**DISSERTATION**

# Design and Implementation of an autonomous, distributed RAID System with a XML Meta Definition language

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter der Leitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Grechenig
E183
Institut für Rechnergestützte Automation

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

**Mag. Thomas Fürle**
Matrikelnummer 8926464
Löhrgasse 3/5
A-1150 Wien, Österreich

Wien, im Mai 2005

[illegible]

# Kurzfassung

Paralleles und verteiltes I/O ist zur Zeit ein hochaktuelles Thema in der Wissenschaft. Bis jetzt wurden meistens proprietäre (herstellerbasierende) oder UNIX Techniken verwendet. Nun gibt es Anstrengungen für einen allgemeinen Ansatz für paralleles I/O im MPI/2 Entwurf, Kapitel 9. Dieser neuer, einheitlicher Ansatz erfordert vom Benutzer ein umfangreiches Wissen über die Zugriffsmuster.

Das Ziel dieser Arbeit ist die Präsentation von ADIOS, einer "client server" basierenden Implementation für verteiltes I/O mit Schnittstellen zu MPI/IO oder herkömmlichen UNIX System Aufrufen. Die Betonung liegt auf den Design Ansätzen der Kern Systems.

Wir starten mit einer einleitenden Darstellung von parallel I/O, File Systemen, für Cluster verfügbare parallele und verteilte Systeme, setzen dann fort mit einer kurzen Beschreibung und Präsentation von ADIOS, seinen Ähnlichkeiten und Verschiedenheiten verglichen zu ROMIO, einer weit verbreiteten Implementation von MPI/IO und vergleichen auch mit PVFS. Dann konzentrieren wir auf das Hauptkapitel, der umfangreichen Darstellung des systemübergreifenden Designs. Den Abschluss bilden Leistungstests im Vergleich zu ROMIO und PVFS.

# Abstract

Currently parallel and distributed I/O is a hot topic in science. Until now most of the times proprietary (vendor) techniques or the Unix like semantics were used. Now there some efforts to generalize parallel I/O in the MPI/2 draft, chapter 9. This new generalized techniques assume a comprehensive knowledge of the access patterns by the users.

The aim of this work is to present ADIOS, a client server implementation of distributed I/O with interfaces to MPI/IO and native UNIX system calls. We want to emphasize on the overall design principles of the core system.

We will start with a introduction to parallel I/O, filesystems, available parallel and distributed I/O systems for clusters, continue with a short description and presentation of ADIOS, its similarities and its differences compared to ROMIO , a well known implementation of MPI/IO and also PVFS. Then we will focus on the main topic, the overall system design, which will be very comprehensive and finally we will close with some performance tests we ran compared to ROMIO and PVFS.

# Table of Contents

# Chapter 1

# Introduction

## 1.1 Motivation

*Perfection is not achieved when there is nothing left to add, but when there is nothing left to take away*

In the last few years grid computing became very popular. Approaches like Globus[7], NetSolve, SETI@home attracted more and more people to join. The basic idea behind these projects is to solve large problems by harnessing the CPU cycles of participating machines over the internet. This approach is followed in the small by so called Beowulf cluster type systems [65]. Off-the-shelf workstations are connected by an affordable network interconnect (Fast-Ethernet, Giganet), and suitable operating and programming environments allow to exploit the cumulative processing power to solve grand challenging problems. Due to their low price (compared to the classic supercomputers) these clusters became very popular and representatives can now even be found in the list of the 500 worlds most powerful computer systems (`http://www.top500.org`).

Parallel to this development applications in high performance computing shifted from being CPU-bound to be I/O bound. That means that performance cannot be scaled up by increasing the number of CPUs any more, but by increasing the bandwidth of the I/O subsystem. This situation is known as the I/O bottleneck in high performance computing.

Besides the cumulative processing power, a cluster system provides a large data storage capacity as well. Usually each workstation has at least one attached disk, which is accessible to the system. Using the network interconnect of the cluster, these disks build a huge common storage resource.

This situation stimulated the development of the Autonomous, Distributed Input Output System (ADIOS), which represents a fully-fledged distributed I/O runtime system focusing on workstation cluster systems. It is available as a I/O server configuration; it supports the standardized MPI-IO[54] interface and UNIX semantics.

## 1.2 Scope of this Work

This thesis is focused on presenting ADIOS, a novel approach for autonomous, distributed I/O. We have designed this system from scratch and we will test its performance comparing it to the state of art in parallel and distributed I/O (e.g. ROMIO, PVFS, UNIX semantics). We also want to concentrate on implementation issues when using PVM, MPI and pthreads for designing, implementing and analyzing the overall system. Finally we want to prove our concept with practical tests from different applications and give some hints for using ADIOS in a practical environment like Beowulfs.

The dynamic reorganization of data on disk (the fragmenter tier in ADIOS) via the usage of hints, is falling out of scope of this thesis.

## 1.3 Aims

- The design of the ADIOS system
  - present the overall ADIOS system design, explain the major components roughly, compare ADIOS to already existing systems
- The implementation of the ADIOS system components
  - present the ADIOS components in detail, also give some details about implementation issues
- implementation specific issues regarding to PVM, MPI and pthreads
  - discuss problems, solutions workaround for the implementation of the working prototype of ADIOS with PVM, MPI and pthreads
- performance of ADIOS compared to Unix semantics, PVFS
  - test the performance of ADIOS compared to other systems
- analysis of performance issues of ADIOS compared to Unix semantics, PVFS
  - analysis of the above performance tests and conclusions for further directions

## 1.4 Thesis structure

This thesis is organized in chapters as follows:

- Chapter 2 gives a general overview of the target environment of this work, which are mainly Beowulf Clusters. We present this kind of machine, operating systems and general assumptions used in this work. We also present the state of the art in the file system field. The last point are available parallel and distributed I/O Systems for Beowulf Clusters
- Chapter 3 describes the overall design of ADIOS, its different operating and implementation modes. We will go through all designed modules and describe especially those, which are just referenced or used in the remainder of this work.
- Chapter 4 describes the components and their respective interfaces of ADIOS in detail
- Chapter 5 presents implementation issues which arise, when implementing ADIOS in an island environment
- Chapter 6 describes all external interfaces of ADIOS
- Chapter 7 tests the performance of ADIOS compared to other systems like PVFS
- Chapter 8 concludes and give analysis of the overall system and some possible future directions in this area

# Chapter 2

# State of the Art in Parallel/Distributed I/O

## 2.1 Introduction

I/O has long been the "poor stepchild" of scientific computing, especially high performance computing. Indeed the very fact that we call it computing, rather than data management or manipulation reflects that bias and vocabulary. We speak of central processing units and primary memory, but of peripherals, secondary and even tertiary storage. In scientific computing, we focus on FLOPS (floating point operations/second) and eagerly rate computers in gigaflops, teraflops, and someday soon, petaflops, but all too rarely do we discuss petabytes stored or terabytes/second transfered.

This compute-centric view beliefs the fact that scientific computing is increasingly about intelligent data management. Faster and more powerful computer systems, along with the emergence of computational science as true third member of the theory, experiment, and computational simulation triumvirate, mean the extracting meaning from data, both experimental and computer-generated, is central to scientific discovery.

Terascale simulation can produce enormous amounts of data, and gaining insights from that data via visualization or intelligent data reduction requires I/O hardware and software that can move data rapidly across networks and to and from arrays of storage devices. Equally importantly, and oft overlooked, an new generation of high-resolution scientific instruments (e.g. CERN) is coming online, and these instruments, ranging from large radio telescopes to advanced particle detectors, also produce large volumes of data in real time.

How can all these programs harness those storage systems efficiently ? This leads to related questions and has motivated the development of ADIOS

- How do file systems manage files ?
- What are the input and output (I/O) patterns of large scientific applications
- How can I/O requests be made to execute more efficiently ?
- What programming interfaces are available to read and write data, and how should they be used ?

## 2.2 Understanding the Levels of I/O

Ideally, computer architects and system programmers could improve I/O performance without forcing application developers to change their programs. However, tuning the overall performance of an application requires an understanding of all the major parts of computer's architecture, including not only the CPU, cache, and memory, but also the I/O system. It includes storage devices, interconnection networks, file systems, and one or more I/O programming libraries.

Perhaps the most important development in the design of storage devices has been the advent of RAID technology, which combines many small disk drives to form, large, high performance storage devices. The combination of multiple disk drives with RAID software and hardware simultaneously increases capacity, improves data transfer rates, and makes the system more reliable. One of the basis ideas of ADIOS is to take the definition of RAID and extend it to a combination of multiple machines and even further to a group of multiple machines (see ADIOS Islands 5.2), but to do this on the file system level (see next paragraph).

The capabilities and programming interfaces of storage devices are quite primitive. *File systems*, the software, that creates these familiar abstractions and make storage devices usable to higher-level applications. File systems are also responsible for maintaining the integrity of stored data and for trying to hide some of the performance quirks of disk drives.

Parallel file systems do all these tasks for multiple processors and multiple independent storage devices. Parallel file systems are not the same as distributed file systems, such as NFS and DFS. While both types of file systems support access to shared files from multiple processes, parallel file systems must efficiently support *simultaneous* access to individual files. Distributed file systems are not designed to support this kind of fine-grained file sharing efficiently. A key challenge for parallel file systems is maintaining data integrity during the parallel access without compromising performance. Computer vendors and research groups have developed a number of parallel file systems over the years that attack these problems. Some systems use novel programming interface that let the user customize the system's behavior to improve performance for specific kinds of data access patterns. Others present a standard interface similar to UNIX I/O and try to offer good performance over a wide range of access patterns. Research file systems have focused on novel programming interfaces or novel architectures. The growing popularity of parallel computers built from clusters of workstations has presented a number of challenges to file systems designers, which several research groups have tried to address with their file systems. ADIOS tries to satisfy the needs of all of this worlds with a best effort approach supporting e.g UNIX I/O as well as novel programming interface like distributed UNIX I/O with an underlying XML Meta language.

Despite the best effort of file system designers, many parallel scientific applications exhibit very poor I/O performance for certain access patterns. Unlike codes on vector computers with a single pool of memory, the parallel codes often write data in small, discontiguous pieces, which are difficult for file systems and storage devices to handle efficiently. To remedy this problem, a number of research groups have developed techniques for collecting small pieces of data into larger units that lower levels of storage hierarchy can manage better. These techniques are called collective I/O, and a number of variations have been proposed and implemented. The most important are

- two phase I/O
- data sieving
- collective I/O
- disk directed I/O
- server directed I/O

These methods try to execute I/O in a manner that minimizes or strongly reduces the effects of disk latency by avoiding non contiguous disk accesses and thereby speeding up the I/O process. More details and even some more techniques can be found in Dictionary on Parallel Input/Output [66].

Another important I/O optimization is the use of *hints*, which let applications tell the I/O system about upcoming I/O access patterns. The I/O system can use this information to select optimization parameters that are likely to yield good performance. A step beyond the use of hints is to analyze the I/O access patterns of a running application and select optimal parameter settings automatically. We pursuit the idea of hints in ADIOS with the help of the compiler and a blackboard approach for a running system, but this topic is falling out of the scope of this thesis.

Both collective I/O and hints require more information from an application than they through a standard sequential I/O programming interface.

Parallel I/O interfaces are more expressive than the standard sequential Unix I/O interface. They can describe I/O operations that are coordinated across multiple processes, and the operations can involve many separate pieces of data. Many interfaces also let applications pass in hints or change some of the I/O system's configuration parameters. Scientific work concentrates on MPI-IO, the I/O interface defined in the MPI-2 message passing standard. This interfaces uses concepts and data structures from MPI (message passing interface) to describe parallel I/O operations, and it has been implemented on many parallel systems.

## 2.3 Runtime I/O libraries

are highly merged with the language system by providing a call library for efficient parallel disk accesses. The aim is that it adapts graciously to the requirements of the problem characteristics specified in the application program. Typical representatives are PASSION [69], Galley [58], or the MPI-IO initiative, which proposes a parallel file interface for the Message Passing Interface (MPI) standard [56, 14]. Recently the MPI-I/O standard has been widely accepted as a programmers interface to parallel I/O. A portable implementation of this standard is the ROMIO library [71].

Runtime libraries aim for to be tools for the application programmer. Therefore the executing application can hardly react dynamically to changing system situations (e.g. number of available disks or processors) or problem characteristics (e.g. data reorganization), because the data access decisions were made during the programming and not during the execution phase.

Another point which has to be taken into account is the often arising problem that the CPU of a node has to accomplish both the application processing and the I/O requests of the application. Due to a missing dedicated I/O server the application, linked with the runtime library, has to perform the I/O requests as well. It is often very difficult for the programmer to exploit the inherent pipelined parallelism between pure processing and disk accesses by interleaving them.

All these problems can be limiting factors for the I/O bandwidth. Thus optimal performance is nearly impossible to reach by the usage of runtime libraries.

## 2.4 File Systems

Most storage devices have no notion of files, directories, or the other familiar abstractions of data storage, they simply store and retrieve blocks of data. A *file system* is the software that creates this abstractions, including not only files and directories but also access permissions, file pointers, file descriptors, and so on.

File systems have other duties as well:

- Moving data efficiently between memory and storage devices
- Coordinating concurrent access by multiple processes to the same file
- Allocating data blocks on storage devices to specific files, and reclaiming those blocks when file are deleted

All modern file systems handle, these task, whether they run on parallel or sequential computers. A parallel file system is especially concerned with efficient data transfer and coordinating concurrent file access.

To see how file systems work, let's review the standard characteristics that are presented to programs and users.

### 2.4.1 UNIX File Model

1. **Sequential consistency**

   Multiple processes can access the same file at the same time. As long as the accesses are all read operations, there will be no conflicts. However, if two or more processes write to the file at the same time, or if one process writes while another is reading, the file system has to guarantee that the results make sense. If the concurrent accesses involve different parts of the file, there is no problem. The difficulty arises if the accesses overlap. Consider the case of two processes writing different sequences of bytes to the same locations in the file at about the same time. What data will end up in the file ? Most Unix file systems guarantee *sequential consistency*; that is, they guarantee that the result will be as if the two write operations happened in a specific order. Either the first sequence of bytes will be written or the second sequence will be written (and it's unpredictable which sequence will prevail), but in no event will bytes from the first sequence be intermixed with bytes from the second sequence. It's relatively easy for a file system to make this guarantee when the conflicting accesses happen on a single (time-shared) CPU with a single disk drive, but in parallel and distributed file systems, sequential consistency is much harder to guarantee.

