the server's control socket. Thirdly, the client will connect to the ad-hoc server by using `MPI_Comm_Connect` with `MPI_COMM_WORLD` on the server's port, which was received in the previous step. After this operation, a working session is established where the clients and servers exchange requests and data to carry out the Expand Ad-Hoc operations. When the working session ends, as it happens on the server, the client must disconnect from the server using `MPI_Comm_disconnect`. And finally, the client will finalize MPI using `MPI_Finalize`.

```
// MPI Ad-Hoc Server pseudocode

send_port(ctrl_sck, port_name) {
  cs = accept(ctrl_sck, &ctrl_addr, ...)
  ...
  send_port(cs, port_name, ...)
  close(cs)
}

main(...) {
  /* Phase 1: init + port */
  int ctrl_sck = server_socket(ctrl_addr)
  MPI_Init(argc, argv)
  MPI_Open_port(MPI_INFO_NULL, port_name)

  while (1) {
    /* Phase 2: send port */
    send_port(ctrl_sck, port_name, ...)


    /* Phase 3: accept + requests */
    MPI_Comm_accept(port_name, MPI_INFO_NULL, 0,
          MPI_COMM_SELF, cli_comm)
    ...
    MPI_Comm_disconnect(cli_comm)

    /* Phase 4: finalize */
    MPI_Close_port(port_name)
    MPI_Finalize()
  }
}
```

```
// MPI Ad-Hoc Client pseudocode

recv_port(ctrl_sck, port_name) {
  ctrl_sck = connect(ctrl_addr, ...)
  recv_port(ctrl_sck, port_name, ...)
  close(ctrl_sck)
}


main(...) {
    /* Phase 1: init */
    int ctrl_sck = socket(..., IPPROTO_TCP)
    MPI_Init(argc, argv)



    /* Phase 2: receive port */
    recv_port(ctrl_sck, port_name, MPI_MAX_PORT_NAME)

    /* Phase 3: connect + requests */
    MPI_Comm_connect(port_name, MPI_INFO_NULL, 0,
          MPI_COMM_WORLD, srv_comm)
    ...
    MPI_Comm_disconnect(srv_comm)

    /* Phase 4: finalize */

    MPI_Finalize()

}
```

Figure 3.1.2: Pseudocode of the MPI interconnection between ad-hoc servers and clients.

In Expand file system for HPC applications, local storage devices such as HDD, SSD, or Shared Memory on various servers deployed ad-hoc for an application are used to create distributed partitions. The goal is to store data near the application, thus reducing the need for remote access to the backend parallel file system. To store data locally, we rely on services provided by the local operating system, such as the ones accessed through the POSIX interface.

The application's processes and the Expand servers might be deployed on the same compute nodes because Expand does not need dedicated storage servers. Although applications and servers run on the same compute nodes, their interference is minimal. This is because applications mainly perform computations, while ad-hoc servers mainly perform I/O. Furthermore, in operating systems, while a process is blocked waiting for an I/O operation, another process runs on the CPU. Hence, the I/O operations of ad-hoc servers and the computation of applications overlap. In addition, today's multicore architectures help to alleviate this problem for two reasons: there is a large number of cores available for both the ad-hoc servers and the application, and it is possible to pin more cores to the application, if required, with minimal interference to the ad-hoc server.

This is the recommended deployment because it minimizes the number of accesses to remote data. Nevertheless, the processes of the application and the Expand servers can be deployed on different compute nodes if is needed. In Figure 3.1.1, you can see both alternatives.

Listing 3.1 shows an example template of a deployment using SLURM (Simple Linux Utility for Resource Management) [5]. First, the machinefile is created with the allocated nodes using `scontrol` command. Second, the ad-hoc servers are deployed, and, third, the application using Expand is deployed.

Listing 3.1: Example of SLURM job template.

```
#!/bin/bash

#SBATCH --job-name=ior_xpn_test
#SBATCH --nodes=32

#Get allocated nodes by SLURM
scontrol show hostnames ${SLURM_JOB_NODELIST} > machinefile

# Expand Ad-Hoc servers deployment
srun -n 32 -w machinefile ./xpn_mpi_server &

# IOR benchmark with Expand Ad-Hoc deployment
export LD_PRELOAD=./xpn_bypass.so

srun -n 256 -w machinefile ./ior -w -r -o /tmp/expand/xpn/ior.txt -t 1024k -b 1024k -s 4096 -i 10
```

Figure 3.1.3 describes the internal design of Expand as an ad-hoc parallel file system. This figure shows the different software layers (syscall intercept library, Expand client, etc.) involved from when an application performs an I/O operation until it is completed.

