Chapter 2

Background and Related Work

This chapter provides solid background to a rich variety of topics[1]. After discussing each topic, the state of the art and related work are discussed. In this context recent research and several software tools are shown[2].

The first three sections provide more detail about HPC hardware and software, and their performance implications. First, the concepts of file systems and several real world representatives are introduced in Section 2.1. Then, aspects involved in the performance of parallel applications are discussed in Section 2.2. This indicates the complexity of those systems, but also enables classifying of relevant aspects. Next, the Message Passing Interface is introduced in Section 2.3 – special emphasis is put on the optimization potential within MPI.

Fast execution of an application is of main interest to the user since scientific goals should be achieved in time. To design fast programs, a user must understand the run-time behavior of the program on the system on which the program will be executed. Nowadays, users typically measure run-time behavior of their application, locate the bottleneck and try to improve performance in these long-running and thus critical code sections. There are also software-engineering concepts that can be applied during the whole software development cycle that take performance factors into account. Methodologies that lead to improved run-times in the implemented application are discussed in Section 2.4.

In Section 2.5 background about creation and verification of a model for a system is provided. The concept of simulation is introduced, too. A simulator implements a model and allows analyzing its behavior in silico. Further, several simulation tools that ease the implementation of models are introduced. While this section is not related to HPC, it provides relevant background to the art of simulation.

At last, in Section 2.6 a couple of simulators related to the simulation of cluster and I/O systems are presented. None of them allow to simulate parallel I/O and MPI at the level of detail required for this thesis.

2.1. Parallel File Systems

In contrast to a distributed file system, a *parallel file system* is explicitly designed for achieving performant and concurrent access to files. Therefore, internally data of a file is physically scattered among a subset of the available servers and their I/O subsystems. This enables those servers to participate in one I/O operation thus bundling their hardware resources to achieve higher aggregated performance.

This section extends the description of storage in Chapter 1 in three directions. First, by introducing representative enterprise and parallel file systems, important fundamental concepts are described. This enables assessing alternative I/O architectures and client-server communication protocols. Therewith, it helps us to define the scope of the architectural model and communication protocol that should be implemented in the simulation; the model should be flexible enough to represent several file systems. With the *Parallel Virtual File System* (PVFS) the architecture of one representative parallel file system is discussed in detail. Then, the I/O path of PVFS is described. This archetypal path is general enough to represent communication optimization strategies available in parallel file systems. With this knowledge, the I/O

[1] Note that a few passages are based on descriptions given in the author's master's thesis [Kun07].

[2] It is my personal opinion that all mentioned third-party software and published paper deserve a tribute because of the time spent by the authors to provide helpful tools and since they drive computer science forward. It is also my belief that software is never static and must be maintained to deal with new user and platform requirements. Such a dynamic software is probably never bug-free, and at any time some desirable features are missing. If I argue about missing features or suboptimal solutions in this chapter, then it is done in order to distinguish this thesis from existing approaches – all of the references deserve to be honored. It is not my intention to highlight or to criticize any of the papers or software mentioned. And I hope I succeeded in honoring work cited in an objective and constructive way.

path used in the simulator can be assessed better. Furthermore, it supports performance considerations which are discussed in the next section.

But first, a few basic terms:

Hierarchical namespace For convenient access of data, sequences of bytes are organized in file systems. The namespace is a concept that describes the organization scheme. Traditionally, file systems organize data in logical[3] objects: files and directories[4]. Files contain raw data. Internally, a *file* is like an array of bytes that can grow or shrink at the end. Thus, data must be serialized into a sequence of bytes to be stored. Directories structure the namespace by allowing to put other file system objects into them and give them a name, which results in a hierarchy of "labeled" objects.