2. **File Buffering and Caching**

Obviously, most files don't fit exactly into a whole number of blocks, and most read and write requests from applications don't transfer data in block-sited units. File systems use buffers to insulate users from the requirement that disks move data in fixed-site blocks. Buffers also give the file systems several ways to optimize data access. File systems allocate their buffers in units the same size as a disk block. The most important benefit of buffers is that they allow the file system to collect full blocks of data in memory before moving it to the disk. If a file system needed to write less than a full block of data, it would have to perform an expensive read-modify-write operations. Write buffering improves performance even when an application writes a full block of data or more. For accesses about the size of a block, the file locations where the data will be written may cross over a block boundary, so a block-sized write may end up as two partial blocks. For larger writes, the file system can delay writing until it has several blocks to transfer. Disks can usually handle these multiblock transfers more efficiently than single-block transfers. The latter technique is sometimes called *delayed write or write behind.*

Similarly, when a file system reads data, it must retrieve a full block at a time. Even if the application program hasn't asked for all the data in the block, the file system will keep the entire block in memory, since the application may later request more data from the same block. This technique is called *file caching.* If a file system detects that an application is reading data sequentially from a file in small steps, it may use *prefetching* (also called *read ahead*) to improve performance further: the file system reads not only the block that contains requested data but also one or more subsequent blocks in the file. The extra cost of reading the additional blocks in a single request is usually less than the cost of reading two or more blocks separately. When the program requests data from the prefetched blocks, they will already be in memory, or at least on their way. Therefore, the file system can complete these subsequent reads more quickly than the initial request. Prefetching reduces the apparent data access time for a disk, since the cost of reading the second and subsequent blocks is hidden from the application. However, prefetching works poorly when an application's read requests don't follow a simple, predictable pattern. In that case, the file system may waste time prefetching blocks that the application doesn't need right away.

The file system uses the same pool of memory for both buffering and caching. This allow it to keep the data consistent when the application writes and the reads back the same file location. These accesses will be very efficient because neither request will require access to the disk. An application can create a file, store a small amount of data, read it back, and delete the file without ever accessing a disk. Because caching and buffering are closely connected, both techniques are often referred as *buffering.*

In some systems, all memory not being used by applications is allocated to the file buffer pool. Nevertheless, a file system has only a finite amount of buffer space. so it cannot keep data there indefinitely. When all the buffer blocks are in use and the file system needs a new block for a read or write request, it must reuse one of the buffer blocks currently in use. If the buffer to be reused contains data that was read from the disk, and the application hasn't written data back to that file block, then the file system can immediately use this buffer to carry out the new request. However, if the buffer contains data that the application has written but that the file system hasn't yet moved to disk, then the file system must *flush* this data to the disk before it can reuse the buffer. Buffers containing data that hasn't yet been written to disk are called *dirty.*

## 3. Nonblocking I/O

Caching and buffering improve performance in two ways: by avoiding repeated accesses to the same block on disk and by allowing the file system to smooth out bursty I/O behaviour. The smoothing happens because the application can quickly write a large amount of data into file system buffers without waiting for the data to be written to disk. The file system can write these blocks to disk at a slower, steady rate while the application continues with other work that doesn't require I/O. This delayed writing can make the file system's instantaneous transfer rate much higher than its sustained rate.

*Nonblocking I/O* gives a program control over prefetching and delayed writing. An application can issue a read request some time before it expects to need the data. Then instead of blocking the program until the data has arrived, the I/O function returns immediately, and the file system completes the request in the background while the application continues to work. When the application reaches a point where it needs the data, it can issue another request to check whether the data is available or to pause execution until the access is complete. Alternatively, some nonblocking I/O implementations can signal the application when the data arrives. An application can also issue a request to write data to a file and then continue computing while the file system moves the data from the user buffer to the disk. Since the application often knows sooner than the file system what data it will need, nonblocking I/O can be much more effective than prefetching. Also, an application can devote a specific memory buffer of exactly the right size to prefetched and delayed-write data, whereas the file system must share its buffers among all jobs and try to guess which disk blocks to keep and which to reuse.

## 2.5 Distributed File Systems

The file systems discussed so far are designed to run on a single CPU. Several processes may access a file concurrently, but the file system guarantees sequential consistency. It usually does this by preventing any processes from writing a file at the same time as another process is either reading or writing the file.

Distributed file systems are designed to let processes on multiple computers access a common set of files. Although distributed file systems have some features in common with parallel file systems (see next paragraph), they are not a complete solution for parallel I/O. In particular, as described below, distributed file systems are not designed to give multiple processes efficient, concurrent access to the same file. Nevertheless, distributed file systems are a good point from which to begin an examination of parallel file systems.

Probably the best know distributed file system is NFS (Network File System) [22], [43], which Sun Microsystems first released in 1985. NFS allows a computer to share a collection of its files with other computers on the network. The computer where the collection of files resides is called a server, and a computer that remotely accesses these files is a client. In NFS, a computer can be a server for some files an d a client for others. Clients "mount" a collection of files - a directory on the server and all its subdirectories - at a particular location in their own directory hierarchy. The remote files appear to be part of the client's directory hierarchy, an programs running on the client can access them using the standard UNIX naming conventions. When a client program reads a file that resides on the server, the client's file system sends a request to the server, which gets the file (or just a part of it) and sends it back to the client. The operation is invisible from the applications point of view, except that accessing a remote file takes longer than accessing a local one. Two other well-known distributed file systems are AFS and DFS. AFS [46] is based on the Andrew File System, first developed at Carnegie-Mellon University in the mid-1980s and later offered as a commercial product. DFS [48], [32] is the Distributed File System, a successor to AFS developed as part of the Open Software Foundation's Distributed Computing Environment. Like NFS, AFS and DFS allow multiple computers to access a collection of file over a network, but they have different architectures and features.

## 2.6 Parallel File Systems

A distributed file system does only part of what a parallel file system needs to do. Distributed file systems manage access to files from multiple processes, but they generally treat concurrent access as an unusual event, not a normal mode of operation. The design of a parallel file system must deal with several important questions:

- How can hundreds or thousands of processes access the same file concurrently and efficiently ?
- How should file pointers work ?
- Can the Unix sequential consistency semantics be preserved ?
- How should file blocks be cached and buffered

### 2.6.1 Intro: sequential access versus multiple file access

Even though parallel file system development is quite advanced, many parallel applications continue to use one of two alternative types of I/O: pure sequential access, in which program sends all of its file accesses through a single task, and multiple file access, in which each task writes its own file.

Sequential access in distributed memory computers does have two attractive features. First, all the data resides in one file, so it is easy to manage. In particular, the user can copy the file as a single unit to tertiary storage or to another computer. Second, sequential access is likely to produce contiguous file accesses that the storage devices and file system can handle efficiently. This advantage is clearest when the program accesses a modest amount of data. Reading or writing a block of data from a single process is often more efficient than having many separate processes access small amounts of data from the same file.

Sequential access also has some important drawbacks in distributed memory computers. A single process running on a single node may not have enough memory to hold all the data that the parallel job needs to read or write. To work around this problem, the program must access the file in several steps, separated by gather or scatter operations. More importantly, the total transfer rate is limited to what a single node can support. For moderately large parallel programs, sequential file access is to slow.

Multiple file access is an alternative to sending all the data to one process. For this technique, used mainly in message passing programs, each processor writes data to a separate file. If the file reside on the node's local disks, file access will be very fast because the data won't need to travel over the computer's message passing network, and the data transfer is perfectly parallel. If the files are temporary or if the user doesn't need a single combined data set, multiple file access is an excellent choice. However, many applications do need to produce a single data set. The postprocessing required to collect many separate files into one large file can easily wipe out the performance benefits of multiple file access. On systems where local disks are not accessible to other nodes, merging files will require another parallel program that runs on the same set of nodes as the program that generated the data.

Parallel file systems try to address the main drawbacks of both sequential access and multiple file access. They combine the high performance and scalability of multiple file access with the convenience of collecting data in single files. To do this, they must allow multiple tasks to access a file at the same time, though not all the tasks will necessarily access the same locations in a given file. ADIOS follows this approach.

### 2.6.2 Concurrent File Access

The first challenge for a parallel file system is to support concurrent access from several processes. Since parallel file systems usually stripe files over multiple disks (connected to different I/O nodes), the file system has to manage two separate data mappings; the mapping from multiple compute nodes to a shared file and the mapping of the shared file to multiple I/O nodes and storage devices. Figure 2.1 shows two examples of these mappings with four compute nodes and two I/O nodes.

The top row in each diagram shows four compute nodes, each with two blocks of data. (Numbers in all the blocks show which compute node produced the data.) The middle row of the diagrams show how these blocks fit into the logical structure of the file. The bottom rows show how the file is striped over the I/O nodes. In the top example, the distribution of data among the compute nodes matches the file's striping, so each compute node can send whole blocks of data to just one of the I/O nodes, which will be less efficient. Although the diagram shows the mapping from the compute nodes to the logical file layout on the I/O nodes, in reality the data moves directly between the compute nodes and I/O nodes; it never resides all in one place as shown in the middle rows.

To access a file in parallel, each process begins by opening the file. In a sequential

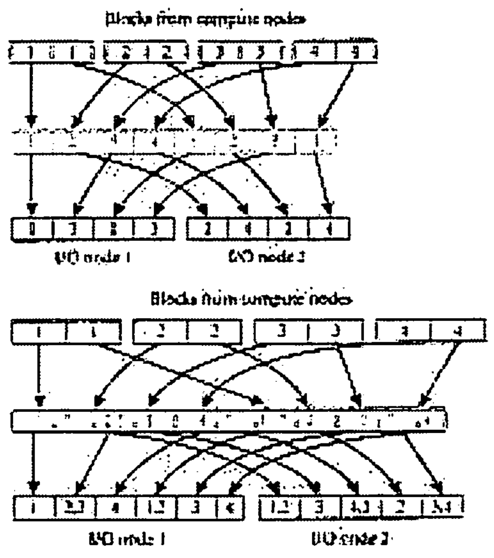

Figure 2.1: Mapping of blocks from compute nodes to logical files to I/O nodes

system, the file system translates the file name to an inode number. In parallel file systems, each I/O node manages a subset of the blocks that make up a file, so every file has an inode (or a similar data structure) on every I/O node. The file system needs a way to look up each of these inodes when it opens a file. It's possible to have each I/O node maintain its own directory information and lookup its own inodes (ADIOS uses this approach). Another solution is to use a central name server, a process that file system software on all the nodes can call to look up inode numbers using file names. This avoids the need to replicate the directory data on each I/O node. The name server typically resides on an I/O node.

Some file systems fix the strip factor and stripe depth when the system is configured; others like ADIOS allow users to specify these parameters separately for each file. When a file is first created, the file system chooses the I/O node that will store the first block of the file. Varying this location distributes the work among the I/O nodes; if the location were fixed, all short files would reside at the same I/O node. Subsequent blocks go to the other I/O nodes in a fixed or pseudorandom order; ADIOS furthermore supports user defined distributions.

Compute nodes can send requests to the I/O nodes in parallel, an of course the I/O node can carry out the requests in parallel, since they are accessing data on storage devices that they control exclusively. As long as processes on different compute nodes don't try to access the same part of the file, striping is easy to manage.

Problems arise when the system has to enforce *sequential consistency.* Suppose two processes write to the same range of locations in a file, and the range spawns two blocks on different I/O nodes. Sequential consistency requires that the two I/O nodes write their portions of the data in the same order, ensuring that the write requests *appear* to occur in a well-defined sequence. How will each I/O node know what the other is doing ? One solution is to use a locking mechanism on the file that prevents more than one process from writing a file at the same time. This solution prevents parallel file access, and it is essentially what sequential Unix does. The problem with locking the whole file is that it prevents parallel access even when the processes are not writing overlapping regions of a file. Most applications rarely write the same file location concurrently from separate processes, so maintaining the Unix model of sequential consistency using file locking needlessly ruins parallel performance for common access patterns. As a result, some parallel file systems offer the user a choice of access modes: one that guarantees sequential consistency at the expense of parallel performance and another that allow concurrent access by relaxing the consistency semantics. In the latter case (which is supported by ADIOS), the application is responsible for preventing different processes from writing data concurrently to the same location.

### 2.6.3 Buffering

Parallel file systems, like sequential ones, use caching and buffering to reduce the need for disk accesses. In systems with separate I/O and compute nodes, buffering can happen in both places. Buffering at the compute nodes is called *client buffering* and buffering at the I/O nodes is called *server buffering*. Some file systems use only client or server buffering; others use both.

File system with client buffering manage a pool of buffer space on each compute node. As in sequential file systems, data to be written to a file is copied from the user's address space into the buffer, and the file system might not send the buffer to the I/O node until some time after the write request appears to the application to have completed. Read requests copy data from I/O nodes into the buffer, and subsequent read requests from the same disk block can be satisfied without further communication with the I/O node.

The problem with client buffering is similar to the caching problem in distributed file systems. If a process writes data to a file, and the data remains in a buffer on the compute node for some time, a process on another compute node trying to read the same location in a file won't see the changes the first node made. This problem is more severe than the sequential consistency problem noted earlier because it can happen even if the two processes access the data in a well-defined order (i.e not concurrently). File systems use a variety of approaches to address the problem. Some offer a relaxed consistency model that requires the program to synchronize the file explicitly to make changes visible to all processes. Another solution is to implement a cache coherence protocol that allows processes to keep track of which nodes are reading and writing each location in a file. ADIOS doesn't need to cope with client buffering, because its on top of the underlying basic file system, so even in the case of client buffering it looks for all other ADIOS processes like server buffering, when the ADIOS processes communicate with each other (see more in the overview of ADIOS in the next chapter)

### 2.6.4 Commercial Parallel File systems

This section looks at some of the parallel file systems that computer vendors have developed to address the problems discussed so far in this chapter.

#### Intel PFS

Intel's PFS (Parallel File System [19]) was a significant early parallel file system. Intel developed PFS for its Paragon supercomputers. The company used an earlier file system called file system CFS (Concurrent File System) [59][55] in the Paragon's predecessor, the iPSC. The Paragon is a distributed memory parallel computer with I/O nodes that manage access to parallel files. The Paragon also supports NFS files and ordinary sequential Unix (called UFS files, for Unix File System). NFS and PFS subdirectories are typically mounted in a UFS directory hierarchy. PFS stripes files over multiple I/O nodes. The system administrator configures the striping parameters, and they apply to all the file that PFS manages in a hierarchy.

#### IBM Vesta, PIOFS and GPFS

IBM developed the Vesta parallel file system [15] [26] as a research project. A central feature of Vesta is that it abandons the Unix model of a file as linear sequence of bytes. When IBM turned Vesta into a commercial product, it renamed the system to PIOFS (Parallel I/O File System) [18]. The two file systems are similar, but not identical.

PIOFS was designed for the IBM SP series of parallel computers. These are distributed memory machines with separate I/O nodes. IBM has replaced PIOFS as its standard parallel file system with GPFS. Nevertheless, PIOFS is interesting because of the unique file model it defines to improve parallel I/O performance. GPFS (General Parallel File System) [45] is based on another IBM file system called Tiger Shark [40]

### SGI XFS and CXFS

XFS [1] [68] [20] is the standard file system on Silicon Graphics computers. Since these machines are all shared memory or distributed shared memory computers, XFS does not have to deal with many of the problems that arise in the other parallel file systems discussed so far. In particular, there are no separate I/O nodes and no replication buffers. As a result, concurrency control is relatively simple. XFS uses standard Unix-style I/O calls, with a few extensions for special services, and it supports Unix consistency semantics.