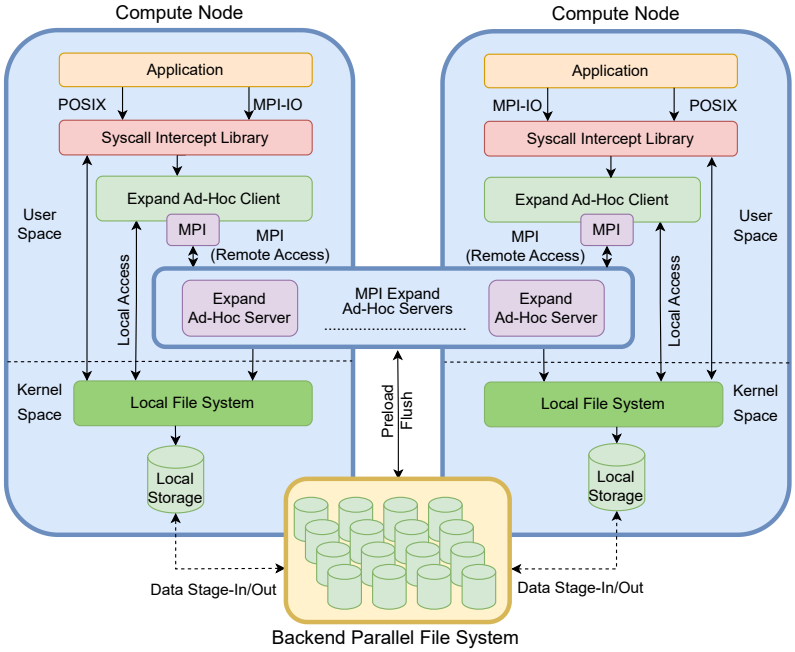


Figure 3.1.3: Expand architecture details.

As seen in this figure, an Expand server is deployed on each compute node, which is responsible for accessing the local storage devices. Furthermore, to mitigate bottlenecks, this server has an I/O request queue that allows buffering of the request peaks, as well as the local file system overload.

In addition, it can also be seen that Expand uses the local file system on the compute nodes to access the local storage devices. This lets Expand adapt to different local storage configurations (HDD, SSD, NVMe, or SHM) in a transparent way.

Finally, it should be noted that applications access Expand services using the system call interception library since this is responsible for calling the Expand services, as will be detailed in Section 3.5.

3.2. Definition of the Expand virtual partition

To define the Expand partitions, creating a configuration file in INI-file format is necessary, as shown in the example in Listing 3.2. Each partition is defined in this configuration file by specifying its name, the Expand block size (partitioning size), the URI of the nodes that comprise it, and other parameters. The URI of the node includes the protocol (e.g., mpi_server, nfs3, etc.), the hostname (or IP address), and the mount point in the compute node's local storage.

The configuration file displayed in Listing 3.2 outlines the definition of a partition named `p1`. This partition is comprised of `two` ad-hoc servers (`server1` and `server2`) that use the `mpi_server` protocol, with a block size set at `64 KiB`. Additionally, the file paths for both servers are `/mnt/xpn`.

Before running an application that uses Expand, the user must create the INI file defining the Expand virtual partition. Expand provides a simple script (`mk_conf.sh`) that can generate the content of this file given all the partition parameters. The Expand client library relies on a configuration file to establish network connections between all Expand clients and servers.

Listing 3.2: Example of configuration file used to define an Expand virtual partition.

```
1 [partition]
2 partition_name = p1
3 bsize = 64k
4 server_url = mpi_server://server1/mnt/xpn
5 server_url = mpi_server://server2/mnt/xpn
```

3.3. Parallel access

When a working session with a file is started (for example, when opening a file), a virtual file handler is created in Expand. This virtual file handler is used for all operations performed on this file during that session. Expand can access all the associated subfiles containing the associated data through this virtual file handler.

When a subfile is accessed, the Expand library uses the virtual file handler and breaks down the application operation into separate parallel sub-requests that are sent to the servers involved. This way, if k ad-hoc servers are involved in a request, Expand can perform k requests simultaneously to the servers using threads to speed up the process. This applies to both data and metadata operations carried out in Expand.

3.4. Data locality

As explained in Section 3.1, Expand servers can be executed on the compute nodes where the parallel application is being run. When this occurs, the application data is distributed among all compute nodes. In some cases, this allows access to the data requested by an application process using the local file system of the compute node by executing this process in the same node where the data is stored (see arrow tagged as "Local Access" in Figure 3.1.3). This helps to optimize data access from the Expand client and avoids delays and network-related issues that might occur when using remote data access.

It is important to keep in mind that data locality is crucial when multiple parallel applications (workflows) need to process the same dataset on the same nodes.

3.5. System call interception library

Expand offers a fully compliant POSIX and MPI-IO interface. With this aim, Expand Ad-Hoc provides a POSIX system call interception library that allows the use of Expand without modifying the source code of existing or legacy applications. This library intercepts the POSIX system calls made while a program is running. If a file is stored in Expand, which is detected because the file pathnames include the Expand mount point, the corresponding Expand API function is called. However, if the system call is made on the compute node's local file system, the corresponding `libc.so` system call will be executed (see Figure 3.1.3). To load the intercept library before everything else, including the Linux `libc.so`, without needing super-user permissions, we have chosen to use `LD_PRELOAD`.