This common organization scheme for file system objects is called *hierarchical namespace*. An example namespace is provided in Figure 1.3. By knowing the absolute path name within the hierarchical namespace, a logical file is unambiguously identified. Specific bytes of data can be addressed by referring to the offset within the "array" of data and the number of bytes (the size) to read or write . This type of addressing is referred to as *file-level* interface. However, the interpretation of the accessed bytes must be known to the program accessing it. For convenient access, a hierarchical namespace supports typically alternative names for individual objects, i.e., a single object can be found under different absolute file names. This is achieved by storing a reference to the original data (or directory) under an alternative name – this reference is called link. With a *global namespace*, the file system hierarchy can be accessed from multiple components in a distributed environment.

Beside the hierarchical namespace, there are other data access paradigms: In the cloud storage provided by *Amazon Simple Storage Service* (S3), data is referenced by a bucket (which can be thought of as a folder) and by using a key (similar to a file name). Arbitrary information can be stored for a given key. With the *Structured Query Language* (SQL), databases offer a high-level approach to access and to manipulate structured data. In contrast to a file-level interface data is managed on a higher level of abstraction: stored data is structured, every element of the structure has a label and datatype. Also, with SQL a user specifies the logic of the operations to perform with data and not the control flow that defines execution[5].

Client and Server Applications (and their processes) that access objects of a distributed file system are referred to as *clients*. In the context of hardware, the term describes a node hosting at least one process accessing the distributed file system. In this sense a node that provides parts of the parallel file system to a client is referred to as *server*.

(Meta)data Data refers to the raw content of a file. *Metadata* refers to the information about files and other file system objects themselves – the organization of objects in the namespace and their attributes. Attributes like timestamps or access permissions describe file system objects further. Usually, data and metadata are treated differently within the file system because of the semantic difference and the amount of stored information.

Block storage File systems use *block storage* to persist data. Block storage offers a block-level interface: Storage space is partitioned into an array of blocks that can be read or written individually. Access must be performed with a granularity of full blocks (typically 512 byte or 4 KiB), that means no block can be accessed partially. A number in a linear space specifies which block to access – this scheme is called *logical block addressing* (LBA). The relation between the blocks a file system object is made of is not defined on the block-level and must be managed by the file system.

[3] The term logical refers to the fact that this kind of object is accessed by using the file system interface. Internally, the file system might use several objects that work together to look like the logical object (e.g., file) to the user.

[4] Directories are also referred to as folders.

[5] Although there are procedural extensions, SQL is a declarative programming language.

A block device is a single hardware component that offers such a block-level interface. Multiple devices can be combined into a larger block storage that looks like a single block device.

2.1.1. Capabilities of Parallel File Systems

There are several requirements for file systems: persistence, consistence, performance, and manageability, just to name a few. *Persistence* describes that stored data can be accessed any time later. Also, the namespace should be in a correct state and all data read should also match the data that has been stored (*consistence*). Both requirements are vital because production of data is costly, and reading of corrupted (wrong) data can be disastrous. *Performance* describes the requirement that the file system should be able to utilize the underlying block storage efficiently[6]. Tools must be supported to mount the file system, to check the correctness or to repair a broken file system.

Additionally, for parallel file systems scalability, fault-tolerance and availability are of interest. *Scalability* of the file system allows deployment of the file system to provide sufficient performance for arbitrary workloads and to operate with any number of clients; performance just depends on the provided hardware resources and is not limited by the software. A *fault-tolerant* file system can tolerate transient and persistent errors to a certain extent without corrupting data, although the file system might be unavailable while errors are being corrected and it might crash when an error occurs. *Availability* describes a file system that continues to operate in the presence of hardware and software errors (usually with degraded performance).

The concepts of scalability and fault-tolerance will be briefly illustrated. In the context of this thesis other features are not relevant and therefore subsequently disregarded.