CXFS (cluster XFS) [50] is an SGI extension to XFS that supports clusters of shared memory computers. A cluster consists of 2 to 16 computers (nodes) communicating over a private network. The nodes are also connected to a SAN that gives them access to a shared collection of storage devices. Although CXFS is based on XFS and shares many features with it, CXFS does not support guaranteed-rate I/O. CXFS also does not efficiently support concurrent writing of the same file by multiple processes on different nodes, except in direct I/O mode.

#### HPSS

HPSS (High Performance Storage System) [72] is very different from the parallel file systems described so far. Strictly speaking, it isn't a file system at all; it's an archival storage system, designed mainly for storing very large files and moving them quickly between primary, secondary, and tertiary storage. HPSS has a number of programming interfaces, one of which is based on Unix.

#### Miscellaneous

Other systems not mentioned yet would be e.g Thinking Machines' Scalable File System (sfs) [51] and nCUBEs Parallel I/O System [21].

### 2.6.5 Conclusion

In comparison to runtime libraries parallel file systems have the advantage that they execute independently from the application. This makes them capable to provide dynamic adaptability to the application. Further the notion of dedicated I/O servers (I/O nodes) is directly supported and the processing node can concentrate on the application program and is not burdened by the I/O requests.

However due to their proprietary status parallel file systems do not support the capabilities (expressive power) of the available high performance languages directly. They provide only limited disk access functionality to the application. In most cases the application programmer is confronted with a black box subsystem. Many systems even disallow the programmer to coordinate the disk accesses according to the distribution profile of the problem specification. Thus it is hard or even impossible to achieve an optimal mapping of the logical problem distribution to the physical data layout, which prohibits an optimized disk access profile.

Therefore parallel file systems also can not be considered as a final solution to the disk I/O bottleneck of parallelized application programs.

## 2.7 Cluster File Systems

### 2.7.1 Introduction

give a combination of the other two approaches, which is a dedicated, smart, concurrent executing runtime system. Cluster of workstations are a popular alternative to integrated parallel systems designed and built by a vendor. Well-known cluster projects include the Berkeley Network of (NOW) [3], NASA's Beowulf [5] and Sandia National Laboratory's Cplant [38]. The architecture of these systems vary, but they are all built from off-the-shelf components, presumably at a lower cost than an integrated system with equivalent hardware. Workstation cluster consist of a collection of PCs or other workstation computers (which usually include disk drives) connected by some kind of message passing network. The systems typically run a standard operating system such as Linux, and the have additional software to manage communication between the nodes.

The general architecture of a cluster is similar to a distributed memory parallel computer; it consists of multiple nodes that exchange data over a message passing network. Therefore, parallel file system for cluster must address many of the same problems as distributed memory file systems. These include consistency control and the choice between client and server buffering. In addition, cluster parallel file systems must contend with heterogeneity at several levels; individual clusters differ from each other; the hardware within a cluster may be heterogeneous. Cluster file systems manage the first two kinds of heterogeneity by using standard programming interfaces.

Some implement file service on top of the native file system running on each node (like ADIOS).

Many cluster computer make do with a distributed file system NFS and DFS can already work in heterogeneous, dynamic environments. Although many research projects have developed parallel file systems for cluster computers, no one system has emerged yet as a de facto standard. The next section looks at some of the design issues for cluster file systems, and the following section describes a few research systems in more detail.

### 2.7.2 Issues for Cluster File Systems

Many cluster have no dedicated I/O nodes, so file systems have to distribute their file management functions over the compute nodes. A cluster file system may logically divide its functions between a client interface library and several server tasks, but the servers often run on the compute nodes and access the local disks. The systems often use client buffering, and they use partitioning or token passing for concurrency control, just as file systems for integrated machines do.

A final problem for cluster file systems is how to stripe data. Since every node can act as both a compute node and an I/O node, the number of storage devices is often equal to the number of compute nodes. The arrangement offers a large aggregate I/O bandwidth.

### 2.7.3 Example Cluster File Systems

Many research groups have developed cluster file systems. This section will look at a few of these to see how individual systems handle the issues described above

### xFS

The xFS [4] file system was developed at Berkeley for their NOW project. xFS stripes files over stripe groups, and each stripe group forms a RAID unit controlled in software. There are no separate servers; file management software runs on all of the compute nodes, and each file is managed by one node. The manager's tasks include keeping track of the file's block on disk. ADIOS uses a approach close to xFS.

### PVFS (Parallel Virtual File System)

PVFS [12] [49] [11] is a research project at Clemson University. It is aimed at Beowulf-class clusters. Unlike xFS, PVFS uses I/O server processes that can run on separate I/O nodes. However, the system also allows the server software to run on compute nodes. The system focuses on file partitioning for concurrency control. If offers several interfaces that let processes define their own nonoverlapping subsets of a file. The model is similar to PIOFS [18] subfiles. Applications can define striping parameters individually for each file they create. PVFS does not manage file blocks directly; instead, it relies on the local native file system at each I/O node to handle this task. Although PVFS doesn't do buffering directly, there may be buffering in the underlying file system; if so, it's effects would be similar to server buffering.

### Other File System Research

Two other parallel computing project of interest are Legion [39] and Globus [29]. These systems focus on widely distributed networks of heterogeneous computers. Legion defines a programming model and an architecture for linking machines over long distances (i.e across the country) to form a unified metacomputing environment (compare with ADIOS islands 5.2).

Globus is another wide are computing project. It has defined a storage system called GASS [7] that gives multiple processes access to a common group of files. However, GASS is not a general-purpose parallel file system. In particular, it does not allow multiple processes to write at arbitrary locations in a file at the same time. Other approaches for data movement at the Globus project are GridFTP [2] and RIO [31].

The PANDA [62, 63] and the ADIOS system are examples for client server systems. (Note that PANDA is actually called a library by its designers. But since it offers independently running I/O processes and enables dynamic optimization of I/O operations during run time we think of it as a client server system according to our classification) ADIOS so far supports the static fit property [1]. The programmer can issue this property via a C function call or with the help of a XML file when using ADIOS like a file system.

## 2.8 Summary

This chapter has shown how file systems organize raw data blocks on storage devices into the familiar view of file and directories. File systems use caching and buffering to improve performance, especially for accesses to small amounts of data and for bursty access patterns. Distributed file systems give programs running on different computers access to a shared collection of files, but they are not designed to handle concurrent file access efficiently. Parallel file systems do handle concurrent accesses, and they stripe files over multiple I/O nodes to improve bandwidth.

[1]static fit: Data is distributed across available disks according to the SPMD data distribution (i.e. the chunk of data which is processed by a single processor is stored contiguously on a disk; a different processor's data is stored on different disks depending on the number of disks available)

Sequential Unix-based file systems have traditionally defined the semantics of read and write operations in a way that makes concurrent file accesses by separate processes appear to occur in a well-defined order. Maintaining these semantics in parallel and distributed file systems is difficult, so some systems relax the traditional semantics to improve performance. Other systems use various techniques to maintain standard Unix consistency semantics while endeavoring to offer good parallel performance.

Distributed memory parallel computers often route file access requests through specialized I/O nodes. Some parallel file systems do "server buffering" on these nodes, while others do "client buffering" on the compute node. Both approaches have strengths and weaknesses.

Computer vendors and researchers have developed many parallel file systems, some with novel programming interfaces. The trend in current commercial parallel file systems appears to be toward offering standard Unix semantics rather than specialized parallel I/O interfaces.

This high level overview was partly created with the help of [52].

# Chapter 3

# ADIOS Design

## 3.1 Introduction

ADIOS is a distributed I/O server providing fast disk access for high performance applications. It is an I/O runtime system, which provides efficient access to persistent files by optimizing the data layout on the disks and allowing parallel read/write operations. The client-server paradigm allows clients to issue simple and familiar I/O calls (e.g. 'read(..)'), which are to be processed in an efficient way by the server. The actual file layout on disks is solely maintained by the servers.

Since ADIOS-servers are distributed on the available processors, disk accesses are effectively parallel. The client-server concept of ADIOS also allows for future extensions like checkpointing, transactions, persistent objects and also support for grid enabled computing using the Internet.

ADIOS is primarily targeted (but not restricted) to networks of workstations. Client processes are assumed to be loosely synchronous.

## 3.2 Design Goals

The design of ADIOS followed a data engineering approach, characterized by the following goals.

1. *Scalability.* Guarantees that the size of the used I/O system, i.e. the number of I/O nodes currently used to solve a particular problem, is defined by or correlated with the problem size. Furthermore it should be possible to change the number of I/O nodes dynamically corresponding to the problem solution process. The *system architecture* of ADIOS is highly distributed and decentralized. This leads to the advantage that the provided I/O bandwidth of ADIOS is mainly dependent on the available I/O nodes of the underlying architecture only.

2. *Efficiency.* The aim of optimization is to minimize the number of disk accesses for file I/O. This is achieved by a suitable data organization (section 3.5.1) by providing a transparent view of the stored data on disk to the 'outside world' and by organizing the data layout on disks respective to the static application problem description.

3. *Parallelism.* This demands coordinated parallel data accesses of processes to multiple disks. To avoid unnecessary communication and synchronization overhead the physical data distribution has to reflect the problem distribution of the SPMD processes. This guarantees that each processor accesses mainly the data of its local or best suited disk. All file data and meta-data (description of files) are stored in a distributed and parallel form across multiple I/O devices. In order to find suitable data distributions to achieve maximum parallelism (and

thus very high I/O bandwidth) ADIOS may use information supplied by the application programmer. This information is passed to ADIOS via prescribing hints. If no hints are available ADIOS uses some general heuristics to find an initial distribution.

4. *Usability.* The application programmer must be able to use the system without big efforts. So she does not have to deal with details of the underlying hardware in order to achieve good performance and familiar *Interfaces* (section 3.5) are available to program file I/O.

5. *Portability.* The system is portable across multiple hardware platforms. This also increases the usability and therefore the acceptance of the system.

## 3.3 ADIOS Design

The system design has mainly been driven by the goals described in chapter 3.2 and it is therefore built on the following principles:

- Minimum Overhead. The overhead imposed by the ADIOS system has to be kept as small as possible. As a rule of thumb an I/O operation using the ADIOS system must never take noticeable longer than it would take without the use of ADIOS even if the operation can not be speed up by using multiple disks in parallel.

- Maximum Parallelism. The available disks have to be used in a manner to achieve maximum overall I/O throughput. Note that it is not sufficient to just parallelize any single I/O operation because different I/O operations can very strongly affect each other. This holds true whether the I/O operations have to be executed concurrently (multiple applications using the ADIOS system at the same time) or successively (single application issuing successive I/O requests). So in praxis the ADIOS system strives for a very high throughput.

- Use of widely accepted standards. ADIOS uses standards itself (e.g. PVM for the communication between clients and servers) and also offers standard interfaces to the user (for instance application programmers may use MPI-I/O or UNIX file I/O in their programs), which strongly enhances the systems portability and ease of use.

- High Modularity. This enables the ADIOS system to be quickly adopted to new and changing standards or to new hardware environments by just changing or adding the corresponding software module.

Some extensions to support for future developments in high performance computing also have been considered like for instance grid enabled computing.

## 3.4 Modules

The ADIOS system consists of the independently running ADIOS servers and the ADIOS interfaces, which are linked to the application processes. Servers and interfaces themselves are built of several modules, as can be seen in figure 3.1.

The ADIOS Interface library is linked to the application and provides the connection to the "outside world" (i.e. applications, programmers, compilers, etc.). Different programming interfaces are supported by *interface modules* to allow flexibility and extendibility. Currently implemented are a (basic) MPI-IO interface module, and the specific ADIOS interface which is also the interface for the specialized modules. Thus

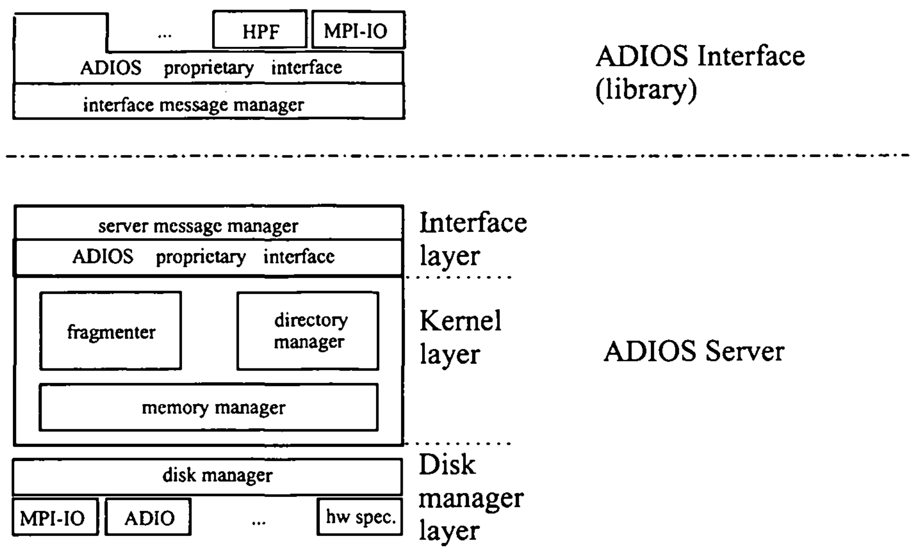

Figure 3.1: Modules of a ADIOS System

a client application can execute I/O operations by calling MPI-IO routines, UNIX file I/O or the ADIOS proprietary functions.

The interface library translates all these calls into calls to ADIOS functions (if necessary) and then uses the interface message manager layer to send the calls to the buddy server. The message manager also is responsible for sending/receiving data and additional informations (like for instance the number of bytes read/written and so on) to/from the server processes. Note that data and additional information can be sent/received directly to/from any server process bypassing the buddy server, thereby saving many additional messages that would be necessary otherwise and enforcing the minimum overhead principle as stated in chapter 3.3. (See chapter 4.4 for more details.) The message manager uses PVM-function calls to communicate to the server processes.

The ADIOS server process basically contains 3 layers:

- The **Interface layer** consists of a message manager responsible for the communication with the applications (*external messages*) as well as with other servers (*internal messages*). All messages are translated to calls to the appropriate ADIOS functions in the proprietary interface.

- The **Kernel layer** is responsible for all server specific tasks. It is built up mainly of three cooperating functional units:

  - The **Fragmenter** can be seen as "ADIOS's brain". It represents a smart data administration tool, which models different distribution strategies.

  - The **Directory Manager** stores the meta information of the data. Three different modes of operation have been designed, centralized (one dedicated ADIOS directory server), replicated (all servers store the whole directory information), and localized (each server knows the directory information of the data it is storing only) management. Until now only localized management is implemented. This is sufficient for clusters of workstations.

  - The **Memory Manager** is responsible for prefetching, caching and buffer management. Until now only buffer management is implemented.

- The **Disk Manager layer** provides the access to the available and supported disk sub-systems. Also this layer is modularized to allow extensibility and to simplify the porting of the system. Available are modules for ADIO [70], MPI-IO, and Unix style file systems.