Also, Expand provides a native API with functions that are very similar to the POSIX API calls, both in their names (e.g. `xpn_open, xpn_read`, etc.) and in the arguments they receive (e.g. `xpn_read(fd, buf, nb)`). To use the Expand API, the source code must be altered, and the application must be compiled with the Expand dynamic library.

3.6. I/O Data staging

If an application needs initial data when using an ad-hoc file system, data staging is needed to preload (stage-in) data from the backend file system (e.g., GPFS) to the ad-hoc file system servers. The transfer of data between the backend file system and Expand file system is done through an MPI program that runs on the Expand servers. This program creates a replicated directory tree and the subfile in parallel on each server for every file stored in the backend.

Likewise, when the application that uses the ad-hoc file system finalizes its execution, persistent data must be stored in the backend file system because the ad-hoc servers will no longer be available. To complete this task, a flush operation (stage-out) is provided to move data from the Expand servers to the backend file system. In Expand, an MPI program, similar to the one mentioned earlier, is used for this data transfer. The program runs on the ad-hoc servers and reads in parallel subfiles stored in the servers, generating the file that will be saved in the backend file system.

Furthermore, the flush operation can also be used during the application execution to store the generated data persistently in the back-end file system. Therefore, this operation helps mitigate data loss if a software or hardware issue occurs during the application execution. Moreover, as this operation is performed on demand, if the data generated by the application is temporary, then by skipping this operation, we avoid the potential overhead caused by its execution.

The Figure 3.1.3 shows both preload and flush operations.

4. EXPAND FOR BIG DATA APPLICATIONS

The solution proposed for Expand in Big Data applications involves designing a connector that enables using Apache Spark by Expand. The main goal is to exploit the benefits provided by both platforms. Spark applications run on the backend file system by default, but there are connectors for other frameworks, such as HDFS or Amazon S3. Their use is specified by the use of URIs: `file://` for the backend file system; `hdfs://` for HDFS; and `s3://` for Amazon S3. The use of Expand is expected to be similar to these platforms.

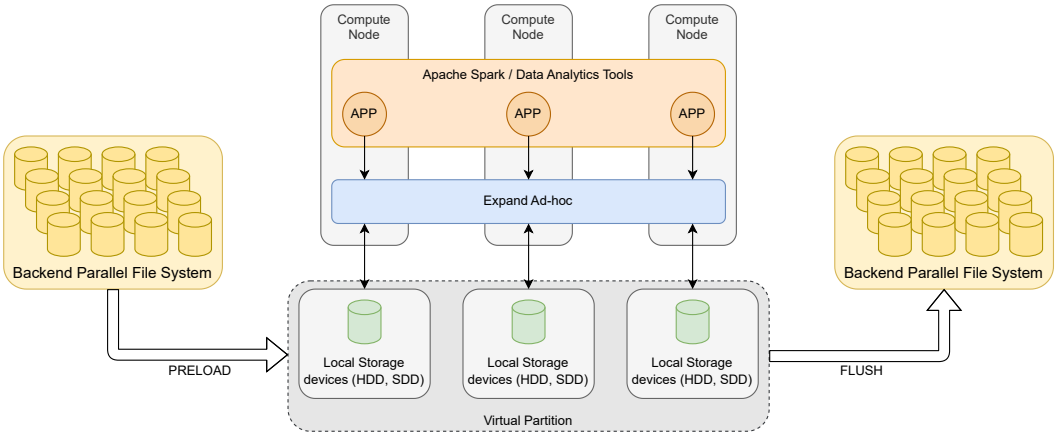


Figure 4.0.1: Workflow for Spark using Expand.

As Expand is designed to be deployed for specific applications, the workflow shown in Figure 4.0.1 must be followed. First, the data needed by the application shall be distributed among the compute nodes. This operation is called preload. When the preload operation is completed, Spark or any data analytics application can be deployed using the distributed data in the node's local file system. Finally, the data produced by the application that is wanted to be preserved shall be saved in the backend file system. This operation is called flush. Both the preload and flush operations are available in the Expand utilities.

To use Expand in Big Data environments, a two-layer connector has been designed as shown in Figure 1.0.1. The top-level layer consists of tailoring Expand to the Hadoop environment by extending the `File System` class from the Hadoop project [6]. This will allow the creation of Expand clients through the Spark tasks. However, since the Expand client functions are written in C, it is necessary to create another layer to translate them to Java functions. To achieve this purpose, the Java Native Interface (JNI) is used to convert the client requests to the servers and the server's response to the clients so that they can communicate. The following sections explain in more detail the design of these layers.

The connector is offered in a `jar` package for ease of use. For this, the Maven [7] tool is used, which makes it easier to compile and package the connector. In addition, it allows the inclusion of the project dependencies.