Scaling performance and capacity The architecture of most existing file systems and appliances can handle the demand of any customer ranging from small to very big (and fast) systems – these architectures are considered to be *scalable*. Furthermore, the requirements for an installed storage system might change over time. Since a storage system is costly and management of multiple systems is difficult, it is also important that the existing system can *scale* with increasing storage demands, either in performance or capacity. Otherwise, the money invested will be lost. Therefore, major enterprise file system vendors offer seamless upgrades of already installed solutions.

In the example storage system depicted in Figure 2.1 – it can be imagined that the single NAS server may be a bottleneck. With a scalable architecture, the storage system could be upgraded easily, with minimal modifications to the existing infrastructure; in the example additional NAS servers could be integrated.

There are two orthogonal principles to extend the capabilities of an existing system: *scale out* and *scale up*. The term *scale out* (or *scale horizontally*) is used to advertise modular systems which allow adding more storage nodes as they are needed – additional storage nodes add further capacity and performance at the same time. In contrast *scale up* (or *scale vertically*) means to equip the existing infrastructure with faster components; for example, by replacing a complete node or a single hardware component of the server. While upscaling is limited by the capability of available hardware technology, a scale-out solution provides high extensionability by adding more components. Both principles also apply to cluster computers.

To enable scaling, vendors typically put a distributed file system on top of storage nodes that aggregates all resources into a global namespace; data of a single file is distributed across all nodes. Thus, with an increasing number of storage nodes, the available capacity and performance can be scaled horizontally to any demand.

[6] Performance aspects of file systems are discussed explicitly in section 2.2.

Fault-tolerance Fault-tolerance as a requirement for high availability in parallel file systems is usually provided by replicating data over multiple devices in such a manner that permanent failure of a single component does not corrupt the file system integrity. Keeping a complete copy of data (mirroring) is costly as twice as much space is required. Therefore, mainly error-correcting codes such as the *RAID* levels 1 to 6, or *Reed-Solomon* codes are implemented.

Traditionally, error-correcting codes are applied on the block-level, in most cases multiple block devices are combined into a *redundant array of independent disks* (RAID). The RAID looks like a regular block device to the file system. If an error occurs and a block device of the array must be replaced, then the data of the broken device must be reconstructed by reading data from all disks and writing the lost information to the new block device.

File-level RAID is a software concept in which the file system controls the redundancy explicitly – for every file, the locations of the file blocks including the redundant data blocks are known. In contrast to block-level RAID, this has the advantage that hardware problems in the system only trigger a rebuild of the currently used space. Thus, empty (not-allocated) space of the system is not rebuilt. Failures that happen during the rebuild of a file, do not invalidate the whole RAID and thus file system – instead, the broken file can be identified and reported.

2.1.2. State of the Art

This section describes some parallel file systems that are deployed in enterprises and in a HPC center. This will allow us later to design an abstract communication protocol to simulate their interactions. Since performance of a file system depends on the communication path between client and servers, the focus of this section lies on the I/O path.

Enterprise storage In enterprise business, often a complete storage "solution" or *appliance* is purchased that is a bundle of hardware and software which can be integrated seamlessly into the existing communication network. Several commercial vendors for enterprise and datacenter storage sell *Network Attached Storage* (NAS) systems. Well-known vendors of network storage are: Panasas [NSM04], Netapp, Xyratex, BlueArc and Isilon [Kir10].

A customer connects the purchased storage system to the existing network infrastructure and accesses storage on the new system via standardized remote network protocols like the *Network File System* (NFS) or the *Common Internet File System* (CIFS)[7].

With NFS and CIFS, a remote node or workstation connects to exactly one file server (this scheme is illustrated in Figure 2.1). Clients forward file system operations to this server which executes them on a local file system. The block storage persisting this file system can be either integrated into the server, attached to it or provided in a *storage area network* (SAN)[8]. Recently, distributed file systems rely on *Object-based Storage Devices* (OSD) [Pan04] as underlying storage. Compared to low-level block devices, object-based storage provides a higher level of abstraction – access is performed by addressing file system objects directly. See Figure 1.7 and the descriptions on Page 9 for an example data access on a file system and the underlying block storage.