## 3.5 Interfaces

To achieve high portability and usability the implementation internally uses widely spread standards (MPI, PVM, UNIX file I/O, etc.) and offers multiple modules to support an application programmer with a variety of existing I/O interfaces. In addition to that ADIOS can use different underlying file systems. Currently the following interfaces are implemented:

- User Interfaces

  Programmers may express their I/O needs by using

  - MPI-IO (see [67])
  - ADIOS proprietary calls (not recommended though because the programmer has to learn a completely new I/O interface. See chapter 6.1.1 for a list of available functions.)
  - UNIX file I/O (see chapter 6.2.)

- Interfaces to File Systems

  The filesystems that can be used by a ADIOS server to perform the physical accesses to disks enclose

  - ADIO (see [70]; this has been chosen because it also allows to adapt for future file systems and so enhances the portability of ADIOS.)
  - MPI-IO (is already implemented on a number of MPP's.)
  - Unix file I/O (available on any Unix system an thus on every cluster of workstations.)

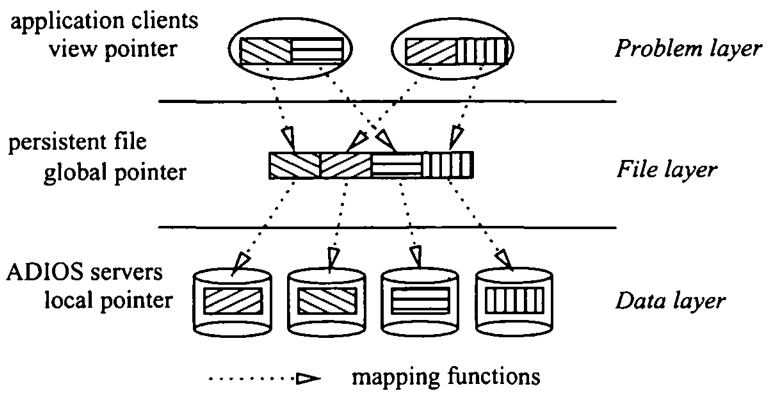

Figure 3.2: ADIOS data abstraction

  - Unix raw I/O (also available on any Unix system, offers faster access but needs more administrational effort than file I/O. Is not completely implemented yet.)

- Internal Interface

  Is used for the communication between different ADIOS server processes. Currently only PVM is used to pass messages.

### 3.5.1 Data Abstraction

ADIOS provides a data independent view of the stored data to the application processes.

Three independent layers in the ADIOS architecture can be distinguished, which are represented by file pointer types in ADIOS.

- Problem layer. Defines the problem specific data distribution among the cooperating parallel processes (View file pointer).

- File layer. Provides a composed view of the persistently stored data in the system (Global file pointer).
- Data layer. Defines the physical data distribution among the available disks (Local file pointer).

Thus data independence in ADIOS separates these layers conceptually from each other, providing mapping functions between these layers. This allows *logical data independence* between the problem and the file layer, and *physical data independence* between the file and data layer analogous to the notation in data base systems ([44, 10]). This concept is depicted in figure 3.2 showing a cyclic data distribution.

## 3.6 System Modes

ADIOS can be used in 2 different system modes, as

- runtime library,
- independent system.

These modes are depicted by figure 3.3.

### 3.6.1 Runtime Library.

Application programs can be linked with a ADIOS runtime module, which performs all disk I/O requests of the program. In this case ADIOS is not running on independent servers, but as part of the application. The ADIOS interface is therefore not only calling the requested data action, but also performing it itself. This mode provides only restricted functionality due to the missing independent I/O system. Parallelism can only be expressed by the application (i.e. the programmer).

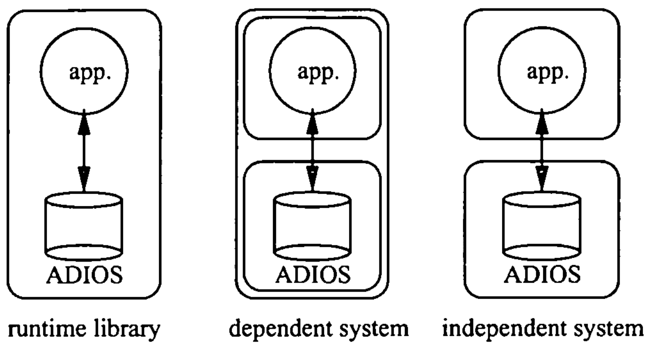

Figure 3.3: ADIOS system modes

### 3.6.2 Independent System.

This is the mode of choice to achieve highest possible I/O bandwidth by exploiting all available data administration possibilities. In this case ADIOS is running similar to a parallel file system or a database server waiting for application to connect via the ADIOS interface. This connection is realized by a proprietary communication layer bypassing MPI. We implemented an approach for coupling MPI worlds via PVM intermediate layers.

### 3.6.3 Implementation of a mapping function description

ADIOS has to keep all the appropriate mapping functions as part of the file information of the file. So a data structure is needed to internally represent such mapping functions. This structure should fulfill the following two requirements:

- Regular patterns should be represented by a small data structure.
- The data structure should allow for irregular patterns too.

```
struct Access_Desc {
        int skip_header;
        int no_blocks;
        int skip;
        struct basic_block *basics;
};

struct basic_block {
        int offset;
        int repeat;
        int count;
        int stride;
        struct Access_Desc *subtype;
};
```

Figure 3.4: An according C declaration

Of course these requirements are contradictionary and so a compromise actually was implemented in ADIOS. The structure which will now be described allows the description of regular access patterns with little overhead yet also is suitable for irregular access patterns. Note however that the overhead for completely irregular access patterns may become considerably large. But this is not a problem since ADIOS currently mainly targets regular access patterns and optimizations for irregular ones can be made in the future.

Figure 3.4 gives a C declaration for the data structure representing a mapping function.

An Access_Desc basically describes a number (no_blocks) of independent basic_blocks where every basic_block is the description of a regular access pattern. The skip entry gives the number of bytes by which the file pointer is incremented after all the blocks have been read/written. skip_header entry gives the number of bytes which are skipped before any information is read/wrote (e.g. skip header information in a file before data).

The pattern described by the basic_block is as follows: If subtype is NULL then we have to read/write single bytes otherwise every read/write operation transfers a complete data structure described by the Access_Desc block to which subtype actually points. The offset field increments the file pointer by the specified number of bytes before the regular pattern starts. Then repeatedly count subtypes (bytes or structures) are read/written and the file pointer is incremented by stride bytes after each read/write operation. The number of repetitions performed is given in the repeat field of the basic_block structure.

# Chapter 4

# Basic System Architecture Implementation

The ADIOS architecture is built upon a set of cooperating server processes, which run independently on an arbitrary number of network nodes and accomplish the requests of client applications. For distributed and cluster computing any network node with access to secondary storage can be used to run a ADIOS server process.

Each application process is linked with the ADIOS interface, which transfers the client requests and additional information supplied into request to ADIOS servers (see Figure 4.1). The interface also manages data transfer between client and servers and translates acknowledge messages from the server processes into appropriate return values for the request function called by the client process.

In order to keep the size of the interface small and to minimize its runtime overhead the interface does not keep any information about which server process manages which disks and files. Therefore it can not choose the server process best suited for a particular task but sends all the request message to one specific server, which is called the *buddy server* to the respective client. The buddy server is assigned to a client process at the time when the application connects to ADIOS and normally

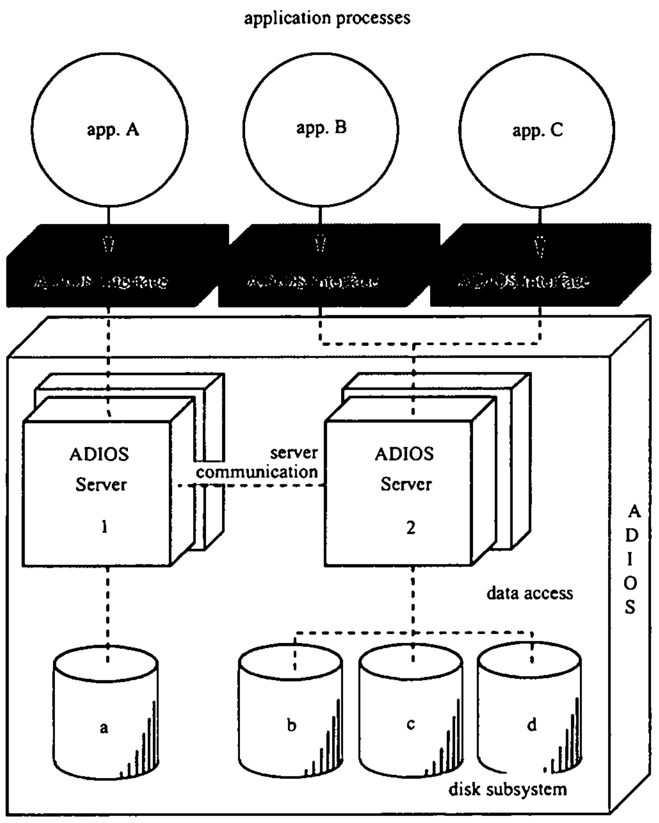

Figure 4.1: ADIOS architecture

remains the same until the termination of the connection. At any point in time each client process is linked to exactly one buddy server but a ADIOS server can serve any number of client processes (i.e. there exists a many-to-one relationship between clients and servers).

Any server process, which is not the buddy server for a specific client is called a *foe server* to that client. Because different client processes generally have different buddy servers the terms 'buddy' and 'foe' are always relative to a client process. So in figure 4.1 server 1 is buddy to application process A and foe to B and C. On the other hand server 2 is buddy to B and C and foe to A.

Server processes may run on dedicated or non dedicated nodes. A node is dedicated if the ADIOS server process is the only program running on that processor. Otherwise the node is non dedicated.

On non dedicated nodes the server process has to share the processor and other system resources with concurrently running tasks (which may also be processes of the client applications) and therefore the processing time consumed for optimizations of I/O operations has to be kept to a minimum.

However the use of dedicated nodes allows for extensive optimizations.

## 4.1 Data Access Modes

Naturally every server process can directly access only the disks connected to the processor node that it is running on. Since an application sends all I/O requests to its buddy server but can access data on any disk in the system two different types of data access have to be treated by a ADIOS server.

- **Local data access** stands for the case where the buddy server can resolve a request from the client application on its local disks. We call it also *buddy access.* (Examples for local accesses in the system depicted in figure 4.1 are requests from application A affecting disk a, or requests from applications B and C affecting disks b,c and d.)

- **Remote data access** denotes the access scheme where the buddy server can not resolve the request on its local disks but has to forward the request to other ADIOS servers. The respective server (foe server) accesses the requested data and sends it directly to the application via the network. We call this access also *foe access.* (Examples for remote access in the system depicted in figure 4.1 are requests from application A affecting disks b,c and d and requests from applications B and C affecting disk a.)

Note that the terms local and remote refer to the fact that disks are local or remote to the processor on which the buddy server process is running, not the processor on which the application process is running. (In case of non dedicated servers this may be the same processor but it does not have to be.)

If a request affects data on the local disks of the buddy server as well as data on remote disks, the request is broken into several parts in a way that each of the resulting subrequests can either be resolved by a local or by a remote data access. A more detailed description of this *request fragmentation* can be found in chapter 4.4.

ADIOS servers do not use special services like NFS to process remote access requests but rely on internal communication between ADIOS server processes. This speeds up the data access (no additional overhead) and also increases portability (independence of availability of remote access services).

## 4.2 Parallelizing I/O

There are two sources of I/O parallelism inherent in the ADIOS design.

An application according to the SPMD programming paradigm can connect each single application process (or subsets of application processes) with different buddy servers. This way each buddy server just performs sequential disk access. For the application as a whole the I/O operations are executed in parallel, since each buddy server can read from or write to its local disks autonomously.

In addition to that a ADIOS server can write to several local disks in parallel if allowed by the underlying hardware. Furthermore the data layout can be chosen in a way that remote disks are accessed. Since remote accesses are served by processes, which run on different processors they effectively can be processed in parallel to the local accesses.

## 4.3 ADIOS Server

A ADIOS server process consists of several functional units as depicted in figure 4.2, namely:

- The **Interface** provides the connection to the "outside world" (i.e. applications, programmers, etc.). Different interfaces are supported by *interface modules* to allow flexibility and extendibility. Up to now we implemented a (basic) MPI-IO interface module, and the ADIOS proprietary interface, which is in turn the interface for some specialized modules.

  Technically the interface is not really a part of the server process but linked to the client application.

- The **Message manager** is responsible for the external (to the applications via the interface) and the internal (to other ADIOS servers) communication.
- The **Fragmenter** can be regarded as "ADIOS's brain". It represents a smart data administration tool, which models different distribution strategies and makes decisions on the effective data layout, administration, and ADIOS actions.
- The **Directory Manager** stores meta information like file names, data distribution, data access logs and so on. In general the directory manager only holds the information for the (part of) data that resides on the local disks.
- The **Disk Manager** provides the access to supported disk sub-systems. This layer is modularized in order to allow extendibility and to simplify the porting of the system. Currently the Disk Manager supports modules for ADIO [70], MPI-IO, and Unix style file systems.

## 4.4 Requests and Messages

The following explains in detail how the various components of ADIOS collaborate to process an I/O request. The example deals with a write request. Read requests are processed similarly except where noted. For the sake of clarity the I/O operation is performed in several phases, which are depicted in figure 4.2 cont. In reality all these phases may overlap whenever possible.

For each phase the figure only depicts the servers actually involved in the processing of the request. Each server holds some part of the file's data, which is represented by small geometrical symbols (circle, triangle, square, diamond and trapezium).