4.1. Expand JNI layer

The implementation of the JNI layer is divided into four stages. First, it is necessary to create the Java interface with the sign of the functions required for implementing the Expand clients. Secondly, this interface must be compiled, and the header file of the necessary functions must be generated in C. Afterwards, it is required to implement the functions in C. In most cases, the goal is to call the Expand functions; however, some of them will have other requirements that are necessary for proper system performance. Finally, compiling these last sources and generating a dynamic library for the connector will be necessary.

The JNI layer must translate the Expand client functions, the POSIX standard `Stat` structure, and the POSIX standard flags used by the Expand client functions. For each of the last two, an object is created to be translated into a structure that the servers can handle. To use the functions by the upper layer, it is necessary to specify the signature of the functions that the Hadoop layer will use. For ease of use, the same name is used as the Expand client functions preceded by the `jni_` prefix, as shown in Table 4.1.1.

Table 4.1.1: JNI Expand client functions.

Function	Parameters
jni_xpn_chdir	String path
jni_xpn_chmod	String path, short mode
jni_xpn_close	int fd
jni_xpn_creat	String path, long mode
jni_xpn_destroy	
jni_xpn_fstat	int fd
jni_xpn_get_block_locality	String path, long offset, String[] url_v
jni_xpn_getcwd	String path, long size
jni_xpn_init	
jni_xpn_lseek	int fd, long offset, long whence
jni_xpn_mkdir	String path, short mode
jni_xpn_open	String path, long flags
jni_xpn_read	int fd, ByteBuffer buf, long size
jni_xpn_rename	String src, String dst
jni_xpn_rmdir	String path
jni_xpn_stat	String path
jni_xpn_unlink	String path
jni_xpn_write	int fd, ByteBuffer buf, long count

After creating the Java objects and the needed interface, compiling the sources to obtain the header file is necessary. This can be achieved with the `javac` command specifying the `-h` option.

Using the header file, it is possible to create a C file to call the Expand client functions. These functions require capturing the parameters specified in the signatures and translating them to C-intelligible values. Then, the Expand client functions are invoked, and the result returned by the server is converted so that it can be passed to the clients.

Finally, it is possible to compile these last sources as any C program by indicating the Expand dependencies. That way, a dynamic library can be obtained, the Expand servers can handle the incoming Java requests, and their responses can also be translated.

4.2. Expand Hadoop layer

The main goal of this layer is to convert Expand to a Hadoop Compatible File System (HCFS) so that the Spark tasks can handle the clients. To achieve this goal, extending the `File System` class available in the Hadoop project [6] is necessary. This class contains the abstract methods shown in Table 4.2.1 that must be overwritten. The functions shown in this table allow Spark to perform I/O and metadata operations such as list file status (`getFileStatus`), list directory status (`listStatus`), create directories (`mkdirs`), read files (`open`), write files (`append` and `create`), rename files (`rename`), or delete files and directories (`delete`).

Table 4.2.1: Overwritten Hadoop File System abstract methods.

Function	Parameters
append	Path f, int bufferSize, Progressable progress
create	Path f, FsPermission permission, boolean overwrite, int bufferSize, short replication, long blockSize, Progressable progress
delete	Path f, boolean recursive
getFileStatus	Path f
getUri	
getWorkingDirectory	
listStatus	Path f
mkdirs	Path path, FsPermission permission
open	Path f, int bufferSize
rename	Path src, Path dst
setWorkingDirectory	Path new_dir

Besides, the functions shown in Table 4.2.2 must be overwritten to ensure the correct operation and improve the system performance. In these functions, Expand clients can be raised (`initialize`), paths can be classified into a file or a directory (`isDirectory`), task locality is implemented (`getFileBlockLocations`), and files and directories permissions can be modified (`setPermission`).

Table 4.2.2: Overwritten Hadoop File System methods.

Function	Parameters
initialize	URI uri, Configuration conf
isDirectory	Path f
getFileBlockLocations	FileStatus file, long start, long len
setPermission	Path path, FsPermission perm

5. FAULT TOLERANT SUPPORT AND MALLEABILITY

5.1. Fault tolerance

The fault tolerance design in Expand Ad-Hoc is based on block replication. This means that the data is replicated as many times as the replication factor that the user selected. The replication factor is the number of copies of a block.

This approach allows a failure in up to $N - 1$ ad-hoc servers, where N is the replication factor. This is possible because, with any N replication factor, the blocks and the metadata are stored in N ad-hoc servers, as shown in Figure 5.1.1.

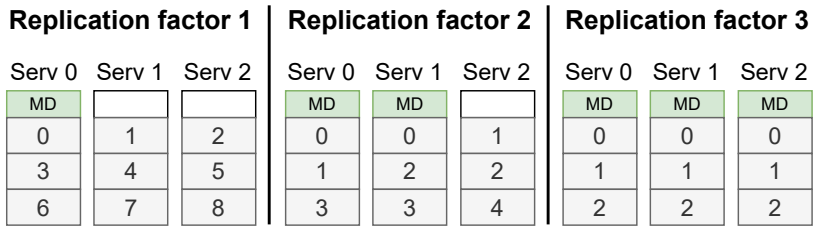


Figure 5.1.1: Data block distribution with replication factor 1, 2, and 3 in Expand Ad-Hoc.