Panasas ActiveStor With *ActiveStor* Panasas delivers a performant and extensible system. The customer buys blade enclosures, which are equipped with a number of metadata servers (so-called director blades) and/or storage servers (so-called storage blades). Internally, the parallel file system *PanFS* distributes data across the available hardware. Metadata operations are performed on director blades, while data is stored on storage blades. The storage grows by adding further metadata and storage server blades or new

[7] An advantage of using those protocols is that they are available for most operating systems.

[8] A SAN is an additional network for block oriented storage. Devices that is part of a SAN, is usually addressed with the SCSI protocol. A single storage can be utilized from multiple servers, although not concurrently.

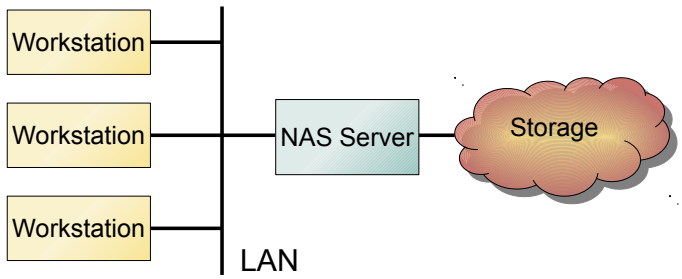
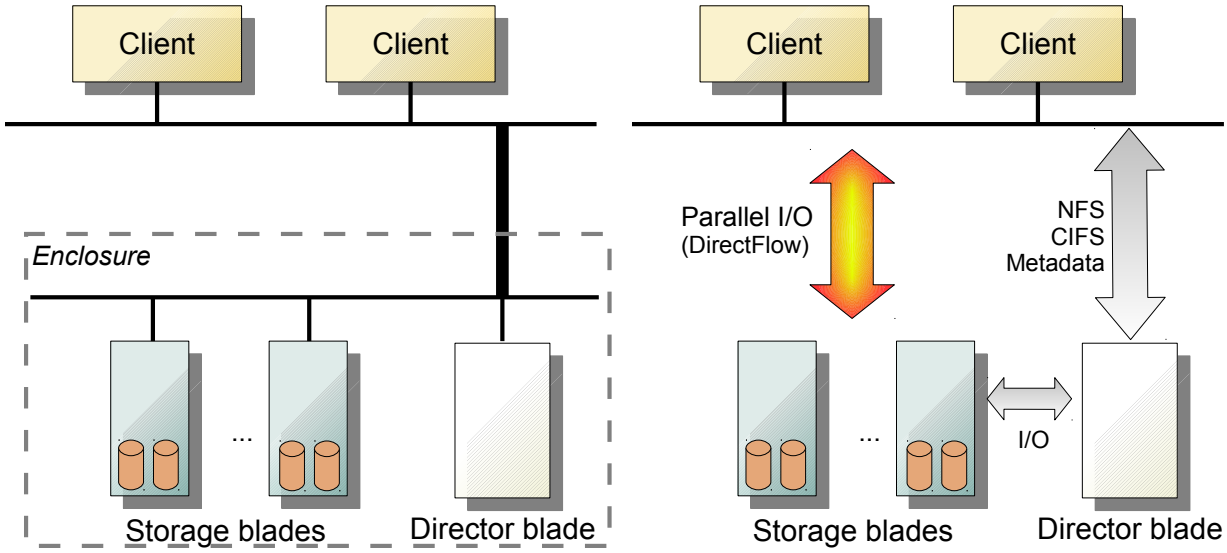


Figure 2.1.: Logical view of network attached storage. Multiple clients can access a central server that manages the persistent storage.



(a) Physical view – an enclosure holds 11 independent blade servers interconnected via a fast network link to the LAN. This example depicts a situation with a single director blade.