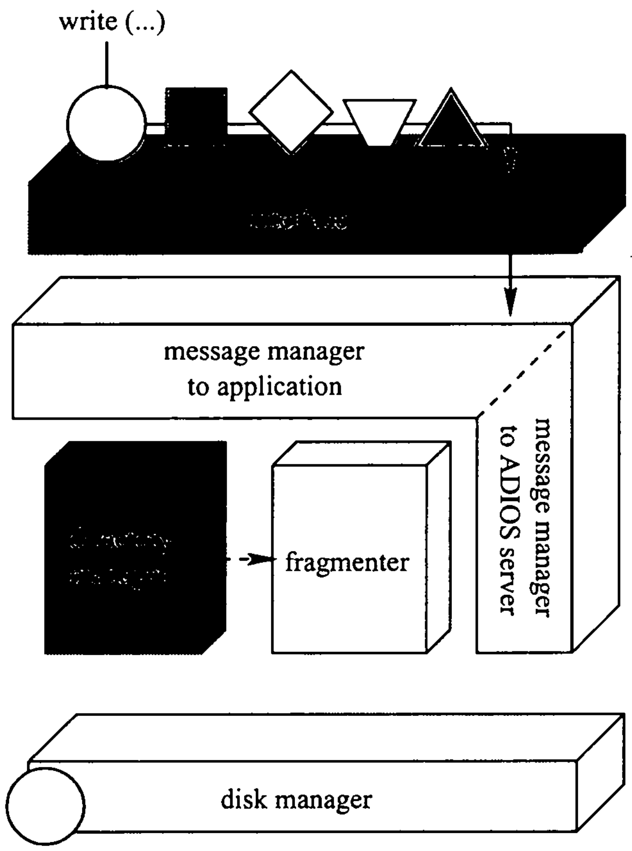

Figure 4.2: The Message Protocol: Phase 1

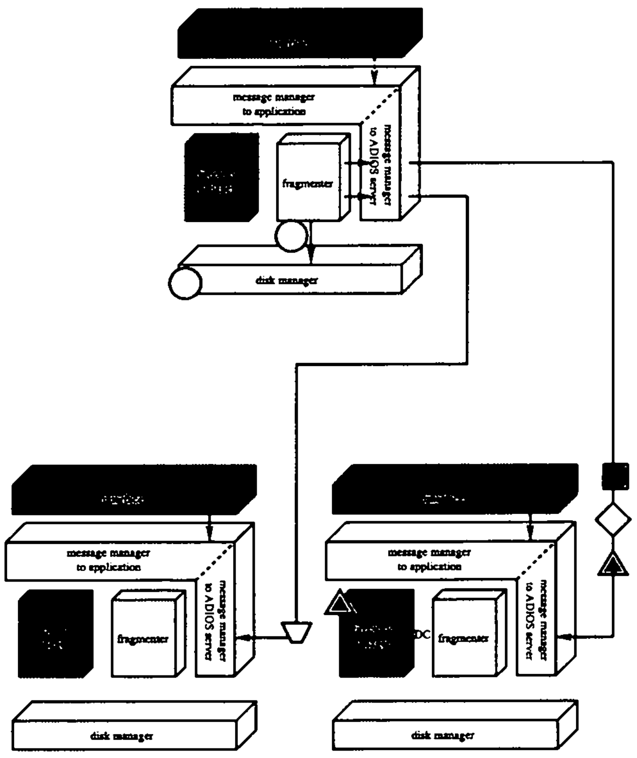
message manager to application
message manager to ADIOS server
fragmenter
disk manager
message manager to application
message manager to ADIOS server
fragmenter
disk manager
message manager to application
message manager to ADIOS server
fragmenter
disk manager

Figure 4.3: The Message Protocol: Phase 2

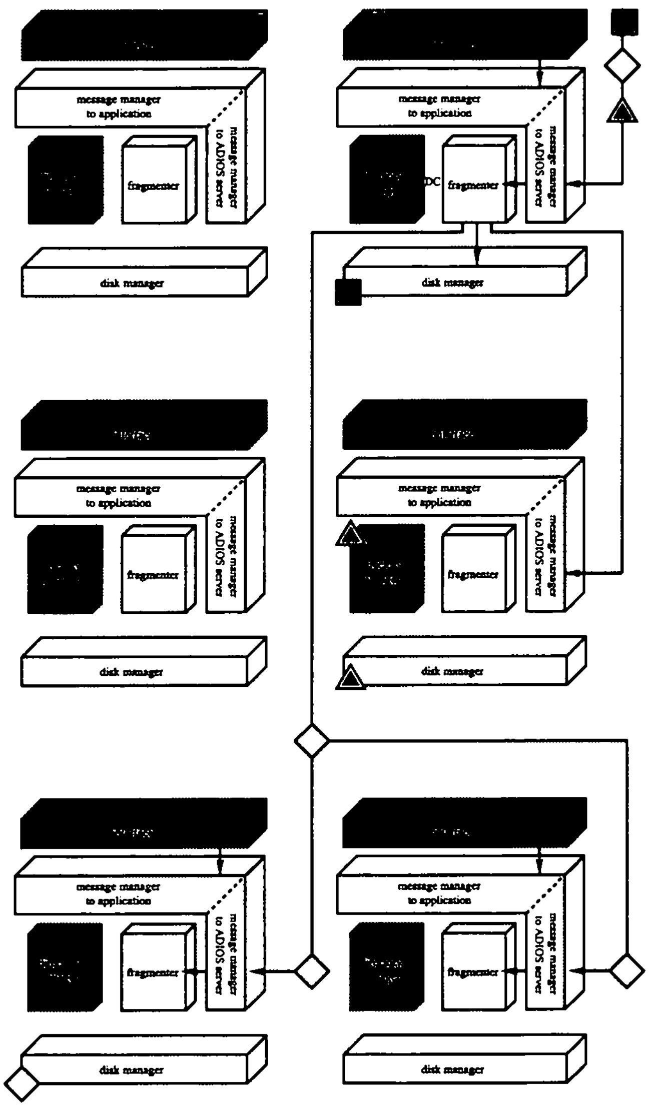
message manager to application
message manager to ADIOS server
fragmenter
disk manager
DC

Figure 4.4: The Message Protocol: Phase 3

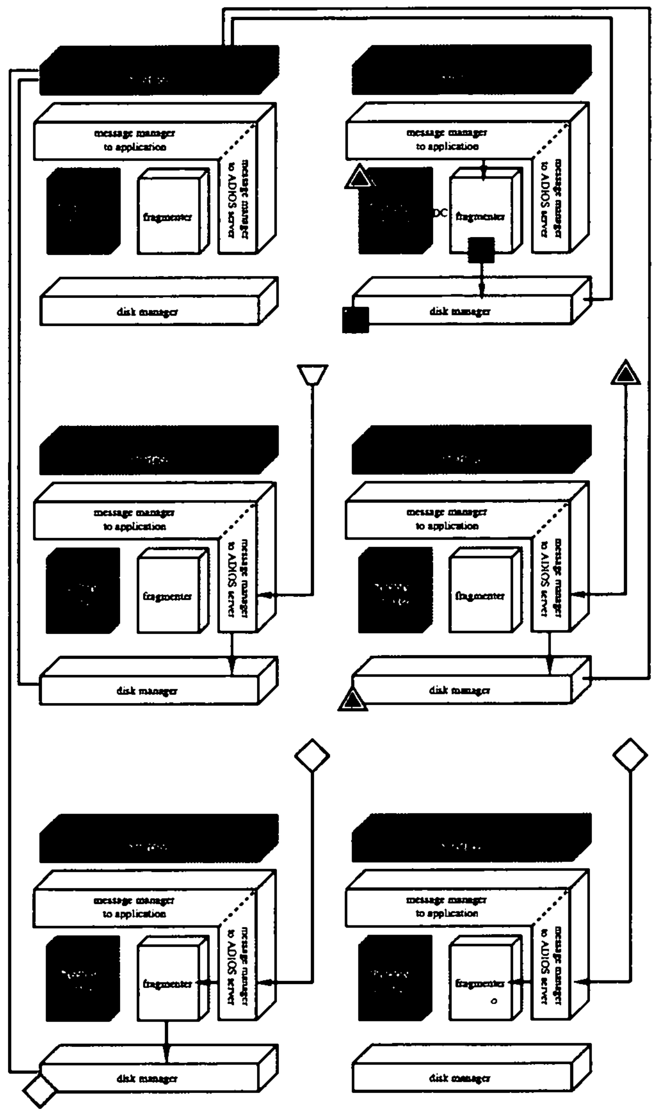

Figure 4.5: The Message Protocol: Phase 4

Full line arrows denote the flow of request messages. The request arrows are also marked with the geometrical symbols indicating the data which is actually requested. The dotted line arrows show the flow of meta information (directory information).

- **Phase 1: Request.** A write request is issued by an application via a call to one of the functions of the ADIOS interface, which in turn translates this call into a request message. Finally, this request message is sent to the buddy server.

- **Phase 2: Request Fragmentation.** The directory manager of the buddy server holds all the information necessary to map a client's request to the physical files on the local disks. The fragmenter uses this information to decompose the request into two sub-requests. One of which can be resolved locally. The other (the remote part) has to be communicated to other ADIOS servers (foe servers).

  If a directory controller (DC) exists for the file accessed, the sub-request for the remote part is forwarded to it. Otherwise the remote part is broadcast to all the other ADIOS servers and phase 3 can be skipped.

  Only for write accesses some part of the data may not be stored on any disk yet (data is appended to the file). The fragmenter then has to distribute this data over the available disks. After the fragmenter has decided, on which servers to store the data it can send corresponding request messages to these servers. In the example the trapezium symbolizes some data appended to the file.

- **Phase 3: Directory Controller Access.** The fragmenter of the directory controller once again breaks down the remaining part of the request according to information retrieved by its directory manager. In the example at hand one part (the square) can be resolved locally. For another part (the triangle) the directory manager can deliver information. This means that the fragmenter knows on which server this part of the data is stored and can therefore send this sub-request directly to the appropriate server. The rest is broadcast to the remaining servers in the system.

- **Phase 4: Disk Access and Data transfer.** At this point each affected server has received the request for the part of the data it administers. Note that messages that have been sent directly to a server can bypass the fragmenter (it is already known, that this server holds the part of the data in question) but messages that have been broadcast once again are filtered by the fragmenter. This time however only the part that can be resolved locally is of interest. Any other part can be safely ignored without triggering any additional messages (the request already has been broadcast to all possible servers).

  The I/O subsystems actually perform the necessary disk accesses for the local request and the transmission of data to/from the client process. For performance reasons each server communicates directly with the client bypassing the buddy server (indicated in the figure by the lines without arrows).

  Note that the part of the data symbolized by the trapezium is new and the appropriate server therefore has no meta data for this file on its disks at the start of the write operation. This is indicated by the lack of the symbol in the disk subsystem.

- **Phase 5: Directory Update and function return.** After the disk accesses have been performed all the directories (local and directory controller) are updated and the function initially called by the client returns indicating the success of the write operation. (This phase is not depicted in the figure.)

## 4.5 Customized Environments

ADIOS offers the adjustment of different system parameters to the application programmer by setting these parameters in an external config file. The path of this config file must be set in an environment variable (ADIOS_CONF=/home/ADIOS/src/develop). We distinguish between different hierarchical config files. (see 4.5.1)

These files are valid only in the scope of an island, which gives the application programmer the freedom to tune the parameters for different islands based on the knowledge about that island.

These parameters are not mandatory for ADIOS, we treat them as descriptive arguments, which ADIOS will fulfill as close as possible. But if there is a logical inconsistency between parameters, a appropriate value is chosen by ADIOS (e.g different message buffer sizes between servers).

### 4.5.1 Hierarchical Environment Config System

- local config file: such a file can exist for every Client and BS/FS. This file should hold data, which is private to that client/server like buffer size, name of the connection controller. For servers there are additional parameters like how many applications are supported

A client/server gets an parameter out of the local config file by issuing a request with the name of the parameter. Data which is consumed by every client and server (e.g buffer size, name of connection controller) must be consistent on all machines. This is in the responsibility of the ADIOS administrator.

### 4.5.2 Customizing

Till now, the following parameters can be customized

server parameters:

- MAX_APP: How many clients can connect to the BS
- MAX_SRV_FILE: How many file handles are offered by the BS
- DATA_BUFLEN: size of the message buffer
- FRAG_MEM_ENTRIES: How many fragments can be concurrently hold in the fragmenter
- SRV_GROUP_NAME: PVM group name for the servers (to distinguish between server and clients or client groups)
- SRV_DEVICE_LIST: How many and what devices are handled by this server

server and client parameters:

- ADIOS_DIR: Virtual mount point for ADIOS
- CC: Hostname of the connection controller (CC)

## 4.6 Unified messaging

### 4.6.1 Introduction

ADIOS is a parallel I/O runtime system, which provides efficient access to stored data sets by optimizing the data layout on disk and allowing parallel read/write operations. However in the last few years the focus of research in high performance computing shifted from parallel computing to distributed computing. New computing paradigms arose like Grid Computing, which enables world-wide-spread sharing and coordinated use of networked resources [28]. In the course of this process we extended ADIOS and developed ADIOS islands [60], which harness I/O resources available in distributed cluster type systems for high performance (parallel and/or distributed) applications. ADIOS islands focus on distributed, heterogenous environments, which is the common infrastructure of the Grid. The conventional approach of ADIOS using MPI [23] as communication layer fails in this environment by not supporting specific goals, as dynamic server lifetime, dynamic system configuration and so on. This situation leads to the development of a new transparent communication layer for heterogenous environments, which we will present in this paper.

We will describe the communication problem in this chapter, define our goals and show different solution approaches. Then we present our new communication architecture and describe ADCL [34], the new ADIOS Communication Layer in more detail.

### 4.6.2 The Communication Problem

Focusing the Grid we have to specify a very general Grid architecture hosting our framework. From our point of view the Grid consists of an arbitrary number of *collaborations*, which are defined by an organizational domain [30], interconnected by WAN technology. In practice such a collaboration will be usually (but must not be) a coherent IT infrastructure represented by a cluster like system, which consists of a number of execution nodes. These *nodes* are *processing nodes* and/or *data (server) nodes.* The latter type provides data storage resources by a number of storage *devices* (e.g. disks, tapes, etc.).

A ADIOS islands consists of a number of interacting ADIOS server processes, which are spread among a set of logically or topologically unified nodes (typical a collaboration). Applications can connect to a ADIOS island by contacting a defined connection controller, which in turn assigns "buddy" servers, responsible for the fulfilment of data requests, to any requesting application process. The original ADIOS system was designed as supporting module for parallel, high performance applications. Thus some typical simplifications were valid, as starting the ADIOS module together with the application, static configuration (server number and node layout) and so on. This made the choice of MPI as basic communication layer of ADIOS simple.

With the design of ADIOS islands and the focus towards the Grid environment a new functionality for the system and in turn new demands for the communication layer showed up.

- *Dynamic service.*

  ADIOS islands provide a data administration service to Grid applications. Therefore the lifetime of a ADIOS island is independent from the application using it. Theoretically an island is providing its service continuously. Thus a communication layer has to provide its functionality independently from any application and has to allow to connect applications dynamically.

- *Multi applications.*

  It must be possible that multiple applications connect concurrently to an islands. Thus any application must have its own communication handle.

- *Dynamic system configuration.*

  The system must be able to react to external events dynamically. For example if the workload increases more servers have to be installed, if the disk space diminishes new devices must be added, if a server crashes a new process has to be spawned, etc. This must be possible without restarting the communication infrastructure.

- *Heterogenous environment.*

  An islands is running on a set of defined nodes, but it is possible, that these nodes show different characteristics, as architecture, operating system, etc. Thus the communication layer must provide means for portability.

These problems are not totally new for us. We faced a similar situation during the design of the original ADIOS. We defined the problems and showed some possible work-arounds then (see [35]). Due to the urge for highest performance we had to choose MPI, with some specific extensions serving our needs.

MPI-1 restricts client-server computing by imposing that all the communicating processes have to be started at the same time. Thus it is not possible to have the server processes run independently and to start the clients at some later point in time. Also the number of clients can not be changed during execution

Some approaches to overcome the limitation are the following:

- *MPI-1 based implementations with work-around.*

  Starting and stopping processes arbitrarily can be simulated with MPI-1 by using a number of "dummy" client processes which are actually idle and spawn the appropriate client process when needed. This simple work-around limits the number of available client processes to the number of "dummy" processes started.