Given a replication factor R, where $R \in \mathbb{N}$, the block mapping function is formally expressed as follows when Expand Ad-Hoc uses replication:

$$f_{mappingRepl} : Off \rightarrow (Serv \times Off)^R$$
$$f_{mappingRepl}(off) = ((serv_i, off_j) \dots (serv_k, off_l))$$

For $R = 2$, in other words, a replication factor of 2, the block mapping function is:

$$f_{mappingRepl}(off) = ((serv_0, off_0), (serv_1, off_1))$$

Where off represents a file offset, off_0 is the subfile offset in $serv_0$, and off_1 is the subfile offset in $serv_1$ where the off is replicated.

In a round-robin distribution pattern, all the blocks are distributed among the ad-hoc, as illustrated in Figure 5.1.1 with 1, 2, and 3 replication factors. As with the previous scenario, the function that performs this mapping with replication is of complexity $O(1)$, which can be seen in Algorithm 2. This algorithm returns only the i element of the tuple $((serv_i, off_j) \dots (serv_k, off_l)$.

There are two possible approaches to perform the detection of the servers with error. One uses any MPI implementation by default, which can detect when an ad-hoc client attempts to connect to one ad-hoc server and the server has problems. It is marked as erroneous, and the client will not use that server. Another approach uses the OpenMPI 5.0 implementation that offers the module ULFM (User-Level Fault Mitigation) [8]. With this

Algorithm 2 : Algorithm for the mapping function for round-robin and R replication factor, that only calculate the i element of the tuple $((serv_i, off_j) \dots (serv_k, off_l))$

function A_MAPREPL(off, i)
  $block = off/b\_size$
  $b\_repl = block \cdot R + i$
  $b\_line = b\_repl/nserv$
  $serv = (b\_repl + first\_node) \mod nserv$
  $serv\_off = b\_line \cdot b\_size + (off \mod b\_size)$
  return $(serv, serv\_off)$
end function

module, the ad-hoc client can detect in the middle of the application that if one communication with a server fails, it marks that server as erroneous to stop using it.

Thanks to the replication factor implemented with the fault tolerance, we have developed an optimization in the read operations that takes advantage of the increase in data locality. This is achieved by having the ad-hoc client check whether the data requested by the user is replicated in one ad-hoc server deployed in the same compute node. If that is the case, the ad-hoc client can directly access the data in the local storage, avoiding the communication overhead with the ad-hoc server.

5.2. Malleability

Three distinct malleability algorithms were developed, each employing a unique methodology and showing different advantages and disadvantages.

The primary distinction between these algorithms lies in the method utilized to move the data from the existing partition to the newly configured one. All the algorithms stop and start the servers to facilitate the brand-new start of the new partition.

It is important to note that, as previously explained, all malleability algorithms take into account the data replication utilized in the fault tolerance model. In the event that a server is down, data will be obtained from replications on other servers.

In a formal context, the malleability operation can be defined as the movement of a partition, stored in a set of servers designated as S, to another set of servers, designated as R. The number of servers in each set allows for classifying this operation as either expand or shrink: an expand operation occurs when $|S| < |R|$, and a shrink operation occurs when $|R| < |S|$.

The relationship between S and R can be conceptualized in the following manner:

$$S \subset R$$

$$R \subset S$$

$$S \cap R = \emptyset$$

To perform this operation, we formally define the following malleability function:

$$f_{malleablity} : (S, Off)^P \rightarrow (R, Off)^Q$$
$$f_{malleablity}(Serv_i, Off_i)^P = (Serv_j, Off_j)^Q$$

Applying this function to each file enables the transfer of data block-by-block from a partition S with the replication factor P to a partition S with replication factor Q.

Given a malleability mapping, we define the cost function of this malleability function as follows:

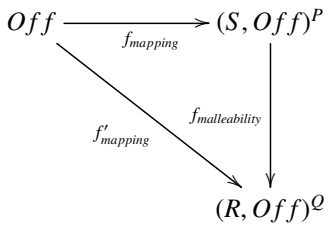
$$Cost : (f_m, S, R, File) \mapsto \mathbb{N}$$

This function determines the number of operations required to move the file designated as $File$ from partition S to partition R where the malleability function is f_m.

In order to accurately execute the cost function, it is necessary to obtain the blocks in a file, taking into account the replication factor. The function is defined as follows:

$$f_{blocks}(File) \rightarrow \mathbb{N}$$

Given a malleability $f_{malleability}$ and a block mapping function $f_{mapping}$, both functions are related as follows:



The new mapping function in the new partition can be defined as:

$$f'_{mapping} = (f_{malleability} \circ f_{mapping})$$

The following sections present three different algorithms, ordered according to their efficiency calculated with this cost function.