(b) Logical view. Clients communicate via a director blade when they use NFS or CIFS. By utilizing DirectFlow, they can communicate with the storage blades directly.

Figure 2.2.: Panasas ActiveStor system.

enclosures. A logical and physical view of an example system consisting of one enclosure is shown in Figure 2.2.

Enclosures have a fast interconnection to the *clients*, for example via 10 gigabit Ethernet (10 GbE). Internally, the enclosure is interconnected with each blade system by a slower 1 gigabit Ethernet – the name corresponds to the nominal data rate, for example, 10 GbE supports a rate of 10 GBit/s. Each blade holds two disk drives (or SSDs) to persist the file system.

Proprietary software can be installed on a client to allow it to interact directly with PanFS. In essence, the so-called DirectFLOW communication protocol implements a variant of the parallel NFS (PNFS) [HH07][9]; block I/O is performed with iSCSI, while an OSD protocol addresses the file system objects.

A client can also access storage via the NFS or CIFS protocol. In this case, clients simply mount the file system on one of the director blades with the one of this protocols. However, all I/O is communicated via the selected director blade, which forwards the I/O to the storage blades. Internally, the director blade translates the client operations into file system calls, and thus accesses PanFS (basically like a client which supports DirectFLOW).

[9] In fact, Panasas are member of the consortium for PNFS and share their file system knowledge to establish an industrial standard that might replace DirectFLOW in the future.

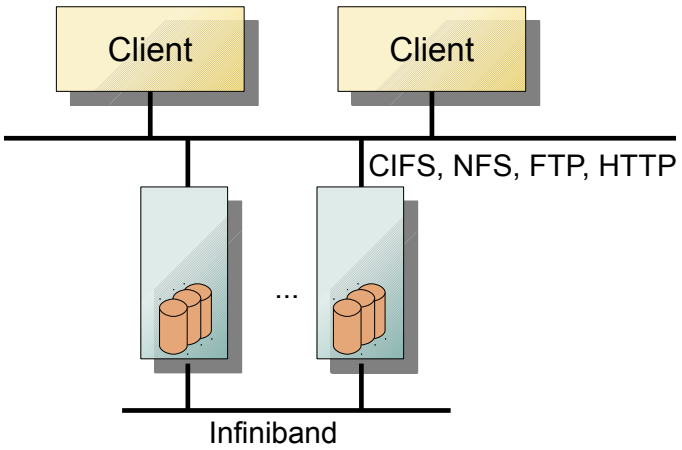


Figure 2.3.: Abstract view on an Isilon scale-out solution.

Isilon S-Series Isilon offers a NAS storage system in which the storage nodes have an additional Infiniband interconnect as a backend that balances load within the storage cluster (the *S-Series* [Har09]). Clients transfer data to one storage node by using either the NFS, CIFS, HTTP or FTP protocol. Clients and servers are interconnected with either Ethernet or Infiniband technology. The parallel file system *OneFS* orchestrates the servers into a coherent global namespace: Data is distributed across all servers of the storage system, the servers exchange data by using the internal Infiniband network. A schematic view is provided in Figure 2.3.

By supporting protocols in which clients connect to only one server, the server might become a bottleneck. To allow coarse grained load balancing, the assigned storage node is rotated by using *DNS round-robin*, that is, each DNS request is assigned to a different storage node. Each node hosts up to 36 storage devices and the scale-out system is expandable up to 10 PByte.

Both Isilon and Panasas support RAID protection on file-level.

Parallel file systems in HPC For HPC it is important that all processes of a parallel program can access the file system efficiently. A storage solution of the enterprise market might not match the architectures of supercomputers well, and thus may not provide the required performance. Performant concurrent access of multiple processes to a single file is especially difficult. For those reasons, only a subset of the existing enterprise solutions can handle HPC workloads.