- *MPI-2 based implementations.*

  Supports the connection of independently started MPI-applications with ports. The servers offer a connection through a port, and client groups, which are started independently from the servers, try to establish a connection to the servers using this port. Up to now the servers can only work with one client group at the same time, thus the client groups requesting a connection to the servers are processed in a batch oriented way, i.e. every client group is automatically put into a queue, and as soon as the client group the servers are working with has terminated, it is disconnected from the servers and the servers work with the next client group waiting in the queue.

- *Third party protocol for communication between clients and servers (e.g. PVM).*

  This mode behaves like MPI-IO/PIOFS [16] or MPI-IO for HPSS [47], but ADIOS uses PVM [37] and/or PVMPI [25] (if it is available sometime) for communication between clients and servers. Client-client and server-server communication is still done with MPI.

All these above possibilities show limitations or are not applicable specifically in a Grid environment. Conclusively this led to the development of a novel, proprietary

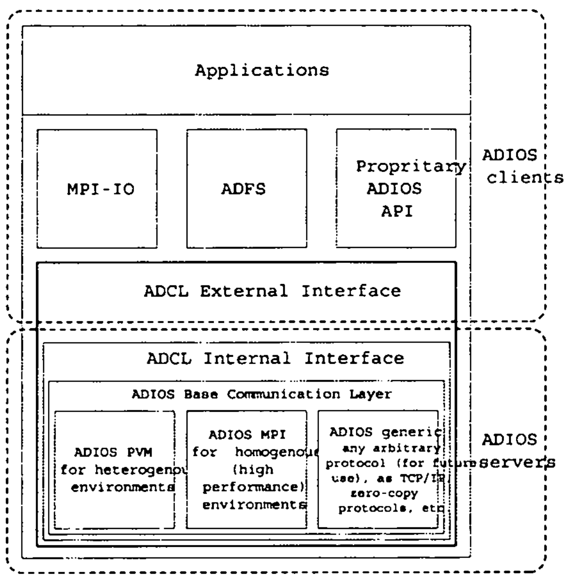
Applications
MPI-IO
ADFS
Propritary ADIOS API
ADIOS clients
ADCL External Interface
ADCL Internal Interface
ADIOS Base Communication Layer
ADIOS PVM for heterogenou environments
ADIOS MPI for homogenous (high performance) environments
ADIOS generic any arbitrary protocol (for future use), as TCP/IP, zero-copy protocols, etc
ADIOS servers

Figure 4.6: ADIOS Communication Architecture

transparent communication layer.

### 4.6.3 The ADIOS Communication Layer

The basic idea of our approach is to develop a new communication layer which inherits the advantages of our main communication libraries in focus, namely PVM and MPI. Simply said we are striving for a system usable as MPI but flexible as PVM. This led to the development of the ADIOS Communication Layer (ADCL).

The structure of our new communication architecture is depicted in Figure 4.6.

Applications connect to a ADIOS island as clients via a small ADIOS library, which has to be statically linked or dynamically loaded at runtime via the underlying operating system. These ADIOS clients have then the possibility to access the ADIOS functionality by several provided interfaces, a standardized MPI-IO interface, a distributed file system (ADFS), and a proprietary ADIOS interface. The

ADIOS system itself can have different execution modes, which are defined by the ADIOS base communication library. This base library is built upon communication standards, which are dependent or advantageous on the underlying hard- and software architecture. Until now we support a PVM-based system for a heterogenous architecture typical for cluster and Grid environments, and a MPI-based system for homogenous environments, as found in classical supercomputers. Also a mixture of both is possible. The ADCL encapsulates the base communication layer transparently to the user/applications and therefore allows for easy portability. Thus one big advantage of the ADCL approach is the possibility to support arbitrary protocols. It is straightforward to move the ADIOS islands to new communication bases, as zero-copy protocols for new network hardware, or pure TCP/IP for conventional Internet environment, because only a limited set of ADCL calls has to be rewritten, mostly in form of simple stubs.

### The ADCL Application programming interface

The ADCL API is basically a superset of the PVM and MPI interface inheriting the advantages of both worlds. It comprises all functions necessary for the communication layer delivering the above stated properties.

From an abstract point of view the set of functions can be partitioned into 2 groups, *external* functions, which can be used from both clients and servers, and *internal* function calls which are used by the ADIOS servers only.

**External API.** Functions of the external ADCL API are used for the communication of the client processes and the servers. They comprise calls for resource identification, data shipment and communication control. However these functions are also used by the ADIOS servers internally, but their main focus is the external environment.

`char *getSrvGrpName ()`

Returns the group name of the servers. This name is defined by the environment variable ADIOS_CONF. It is used to restricted broadcast operations onto ADIOS servers.

`int adios_msg_init ()`

Initializes the active send buffer.

`int adios_msg_pack (void *inbuf, int incount, int type, void *adios_buf, int *req_pos)`

Packs `incount` objects of type `type` referenced by `inbuf` into the active send buffer `adios_buf`.

`int adios_msg_unpack (void *adios_buf, int *adios_pos, void *inbuf, int incount, int type)`

Unpacks `incount` objects of type `type` from `inbuf` into `adios_buf`.

`int adios_msg_send (int send, int tag)`

Sends the contents of the active buffer to process with tag `tag`.

`int adios_msg_psend (void *inbuf, int incount, int type, int send, int tag)`

A simplified pack & send call, useful for sending scalar values. Initializes the buffer and sends `incount` objects of type `type` to process identified by tag `tag`.

`int adios_msg_recv (int recv, int tag)`

Receives the active receiving buffer of process `recv` identified by tag `tag`

`int adios_msg_precv (void *inbuf, int incount, int type, int recv, int tag)`

A simplified receive & unpack call, useful for receiving scalar values. Receives `incount` objects of type `type` in buffer referenced by `inbuf` from process `recv` identified by tag `tag`

`int adios_msg_bcast (const char *gname, int tag)`

Broadcasts the active send buffer to all subscribing processes of the process group `gname` identified by tag `tag`. See also the `getSrvGrpName` call for broadcasts within the ADIOS server group.

**Internal API.** The call of the internal API are invisible to the clients. They are used internally between the ADIOS servers for runtime information gathering. They basically map dynamic process identification to a unique, persistent (static) global server identifer (GA_ID). The GA_ID is similar (can be identical, but in our actual implementation in coded form) to the IP-address of the processing node the ADIOS server is running on. Due to the restriction that only one ADIOS server is allowed per node the GA_ID is unique.

`int adios_get_msgid_from_gaid (GA_ID gaid, int *msgid)`

Maps the GA_ID to the dynamic process identifer of the server (compare to `pvm_tid ()`).

`int adios_get_msgid_from_hostname (const char *hostname,`
`int *msgid)`

Maps the hostname of the node to the dynamic server process identifier.

`int adios_get_gaid_from_msgid (int msgid, GA_ID gaid)`

Maps the dynamic process identifier to the GA_ID.

`int adios_get_hostname_from_msgid (int msgid, char *hostname)`

Maps the dynamic process identifier to the hostname of the server.

`int adios_get_gaid_from_hostname (const char *hostname, char gaid[])`

Maps the hostname to the GA_ID.

### 4.6.4 Conclusion

ADCL is a proprietary, novel communication layer for ADIOS islands, an I/O service system for distributed and Grid computing. The use of ADCL solves many problems typical in an MPI environment and allows for dynamic system lifetime and dynamic configuration, multi application support and heterogenous environment providing a conventional MPI interfaces to distributed applications.

A prototype of ADIOS islands is already implemented. Preliminary performance tests showed that the usage of the ADCL is not for free. To reach the above defined design goal and the quality of portability we have to pay about 10% performance loss. However we are convinced that this is a small price for the new qualities reached.

We are just on the way to adapt ADIOS islands to come closer to the emerging Grid standards. Specifically we are just working on a version to be more OGSA [30] compliant. Finally we hope that we will get a fully functional Grid I/O service specifically targeting the problem domain arising in the CERN Datagrid project [64].

## 4.7 Queues

### 4.7.1 Introduction

As mentioned in Chapter 2.4 I/O systems use file buffering and caching to improve their performance. So we want to explain the techniques which are applied in ADIOS.

To avoid all the issues with cache coherence ADIOS doesn't support client buffering and replication on servers. Instead we introduce a request queue which uses server buffering (delayed write) for write requests and data sieving for read requests. The request queue acts like a FIFO for requests. The current implementation of ADIOS doesn't support pthreads, so ADIOS always switches between receiving request and fulfilling them, which could be made autonomous with a thread safe implementation, where one thread is responsible for receiving requests and one or more other threads are responsible for fulfilling them.

### 4.7.2 Optimization for read requests

Read requests gathered from the clients are written into the request queue. The servers works through the request queue and puts all requests and their data for the same client on a finite buffer. If either the buffer is full or the server finds a request for another client it sends the buffer to the client, which scatters the received data into the clients memory and resumes.

### 4.7.3 Optimization for write requests

Write requests gathered from the clients are written into the request queue. The servers works through the request queue and collects all requests for the same client on a finite buffer. If either the buffer (request header information plus reserved data size) is full or the server finds a request for another client it pulls the data from the client, and writes the data on the disk. This leads to a pipelining on the server because of delayed write, i.e the program resumes while the I/O buffers are written to disk.

### 4.7.4 Conclusion

With this simple optimizations for read and write requests the network traffic is reduced to the minimum and the challenge is to find the right size for the "finite buffer" to optimize the tradeoff between the size of the buffer in memory (which is lost for other buffer activities and the application demanded memory) and the number of messages in the network. See the chapter of practical tests for further discussion of this tradeoff.

Further optimization can be achieved by developing an algorithm which does "intelligent" reordering in the request queue, but this is out of scope of this thesis.

## 4.8 Solving the EOF problematic in distributed I/O systems without a centralized directory structure

This chapter presents a new algorithm for solving the EOF problematic in distributed filesystems without a centralized directory server. The problem arises because the canonical form of a file is not in an one-to-one relationship to the physical representation (but in an one-to-many relationship). So, the classical way of signaling EOF doesn't work anymore when reaching the last physical byte like in traditional UNIX file systems. The novel algorithm will be presented by explaining how it is manufactured into ADIOS.

### 4.8.1 Introduction

We want to introduce our novel algorithm with three examples including increasing complexity. A further constraint is the absence of a centralized directory server, i.e all participating server only knows the part of files, which they physically manage. Every ADIOS server has a local directory service, which stores name and size of every locally managed file. This is basis for our algorithm to signal EOF correctly. We start with the first, very simple example.

### 4.8.2 Single server - single directory structure

This is the classical case, the file is stored physically exactly as it is represented by its canonical form. If the request goes beyond of the size of the file, stored in the local directory service, the server sends EOF to the client. This is the classical UNIX behaviour with the difference, that the local directory service decides, if we have reached EOF and not the physical file itself.

### 4.8.3 Single server - multiple directory structure

In this case ADIOS distributes the file guided by the embedded XML structure from the client or automatically on the server, rather similar to software RAID. Now we have the issue that the mapping from the canonical to the physical representation of the file is not an one-to-one, but an one-to-many relationship; e.g an cyclic distribution with block size of 4K on the server. Compared to the first example we have now the problem, that the size of the file stored in the local directory service gives no qualitative usefull answer about EOF, because there are now multiple entries for every file name (one entry per mountpoint).

Excursus: Example of a 13K sized file "test.adios" distributed one a server with "stripe factor = 3" and "stripe depth = 4K"

Environment variable (see ADIOS.conf on the Server): SRVR_DEVICE_LIST 3 /home/fuerle/ADIOS/dev1/ /home/fuerle/ADIOS/dev2/ /home/fuerle/ADIOS/dev3/

Related XML structure:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PARSTORAGESYSTEM "Xparstorage.dtd">
<PARSTORAGE VERSION="1.0"TIMESTAMP="testfile_twodevices">
<ISLAND>
  <SERVER HOST="adclus10">
    <DEVICE DEVICE_ID="0">
      <VIEW SKIP_HEADER="0" SKIP="8192">
        <BLOCK OFFSET="0" REPEAT="1" COUNT="4096" STRIDE="0">
          <BYTEBLOCK/>
        </BLOCK>
      </VIEW>
    </DEVICE>
    <DEVICE DEVICE_ID="1">
      <VIEW SKIP_HEADER="0" SKIP="4096">
        <BLOCK OFFSET="4096" REPEAT="1" COUNT="4096" STRIDE="0">
          <BYTEBLOCK/>
        </BLOCK>
      </VIEW>
    </DEVICE>
    <DEVICE DEVICE_ID="2">
      <VIEW SKIP_HEADER="0" SKIP="0">
        <BLOCK OFFSET="8192" REPEAT="1" COUNT="4096" STRIDE="0">
          <BYTEBLOCK/>
        </BLOCK>
      </VIEW>
    </DEVICE>
  </SERVER>
</ISLAND> </PARSTORAGE>
```

Only the addition of all file sizes in the local directory service for the given filename results in the total size, so that we reach the behaviour of the first example; i.e the server has to add 4K of device 0 plus 4K of device 1 plus 4K of device 2 plus 1K of device 0, this a simple local addition on the server.

### 4.8.4 Multiple server - multiple directory structure

In this case ADIOS distributes the file guided by the embedded XML structure from the client or automatically on the server, rather similar to machine wide software RAID. Now we have the issue that the mapping from the canonical to the physical representation of the file is not an one-to-one, but an one-to-many relationship; e.g an cyclic distribution with block size of 4K on the server. Compared to the second example we have now the problem, that even the cumulative size of the file stored in the local directory service gives no qualitative usefull answer about EOF, because there are now multiple entries on multiple servers for every file name (one entry per mountpoint on every participating server) and the absence of global directory controller.

Excursus: Example of a 13K sized file "test.adios" distributed one a server with "stripe factor = 3" and "stripe depth = 4K"

Environment variable (see ADIOS.conf on the Server): SRVR_DEVICE_LIST 3 /home/fuerle/ADIOS/dev1/ /home/fuerle/ADIOS/dev2/ /home/fuerle/ADIOS/dev3/

Related XML structure:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PARSTORAGE SYSTEM "Xparstorage.dtd">
<PARSTORAGE VERSION="1.0" TIMESTAMP="testfile_twodevices"> <ISLAND>
  <SERVER HOST="adclus10">
    <DEVICE DEVICE_ID="0">
      <VIEW SKIP_HEADER="0" SKIP="8192">
        <BLOCK OFFSET="0" REPEAT="1" COUNT="4096" STRIDE="0">
          <BYTEBLOCK/>
        </BLOCK>
      </VIEW>
    </DEVICE>
  </SERVER>
  <SERVER HOST="adclus11">
    <DEVICE DEVICE_ID="0">
      <VIEW SKIP_HEADER="0" SKIP="4096">
        <BLOCK OFFSET="4096" REPEAT="1" COUNT="4096" STRIDE="0">
          <BYTEBLOCK/>
        </BLOCK>
      </VIEW>
    </DEVICE>
  </SERVER>
  <SERVER HOST="adclus12">
    <DEVICE DEVICE_ID="0">
      <VIEW SKIP_HEADER="0" SKIP="0">
        <BLOCK OFFSET="8192" REPEAT="1" COUNT="4096" STRIDE="0">
          <BYTEBLOCK/>
        </BLOCK>
      </VIEW>
    </DEVICE>
  </SERVER>
</ISLAND> </PARSTORAGE>
```

An obvious possibility for this issue would be to qualify one server as the master server, which does the job of adding up the accumulated file sizes per server over all servers and sends EOF. This concept was part of design of ADIOS, but was not accomplishable because of technical restrictions. ADIOS servers are not thread-able, and the implementation of this concept would have resulted in a dead lock situation in the case of EOF (because all servers would have tried to communicate with a blocking send-receive with each other).