5.2.1. Malleability: backend rebuild

The initial algorithm employs the supercomputer's backend file system to facilitate the data transfer process. This is achieved by first transferring all data from the current Expand Ad-Hoc partition to the backend file system via an MPI application that runs in parallel on all the nodes that contain servers. Upon completion of this transfer, the data is then transferred in the opposite direction to the new configuration of the partition. A more illustrative representation can be observed in Figure 5.2.1.

The cost of this malleability function can be defined as follows:

$$Cost(f_{backendRebuild}, S, R, File) = f_{blocks}(File) \times 2$$

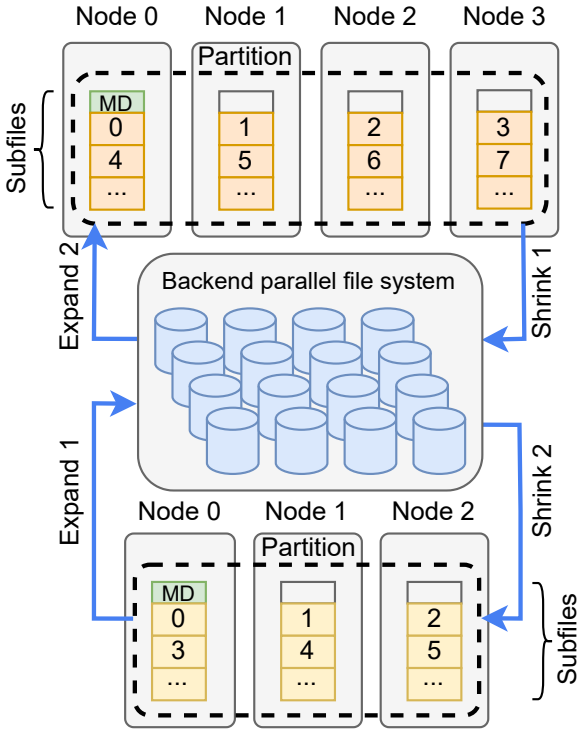


Figure 5.2.1: Malleability: backend rebuild.

This algorithm relies on the supercomputer's backend file system for both data writing and reading, so it depends on that system's bandwidth. As a result, the transfer speed from one Expand Ad-Hoc partition to another is dependent on the speed of the backend file system. It is evident that this approach is not the most efficient one. Therefore, we developed the next algorithm, which is explained in the following section.

5.2.2. Malleability: direct rebuild

We developed a new MPI application with this next algorithm to eliminate dependency on the backend file system. This MPI application directly transfers the data blocks from the old Expand Ad-Hoc partition to the new one. To make this possible, we consider the two sets of nodes:

$$S = \{S_1, S_2, ..., S_k\}$$
$$R = \{R_1, R_2, ..., R_r\}$$

The variable S refers to the set of servers in the old partition, and R corresponds to the set of servers in the new partition.

The MPI application starts a number of processes equal to $|S|+|R|$. For the sake of convenience, the processes

in the set S are designated as *readers* having $|S|$ as the number of *readers*, while the set of processes in R are designated as *writers* having $|R|$ as the number of *writers*. The *readers* are responsible for reading each block, calculating its new position and server, and sending it to the corresponding *writer* process. Consequently, the *writers* are awaiting the reception of the blocks and their respective positions. Upon the receipt of a block, the writer only writes it.

The cost of this malleability function can be defined as follows:

$$Cost(f_{directRebuild}, S, R, P, File) = f_{blocks}(File)$$

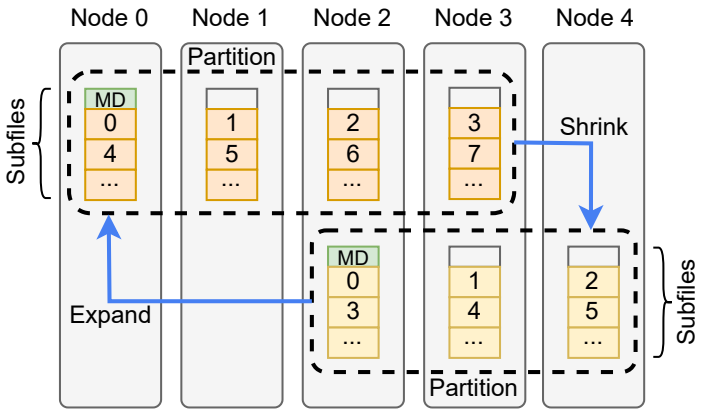


Figure 5.2.2: Malleability: direct rebuild.