The parallel file systems Lustre [SM08, YV07] and IBM's General Parallel File System (GPFS) [SH02, BICG08] are widely deployed in HPC environments – currently, 15 out of the 30 fastest systems use Lustre. GPFS is installed on many lower ranked systems of the Top500 list, too. Besides those file systems, mostly enterprise systems serve the storage needs of smaller supercomputers.

Xyratex offers Lustre based storage servers. In contrast to solutions mentioned from other vendors, this enables truly parallel access without modification of the client, i.e., the standard Lustre client can be installed. In the long term perspective, the parallel NFS (PNFS) will become widely supported, thus, allowing parallel I/O across I/O systems from various vendors.

The Parallel Virtual File System (PVFS) is an open source file system developed for efficient reading and writing of large amounts of data. There are many file systems designed with similar principles as PVFS, therefore, PVFS introduced as an archetype for parallel file systems.

2.1.3. The Parallel Virtual File System PVFS

PVFS is designed as a client-server architecture – servers provide storage and the client contains the logic to access this distributed storage. Multiple file systems can be served by one server infrastructure. According

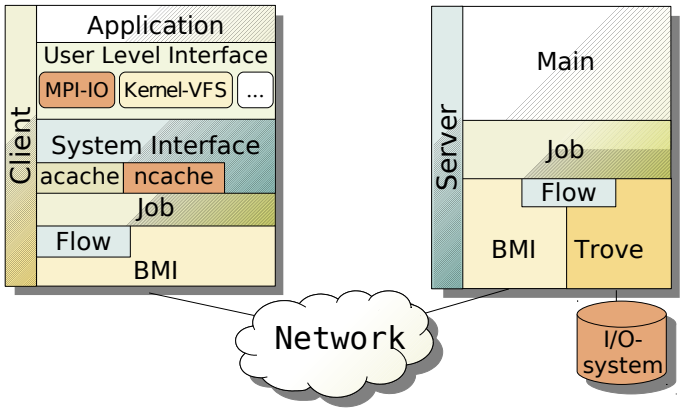


Figure 2.4.: PVFS software architecture.

to the type of storage provided, servers can be categorized into data servers and metadata servers. Data servers maintain parts of logical files. Metadata servers store attributes of logical file system objects – their metadata, and the namespace. A server operates either as metadata server or data server, or both at once.

Logical file system objects which can be stored in PVFS are files, directories and symbolic links. Internally, additional system-level objects exist: *metafiles* contain metadata for a file system object, *datafiles* contain pieces of the file data and *directory data* objects map names of logical objects to internal objects. A logical file is represented as a metafile which refers to several datafiles – file data is partitioned among the datafiles which are then placed on the available data servers. In PVFS neither file data nor metadata is replicated explicitly. Instead, configurations aiming for high availability rely on shared storage, i.e., storage that is accessible from multiple servers over a network – SAN storage for example.

Architecture PVFS uses the layered architecture illustrated in Figure 2.4. One of the main advantages of a layered architecture is that an implementation for a layer can be replaced, in order to match the underlying hardware.

The *user-level interface* allows access to a file system with a standardized interface – PVFS provides interfaces for the Linux kernel, the *Filesystem in Userspace* (FUSE)[10], and MPI. The kernel interface integrates PVFS into the kernel's Virtual File System Switch (VFS), a user-space daemon connects to the kernel module via the `/proc` interface and communicates with the servers.

The *system interface* enables direct manipulation of file system objects, yet some internal details are hidden from the user. Internally, the processing of file system operations is modeled with finite state machines – states can fetch information from a server, or send a request to a server to modify the file system.

Several *client-side caches* are incorporated to reduce the number of requests to the servers. An attribute cache (acache) maintains metadata, the name cache (ncache) buffers mappings from file names to internal objects.

The *job layer* is a thin layer consolidating all lower interfaces – BMI, Flow and Trove, into one interface.