We needed to find a solution, where not the servers communicate with each other about EOF, but the client can find a decision via the answers of the servers. But in this case the problem is, that servers only provide information about bytes they own. They can also provide a kind of EOF, that they don't own any other bytes, but the client can't make a final decision about EOF with this information.

Example: client reads test.adios (13 K overall size) in 4K steps

The first 3 requests are working properly, because the request can be fully satisfied. In the 4th run of the loop the client requests 4K. It gets EOF with 1K from server 1. The clients has now 1K of data and the information EOF (from server 1) and still waits for another 3K of data.

### Questions and Challenges

1. The client is perhaps confronted with multiple EOF answers, what shall it do ? An example for this behaviour in the above example would be a request size of 16K instead of 4K. In this case all servers would respond with their data and EOF, i.e the clients gets EOF 3 times ?

2. The overall size can only calculated by accumulating all server answers, but the client doesn't know the number of participating servers ?

3. Even if the client would know the number of participating servers, it would need to communicate with servers, which doesn't contribute to the final result (see example 1.) server 2 and server 3) to get the size of the owned files (4K per server) and to add up to 13K, if the client gets more than one EOF and we calculate in absolute numbers ?

### Solution based on an analogon in communications engineering

The solution to this challenge is based on an analogon in communications engineering, the so called "power factor correction" in neon lights. The have an induction coil, which consumes additional to the "real current" a considerable fraction of "reactive current" when starting (i.e when the light switch is powered on), which is compensated (i.e neutralized) with a capacitor ("negative reactive current").

Spoken in words of mathematics this means that in the complex numbering system the sum over the "fictive parts" must be zero.

Back to the example with 4K requests we want to demonstrate this novel approach in the last run, where the client requests 4K and had already got 12K out of 13K till now.

The request of the client consists in our case of a "real part" (1K in our example, i.e the difference between 13K and 12K) of the available physical bytes on the servers and the "fictive part" (3K in our example, i.e the difference between 4K total request size and 1K physical available bytes) not available physical (i.e "fictive")bytes. The client doesn't know about the "fictive part" and hence sends the total request (i.e.

"real part" + "fictive part" = 4K) to the servers. The servers qualify the "real part" and the "fictive part" ("negative fictive part" = capacitor effect), that they go on with reading even in the case of EOF (hint: logical reading in the directory service, because physical reading wouldn't be feasible) until the whole request is fully satisfied. All bytes behind EOF are added up to the "negative fictive part" of each server.

Now all servers send their answers to the client which can realize by the use of the sent "negative fictive parts" its own "fictive part" and neutralizes it, i.e after processing all answers from the servers the sum of all "fictive and negative fictive parts" must be zero and the client can decide (hint: total request size must be equal to sum of "fictive parts" + sum of "real parts") , if it has received all "real parts" (i.e physical bytes) and if EOF has appeared (at least one server has send an "negative fictive part",i.e. the "negative fictive part" is greater than zero).

### Questions and Challenges revised

1. The client is confronted with multiple EOF answers, what shall it do ?
   The client doesn't get EOF anymore, it generates EOF explicitly when at least one server sends a "negative fictive part" greater than zero.

2. The overall size can only calculated by accumulating all server answers, but the client doesn't know the number of participating servers ?
   With our new algorithm it's not necessary anymore for the client, how many participating servers are around, because it uses instead of absolute only relative numbers based on the last issued request (e.g request 4K), and after that it subtracts as long "real and negative fictive parts" as it reaches zero (4K - 1K "real" - 3K "fictive" = 0). The request is finished, when it reaches zero.

3. Even if the client would know the number of participating servers, it would need to communicate with servers, which doesn't contribute to the final result (see example 1.) server 2 and server 3) to get the size of the owned files (4K per server) and to add up to 13K, if the client gets more than one EOF and we calculate in absolute numbers ?

   The client needs to communicate only with the contributing servers, i.e which have a "real or negative fictive part" greater than zero for this request. This is only server 1 in our example, server 2 and 3 don't send an answer to the client, because the "real and negative fictive parts" for those are servers are zero. As a positive side effect this also reduces network traffic.

### 4.8.5 Conclusion

This novel approach doesn't need a centralized directory controller and hence scales 100 %. So this algorithm subsummarizes all previous mentioned (see 4.8.3, 4.8.4) algorithms and hence is the used algorithm for all use cases in ADIOS.

# Chapter 5

# Extended System Architecture for distributed I/O

## 5.1 Introduction

The basic concepts of ADIOS described thus far need some extensions in order to harness I/O resources distributed over the internet. The main challenges in this context are

- The message protocol described in chapter 4.4 uses broadcasts in some situations. Since it is clearly impossible to broadcast across the internet some *notion of locality* is needed, which ensures that broadcast messages only have to be sent to a (small) well defined subset of all the ADIOS server processes running.
- *Name spaces* have to be provided to avoid file naming conflicts.
- *Client grouping* ensures that collaborating client processes can use shared file-pointers or access a file exclusively (i.e. only processes belonging to a specific group can use the file concurrently, whereas all other processes are denied access).

- Hard- and software environments across the internet are very inhomogenous. Hence the *adaptability* of ADIOS is a major issue. Administrators should be able to tailor the system to their needs.

## 5.2 The ADIOS Island

A *ADIOS island* is defined to be a closed system with its own name space consisting of a number of ADIOS servers and a connection controller, which assigns application processes to their buddy servers on request.

The idea is to segment the distributed I/O services into domains (islands). To reach such an island the client needs to know the hostname (or IP-address) of the connection controller responsible for that island.

### 5.2.1 The Connection Controller

At any given time, a client or a group of clients can connect/disconnect to/from a ADIOS island. To connect the client calls an interface function and specifies the IP-address of the targeted island's connection controller (see figure 5.1). The ADIOS interface then sends a connect message to that connection controller, which in turn selects a buddy server for the client process (based on information about network topology, data layout and so on). The address of the buddy server is sent back to the ADIOS interface. The interface converts this address into a *buddy handle* and returns this handle to the calling client process. The client has to use this handle for all further requests to the respective ADIOS island.

A client process may connect to an arbitrary number of ADIOS islands concurrently (like indicated in figure 5.2). Since there is a different buddy server to the

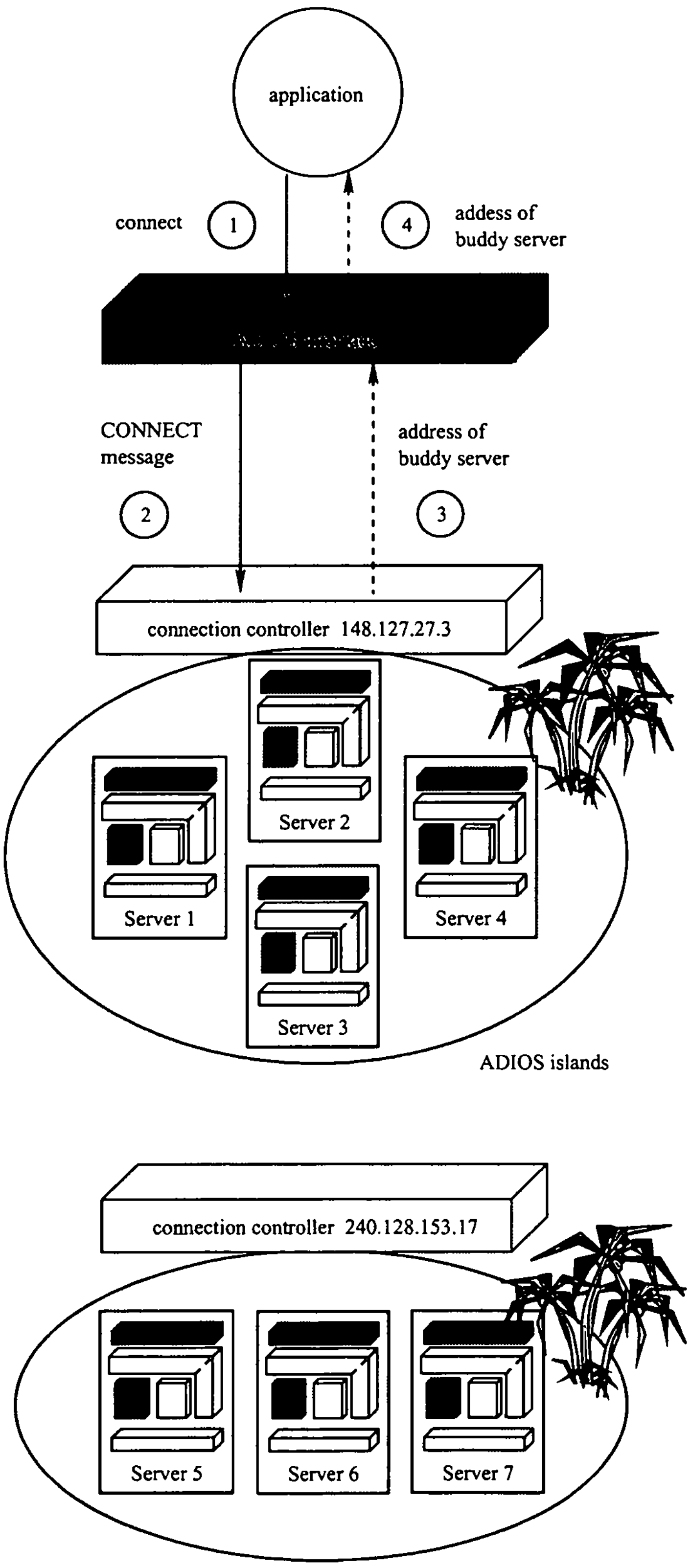

Figure 5.1: Four steps to connect to a ADIOS island

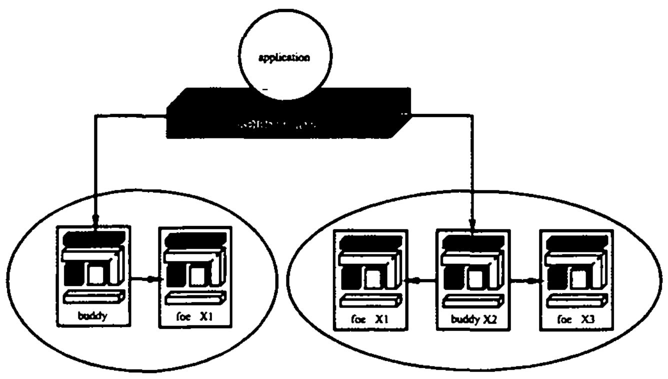

Figure 5.2: ADIOS islands

application in each island the many-to-one relationship between applications and buddy server (see 4) holds no more. Each application has exactly one buddy server in each island it is connected to. This behaviour is currently not implemented. A simple extension of all function calls in the ADIOS client stub interface with the buddy server id could do so.

### 5.2.2 Name Space of ADIOS

Each ADIOS island has its own name space, i.e a file name is unique within an island, but on the other hand the same file name can occur in different islands.

All parts of a single file are stored on one dedicated island. Therefore it is not possible that for any file some bytes have to be retrieved from one island and other bytes have to be retrieved from another island. If a part of the file is located on an island, the rest can be found on the same island. This simple rule restricts the range of broadcast messages to one single island. Whenever a part of a file is searched on an island, which can neither be found locally nor by the directory controller, it suffices to broadcast the request to all the servers in the island. One of them has to hold the data.

To distinguish between files on different islands with the same name, the application programmer would need to specify the buddy handle when calling an I/O operation (see last paragraph in 5.2.1).

## Excursion: Technical background of the distributed version of ADIOS

To be able to connect/disconnect at any given time, ADIOS must be run as a client-server based system. So the servers are running like daemons, which must be started ahead of the clients.

So, the current client-server version of ADIOS is implemented in PVM and transforms from MPI to PVM, where necessary. A big advantage of PVM is that is well prepared for working on heterogeneous platforms, even for heterogeneous message passing and it offers the possibility to create failure tolerant applications.

We started implementing ADIOS with LAM/MPI to implement such a system, because mpich doesn't offer such MPI2 functions (spawning of processes, port functions) . But we stopped that development for two reasons

- no possibility for failure recovery in MPI, if one process fails, the whole system fails. We are not interested to recover from hardware crashes like a harddisk crash, this is not the aim of the recovery component. But we are more interested in a system, which can survive minor failures like temporary unavailability of services or network congestion.

- LAM/MPI in those days, which was 6.1, doesn't seem to be stable enough for writing programs, which use that kind of MPI2 functionality.

- Different MPI implementations on different platforms doesn't work together, whereas PVM programs run on all supported platforms.

We are thinking of implementing a LAM/MPI based client-server version of ADIOS without failure recover component for performance comparison reasons. On the other hand we want to investigate in the portability of our unified messaging system, which will be explained later on. This system allows switching of the underlying messaging system without the need to rewrite application code, even the internal code of ADIOS.

### 5.2.3 The Global Application ID

After connecting to this island, the client receives a unique GA_ID directly from the buddy server, which is assigned for this client based on decisions made by the connection controller. This GA_ID is used by the client for all further requests to ADIOS, so that the client stub can distinguish, to which buddy server the requests must be forwarded to. So from the client point of view there is no difference sending a request to the local or a remote island, this is transparent.

The GA_ID for clients consists of the hostname (respective the IP-address, which is via DNS more or less the same) and the process-ID of the client. We need the process-ID of the client, because it is possible, that a machine runs one or more clients (at least in a UNIX multitasking environment), so we need an additional parameter to the machine name. So, the process-ID seems to be a good choice, because it is as long valid and unique for that process on the machine, as the process lives.

an according C struct for the GA_ID

```
union {
    int          pvm_id;
    MPI_Comm     mpi_id;
} GA_ID;

typedef struct {
    long int     ip_address;  /* long int because
                                 of upcoming IPv6 */
    int          process_id;

    char         *group_name; /* for access right */
    GA_ID        bh;          /* buddy handle id  */
} FULL_GA_ID;
```

The buddy server also stores the GA_ID for following reasons

- which and how many clients are connected Hint: Only the buddy server stores the GA_ID of the client, because all requests from the client must be sent to the BS, which forwards requests, which cannot be fulfilled locally to the foe servers. This doesn't necessarily means, that the foe servers must be aware of the remote clients.
- access rights for this clients (e.g. shared file pointer)

The GA_ID for servers consists only of the hostname, because we assume, that there can be only one server process per host.

When disconnecting a client from the server, the BS discards the GA_ID of the client.

Hint: After deleting all clients, a BS can not be shutdown, because it must still answer to remote requests issued by foe servers. I.e ADIOS servers can be only shut down together after a broadcast operation, where all servers confirmed, that no clients are connected anymore.