One example of this malleability operation can be observed in Figure 5.2.2, which illustrates the expanding and shrinking of servers within the Expand Ad-Hoc malleability. In this example, we can see the expanding of three servers to four servers, as well as the shrinking of four servers to three servers. For the expanding, the set of readers is $S = \{Node_2, Node_3, Node_4\}$ and the set of writers is $R = \{Node_0, Node_1, Node_2, Node_3\}$, so the number of process to run is $|S| + |R| = 7$, being readers $|S| = 3$, and writers $|R| = 4$. Likewise, for the shrinking, the set of readers is $S = \{Node_0, Node_1, Node_2, Node_3\}$ and the set of writers is $R = \{Node_2, Node_3, Node_4\}$, so the number of processes to run is $|S| + |R| = 7$, being readers $|S| = 4$, and writers $|R| = 3$.

5.2.3. Malleability: metadata rebuild

We realized that the previous algorithm was not the most efficient design, leading to the creation of this newer and more efficient algorithm. The objective of this algorithm is to move only the minimal and necessary data from one partition to another.

For example, when there is an expand operation in which only additions to the new partition are made, the data can be accessed through the old servers and does not need to be moved. In other cases, when there is a shrink operation, only the data that cannot be accessed because the servers are removed must be moved.

This function's cost depends on whether the operation is of expand or shrink type. The cost for an expand

operation is defined as follows:

$$Cost(f_{metadataRebuild}, S, R, P, File) = 0$$

The cost for a shrink operation can be defined as:

$$Cost(f_{metadataRebuild}, S, R, P, File) = \frac{f_{blocks}(File)}{|S|} * (|S \setminus R|)$$

In order to achieve this malleability, we take advantage of the metadata and save the sections of the file in which the malleability was performed. A new mapping function with a $O(n)$ complexity was developed for the new file structure, where n is the number of reconstructions. This allows the data to be located within the files, regardless of the number and order of malleability operations. Despite the complexity being $O(n)$, the typical number of server reconstructions is low, resulting in a time complexity close to that of the previous mapping function with a complexity of $O(1)$.

The formality of this mapping function is identical to that of the mapping with replication, as it also takes into account data replication.

$$f_{mappingMalleability} : Off \rightarrow (Serv \times Off)^R$$
$$f_{mappingMalleability}(off) = ((serv_i, off_j) \ldots (serv_k, off_l))$$

For $R = 2$, in other words, a replication factor of 2, the block mapping function is:

$$f_{mappingMalleability}(off) = ((serv_0, off_0), (serv_1, off_1))$$

The mapping algorithm consists of a loop that iterates over each malleability section defined in the metadata, calculating the position and server of the blocks. For performance reasons, the algorithm initially verifies the absence of malleability and redirects to the original function, thereby maintaining the complexity in constant time ($O(1)$). If malleability is present, the algorithm iterates through each segment, calculating whether the block is located within the actual segment and, therefore, calculating its position and server. A segment is the section of the file that has been written to a partition. For example, if 3 blocks are written in one partition and malleability is performed to another partition, in which 2 new blocks are written, we would have two segments in the file, one of 3 blocks and one of 2 blocks. It can be a shrinking or expanding segment when the shrink or expand malleability operation has been performed respectively. In the case of the block situated within a shrinking segment, a further recalculation is performed to determine the new position and server. A more visual representation of the mapping algorithm can be found in Algorithm 3.

So, the MPI application developed to perform this algorithm is responsible for calculating the data that needs to be inserted into the metadata. If it is a shrink operation, it is also responsible for moving only the data from the server that will be removed to the other ones. The data movement is performed similarly to the previous malleability algorithm, in which the servers that will be removed are the *readers* and the others are the *writers*.

For illustrative purposes, three examples of the new mapping can be found in Figures 5.2.3, 5.2.4, and 5.2.5.

Algorithm 3 : Algorithm for the malleability mapping function with metadata for round-robin and R replication factor, returning only calculate the i element of the tuple $((serv_i, off_j) \dots (serv_k, off_l))$.

```
function A_MAPPINGMALLEABILITY(off, i)
    using F_mR = F_mappingRepl
    if not malleability then
        return F_mappingRepl(off, i)
    end if
    for segment in segments do
        data_nserv = mdata.data_nserv[i]
        offset = mdata.offset[i]
        calculate_segment_serv_off(off, i)
        if block in segment then
            serv_off, serv = F_mR(off, i)
            serv_off = add_segment_serv_off()
        end if
        if block in segment_shrink then
            serv_off, serv = F_mR(serv_off, i)
            serv_off = add_segment_serv_off()
        end if
    end for
    return (serv ,serv_off)
end function
```

As shown in Figure 5.2.3, the expanding process can be observed. Figure 5.2.4 presents the process of shrinking. Figure 5.2.5 demonstrates a combination of expansion and contraction. In these figures, the metadata fields "data_nserv" refer to the number of servers present in each segment, while "offset" specifies the block at which the segment begins.