The *Buffered Message Interface (BMI)* provides a network independent interface for message exchange between two nodes. Clients communicate with the servers by using the so-called *request protocol*, which defines the layout of the messages for every file system operation. Depending on the underlying network technology like TCP or Infiniband, the appropriate communication method can be selected.

Trove interfaces with the storage subsystems to store the system level objects. Data can be either stored as a key/value pair or as a bytestream. Key/value pairs are used to store metadata, while byte streams keep the file data. By default, PVFS uses multiple Berkeley databases to store the metadata and regular UNIX files to store bytestreams.

[10] `http://fuse.sourceforge.net/`

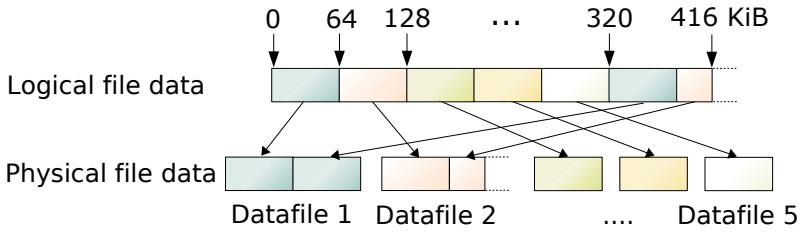


Figure 2.5.: Exemplary file distribution for 5 datafiles – data is split in 64 KiB chunks and striped over the datafiles in a round-robin fashion.

The *I/O subsystem* indicated in the figure can be any block storage – disks integrated in a SAN or direct attached storage, usually RAID systems. When a SAN is deployed, PVFS can be configured for high availability, i.e., to tolerate server crashes.

A server's *main loop* drives the processing of communication and I/O layers. Also, the server checks for new requests and initiates state machines to process them.

On the server side, it is important to overlap I/O with communication, the orchestration of data streams between two endpoints is performed by *Flow*. Data might flow between memory, BMI, or Trove. Data on the server flows between network and I/O subsystem, while on the client side it is transferred between memory and the network. The flow protocol defines how data is transferred between two endpoints including the caching strategy. By default, PVFS allocates 8 data buffers for every I/O request, a buffer is either accessed by the network or the storage layer. That is, up to 8 operations can be pending on Trove or BMI – once all buffers are filled due to a congestion on I/O subsystem or network, further operations are deferred.

Data distribution of a logical file A selectable *distribution function* defines how data of a file is distributed among the different datafiles (often also referred to as *stripes*). By default, file data is split in 64 KiB chunks which are distributed over all file servers. Similar to a RAID-0, the first chunk is mapped into the first datafile, the second chunk into the second datafile, until one chunk has been mapped to each datafile. Then, the next chunk is assigned to the first stripe again, until all chunks are assigned. The mapping of a logical file with a size of 416 KiB into 5 datafiles is illustrated in Figure 2.5. A datafile is mapped to exactly one server; this way file data is distributed among the available servers.

When a user creates a new logical file in PVFS, the datafiles are created and assigned to the file servers. Actually, PVFS tries to pre-create logical files and keeps them in a pool to avoid the overhead of creating and mapping of datafiles.

Small files might never use all datafiles; thus, the empty datafiles cause overhead during file access and they crowd the underlying local file system. Therefore, some parallel file systems like PanFS from Panasas grow the number of stripes dynamically. In this concept, small files are first stored within the metadata, then the number of stripes grows with the file size – while appending new data, existing data could be redistributed among the servers to re-balance it optimally, or just be kept in place.

2.1.4. Client-Server Communication

Due to the distributed nature of the clients and servers, a protocol defines how the clients interact with the servers to access and manipulate the file system. In contrast to Isilon's OneFS, with PVFS (or GPFS) a server accesses only its own storage and thus, knows local objects. Therefore, no server has an overview of the complete file system. To perform an operation, a client gathers intelligence where metadata and data reside and communicates with the required servers. A slightly simplified client-server protocol of PVFS is discussed here, but it is close to the request protocol actually used.