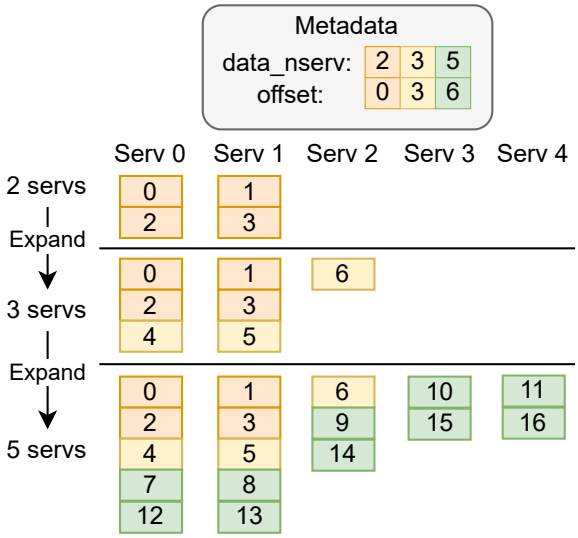


Figure 5.2.3: Malleability: metadata rebuild using expand. From 2 to 3 to 4 servers.

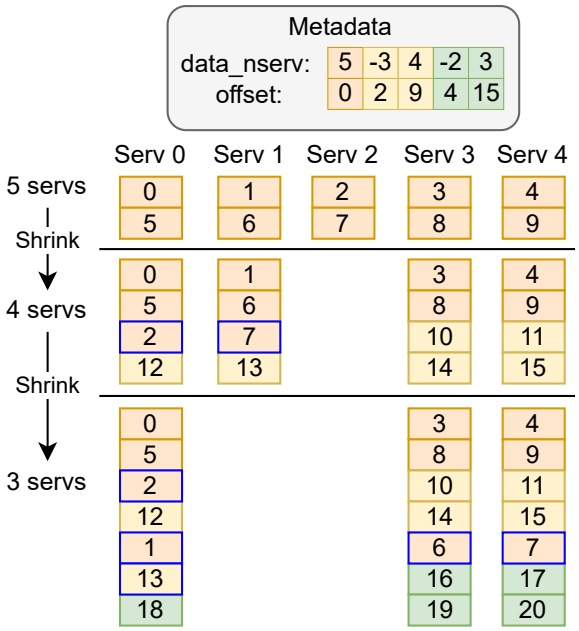


Figure 5.2.4: Malleability: metadata rebuild using shrink. From 5 to 4 to 3 servers.

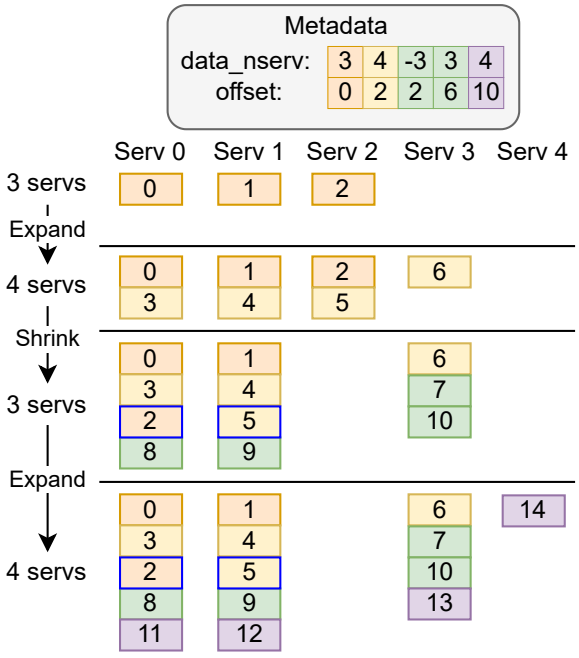


Figure 5.2.5: Malleability: metadata rebuild using both expand and shrink. From 3 to 4 to 3 to 4 servers.

BIBLIOGRAPHY

[1] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using MPI in high-performance computing services," in *Proceedings of the 20th European MPI Users' Group Meeting*, 2013, pp. 43–48.

[2] L. A. N. L. Howard Pritchard, *Open MPI and recent trends in network APIs*, Accessed Nov. 9, 2023. [Online], 2016. [Online]. Available: `https://www.openfabrics.org/images/eventpresos/2016presentations/110mpiandapi.pdf`.

[3] A. MPICH, *MPICH readme*, Accessed Nov. 9, 2023. [Online], 2023. [Online]. Available: `https://www.mpich.org/static/downloads/4.1.2/mpich-4.1.2-README.txt`.

[4] M. A. Open, *Open MPI readme*, Accessed Nov. 9, 2023. [Online], 2023. [Online]. Available: `https://docs.open-mpi.org/en/main/tuning-apps/networking`.

[5] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[6] A. Hadoop, *Github repository*, 2024. [Online]. Available: `https://github.com/apache/hadoop`.

[7] Apache Software Foundation, *Welcome to Apache Maven*, 2024. [Online]. Available: `https://maven.apache.org/`.

[8] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of mpi communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013. DOI: `10.1177/1094342013488238`.