The GA_ID is transformed internally to the respective id of the underlying messaging system. This internal id is used for the buddy handle id, which is used for communication between the client stub and the server. If the respective internal id is not unique, the FULL_GA_ID is used for that purpose.

The idea of a GA_ID is not implemented currently, we still use the id of the underlying messaging system for the buddy handle id.

### 5.2.4 Group tagging

On the first view different processes can connect at any time, so one important feature of a distributed I/O versions is the tagging of processes, which form together a group with common access rights (e.g. opening of a shared file pointer).

We pursue the simulation of a SPMD approach for a consumer (e.g. any kind of calculation) -producer (e.g a visualizer) program. In general there will be at some discrete time stamp a group of clients for the producer program and at another (probably some time later) discrete time stamp a group of clients for the visualizer program. A major problem, when e.g. using PVM is, that even for that discrete time stamp PVM can't recognize, that this clients are one common group, because PVM pursues the concept of independent processes (but groups can be formed explicitly by issuing group functions); compare that with MPI_COMM_WORLD in MPI-1, where groups are set up explicitly.

From the point of view of the connection controller we need a common tag, that those clients should tie together to a group and we need to know, how many are clients are waiting to be connected.

Another property of the common tag must be multiple usability, i.e we need to know, that all clients of the producer, but also all clients of the consumer program establish a common group (e.g. common access rights for a shared file pointer).

To address this requirements we chose two additional parameters for the connect function in ADIOS

- **a user defined group name** The application programmer defines a customized group name, which must be issued at the connect function for all participating clients (e.g for all clients of the producer-program). This makes it possible, that e.g. two independent consumer-producer programs work side a side without disturbing each other. The drawback of this approach is that different groups of application programer could unintentionally choose the same group name and each program interferes the other one. But on the other hand this first approach is aimed for application programmers, which are aware of each other and choose unique names (e.g. technical reports names or similar).

- **number of clients** In the case of issuing a customized group name the connection controller needs to know, how many clients wait for registration at the CC, because at this time stamp there is no explicit barrier function available like in MPI-1 (e.g when using PVM as underlying messaging system). This is mainly necessary for letting the CC know, when the registration procedure for this client group is over, and on the other hand for optimization purposes, so the CC can assign the best available BS for the waiting clients. Hint: When using MPI-2 (e.g LAM/MPI) as underlying messaging system, this information is implicit given by brokering (the ADIOS stub does that automatically in that case) the MPI_COMM_WORLD communicator of the waiting client group.

Another property of this group tagging functionality as mentioned before (recall the consumer-producer program) is the extendibility of already registered groups.

After registering the producer clients at the CC at any given time stamp later the consumer clients can register at the CC by issuing the same group name as before. The number of consumer clients may not be same as the number of producer clients, the internal client table of the CC is just extended by the number of the new clients.

A nice side feature of this functionality is that client groups can be adjusted in a customized way by the application programer by even just adding/deleting one or more "worker" processes at any given time stamp.

## 5.3 File Operations

For all further requests (read, write, close, etc.) the client needs to issue the buddy handle together with the Island File_ID.

### 5.3.1 The Island File ID

ADIOS needs to offer the ability of shared file pointers and grants for accessing a file exclusively (compare with group tagging). Therefore we need a unique file_ID for every file name in an island.

We denote such a file_ID an Island File_ID. When creating a new file, the buddy server broadcast such a request to the remaining foe servers. The ADIOS server, which holds at least the first byte (part) of this file uses its GA_ID plus a consecutive number to establish the IF_ID and returns this value to the client, which use the IF_ID and the GA_ID for further requests on that file.

When a BS receives a request, it opens (if not done yet) the necessary physical file handle and broadcasts the IF_ID to all remaining FS, which in turn open also the necessary physical file handle and insert the IF_ID in a map, which maps the relation between the logical IF_ID and the physical file handle.

A BS distinguishes between a regular/shared file pointer by returning an autonomous reference to that IF_ID in the former case (every group gets a different reference) and the same reference to the IF_ID (for multiple groups of clients) in the later case. Shared file pointers are not implemented yet.

# Chapter 6

# Interfaces

ADIOS offers a wide variety of (external) interfaces for different purposes.

The main interfaces are:

- *A native ADIOS interface*, which is functionally viewed a superset of the traditional Unix interface, with extensions similar to MPI-IO and PVFS. It is used internally, but can also be used for application programming.
- *ADMPIOS: a MPI-IO interface*, which is an almost complete implementation of chapter 9 of the MPI-2 draft.
- *ADFS: a file system interface*, which implements a file system with its common tools on top of ADIOS delivering persistence and a canonical view for the distributed files.

We want to concentrate on the native ADIOS and the ADFS interface; ADMPIOS, the MPI-IO interface of ADIOS is out of the scope of this work (see [67] for details).

## 6.1 The Native ADIOS Interface

The native interface of ADIOS is the main interface to ADIOS. It provides functions for connecting to and disconnecting from the system, file manipulation and data access and various administrative tasks. Due to its proprietary status it is usually transparent to the application programmer, but builds the basis for the standardized interfaces as MPI-IO and ADFS.

The native interface comprises functions for

- ADIOS administration, connecting to and disconnecting from ADIOS,
- basic file administration and manipulation, as creation, opening, closing, querying and deletion of files,
- file access in blocking and non-blocking mode supporting the various data layout patterns.

To explain how to apply the ADIOS native interface we use as example a simple application program written in the MPI/MPICH framework. It is assumed that the *ad_serv* program has been precompiled and the ADIOS native interface library *libadios.a* resides in the same directory as the example program.

First, the application program must be compiled and linked with the ADIOS library. The syntax is the same as for an usual C or FORTRAN compiler. For example,

```
mpicc -o ad_client application1.c libadios.a
```

Thus, the application program *application1.c* is compiled as a client process called *ad_client*.

Next, the application schema must be written. This is a text file which describes how many server and client processes are used and on which host they run. A possible application schema *app-schema* for one server and one client process is:

```
adios2 0 /home/usr1/ad_serv adios1 1 /home/usr1/ad_client
```

In that example the server process *ad_serv* is started on the host called *adios2* whereas the client process *ad_client* is started on the host *adios1*.

The simple example program connects to the island *"adios.tuwien.ac.at"*, opens a file called *infile*, reads the first 1024 bytes of the file and stores them in a file called *outfile* and disconnects from ADIOS.

The client program *application1.c* looks like follows:

```
#include <stdio.h>
#include "mpi.h"
#include "ad_func.h"

void main ( int argc, char **argv ) {
  int   i,fh1, fh2;
  char  infile [15], outfile [15], buf[1024];
  GA_ID bh;

  MPI_Init (&argc, &argv);

  ADIOS_Connect ("adios.tuwien.ac.at", NULL, -1, &bh);
  ADIOS_File_open (bh, infile, AD_MODE_RDONLY, NULL, &fh1);
  ADIOS_File_read (fh1, -1, (void *) buf, 1024);
  ADIOS_File_close(fh1);

  ADIOS_File_open (bh, outfile, AD_MODE_WRONLY | AD_MODE_CREATE,
                   NULL, &fh2);
  ADIOS_File_write(fh2, -1, (void *) buf, 1024);
  ADIOS_File_close(fh2);

  ADIOS_Disconnect(bh);
}
```

The next step is to specify e.g the number of servers (2) and clients (4) which should be involved in the computation. Thus, a text file has to be defined called e.g. *appl1-schema*, which contains the following lines:

```
ADIOS1 0 /home/usr1/ad_serv adios2 1 /home/usr1/ad_serv adios2 4
/home/usr2/kurt/ad_client
```

The server and the client program reside in the specified directories, and the server process *ad_serv* is started once on *ADIOS1* (the 0 denotes the machine, where this scheme is started from with mpirun -p4pg appl1-schema) and on adios2; the four client processes on *adios2*.

Note: If you use PVM as the underlying messaging system, you don't need such schemes. Processes (server and clients) are spawned directly from the PVM console or can be called directly on the shell.

### 6.1.1 Native Interface Prototypes

### Connecting and Disconnecting

Before an application program can use ADIOS, a connection must be established.

**int ADIOS_Connect(const char* ADIOS_Island_ID, const char *gname, int gcount, GA_ID *bh)**

| | | |
|---|---|---|
| IN | ADIOS_Island_ID | hostname of CC |
| IN | *gname | connecting group name |
| IN | gcount | number of clients in "gname" |
| OUT | bh | the buddy server id |

**Description:** Initializes ADIOS via the DNS entry of the CC in the corresponding island and establishes a connection between an application program and ADIOS. Returns the buddy server id.

**Example:** `ADIOS_Connect("adios.tuwien.ac.at", NULL, -1, &bh);`

**int ADIOS_Disconnect(GA_ID bh)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |

**Description:** Disconnects the application program from ADIOS.

**int ADIOS_Shutdown(void)**

**Description:** Shuts ADIOS down. All processes are closed and FAT of ADIOS is written back by the local DCs. This function can only be used by an administrative interface.

File Manipulation

**int ADIOS_File_open(GA_ID bh, const char *filename, int amode, SERVER_DIST *desc, int *fh)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | filename | name of the file |
| IN | amode | file access mode |
| IN | *desc | desired distribution |
| OUT | fh | file identifier |

**Description:** Opens an existing or creates a new file with the mode defined in *amode*.

**Example:** `ADIOS_File_open (bh, "matrix", AD_MODE_RDONLY, NULL, &fh1);`

**int ADIOS_File_close(int fh)**

| | | |
|---|---|---|
| IN | fh | file identifier |

**Description:** Closes an open file.

**int ADIOS_File_delete(GA_ID bh, const char *filename)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | filename | name of the file |

**Description:** Deletes an existing ADIOS file.

**int ADIOS_File_set_size (int fh, int size)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| INOUT | fh | file identifier |
| IN | size | size (in bytes) to truncate or expand file |

**Description:** Resizes the file defined by *fh*.

**int ADIOS_File_get_size (int fh, int *size)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier |
| OUT | size | size of the file in bytes |

**Description:** Returns the current size in bytes of the file defined by *fh*.

## Data Access

### Blocking Routines

**int ADIOS_File_read (int fh, int at, void *buf, int count)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | at | byte offset |
| OUT | buf | initial address of buffer |
| IN | count | number of bytes to read from file |

**Description:** Reads data from an open file denoted by the file identifier into *buf*. The parameter *at* states whether the operations is a so-called routine with *explicit offset* or not. Further information is given in the next routine.

**Example:** `ADIOS_File_read (fh1, -1, buf, 15);`

**int ADIOS_File_read_struct (int fh, int at, void *buf, int count)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | at | offset relative to the displacement |
| OUT | buf | initial address of buffer |
| IN | count | number of bytes to read from file |

**Description:** Reads data from an open file in a strided way according to the file access pattern, i.e. the file view, specified by *ADIOS_File_set_view ()*. The parameter *at* allows distinguishing between data access with explicit offset and data access with an individual file pointer. The value -1 means that the file is read from the current position. Any value greater than 0 sets the file pointer to the specified position. However, since data access with explicit offsets should not interfere with data access with individual file pointers (see corresponding section in the chapter about MPI-IO), the file pointer is not updated after the read operation. Thus, the file pointer is only updated if the value of the parameter *at* is set to -1. This behaviour is not implemented yet.

**Example:** `ADIOS_File_read_struct (fh1, -1, buf, 40);`

40 byte values are read according to the file access pattern defined by *ADIOS_File_set_view ()*. Since the parameter *at* is set to -1, the data access with an individual file pointer is simulated. Thus, the file pointer is updated.

**Example:** `ADIOS_File_read_struct (fh1, 80, buf, 40);`

Here, data access with explicit offset is simulated. The file is read from position 80 relative to the file access pattern. In contrast to the previous example the file pointer is not updated.

**int ADIOS_File_write (int fh, int at, const void *buf, int count)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | at | byte offset |
| IN | buf | initial address of buffer |
| IN | count | number of bytes read from file |

**Description:** Writes data contained in *buf* to an open file denoted by the file identifier. The parameter *at* states whether the operations is a so-called routine with *explicit offset* or not.

**Example:** `ADIOS_File_write (fh1, -1, buf, 15);`

**int ADIOS_File_write_struct (int fh, int at, const void *buffer, int count)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | at | offset relative to the displacement |
| OUT | buf | initial address of buffer |
| IN | count | number of bytes to write to file |

**Description:** Writes data in a strided way according to *ADIOS_File_set_view ()* to an open file. The parameters and *at* have the same meaning as in *ADIOS_Read_struct* we have analyzed above.

**Example:** `ADIOS_File_write_struct(fh1, -1, buf, 40);`

## Non-Blocking Routines

**int ADIOS_File_iread (int fh, int at, void *buf, int count, int *req_id)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | at | byte offset |
| OUT | buf | initial address of buffer |
| IN | count | number of bytes to read from file |
| IN | req_id | identifier of the request |

**Description:** Reads data from an open file denoted by the file identifier into *buf* in a non-blocking way.

**Example:** `ADIOS_File_iread (fh1, -1, buf, 15,&req_id);`

**int ADIOS_File_iread_struct (int fh, int at, void *buf, int count, int *req_id)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | at | byte offset |
| OUT | buf | initial address of buffer |
| IN | count | number of bytes to read from file |
| IN | req_id | identifier of the request |

**Description:** Reads data from an open file denoted by the file identifier into *buf* in a non-blocking way.

**int ADIOS_File_iwrite (int fh, int at, const void *buf, int count, int *req_id)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | at | byte offset |
| IN | buf | initial address of buffer |
| IN | count | number of bytes read from file |
| IN | req_id | identifier of the request |

**Description:** Writes data contained in *buf* to an open file denoted by the file identifier in a non-blocking way.

**Example:** `ADIOS_File_iwrite (fh1, -1, buf, 15,&req_id);`

**int ADIOS_File_iwrite_struct (int fh, int at, const void *buf, int count, int *req_id)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | at | byte offset |
| IN | buf | initial address of buffer |
| IN | count | number of bytes read from file |
| IN | req_id | identifier of the request |

**Description:** Writes data in a strided way according to *ADIOS_File_set_view ()* to an open file.

**int ADIOS_File_test (int req_id, int *flag)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | req_id | identifier of the request |
| OUT | flag | flag |

**Description:** This routine checks whether an outstanding non-blocking routine has finished. The result is given in *flag.*

**Example:** `ADIOS_File_test (req_id, &flag);`

**int ADIOS_File_wait (int req_id)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | req_id | identifier of the request |

**Description:** This routine waits until an outstanding non-blocking routine has finished.

**Example:** `ADIOS_File_Wait (req_id);`

**int ADIOS_File_set_struct (int fh, Access_Desc *desc)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | desc | initial address of the access descriptor |

**Description:** Stores the given Access_Desc *desc* as default view for all further xxx_struct accesses of file *fh* in the island *bh.*

**int ADIOS_File_get_struct (int fh, Access_Desc *desc)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | desc | initial address of the access descriptor |

**Description:** Retrieves the current Access_Desc *desc* of file *fh* in the island *bh.*

### Further Access Routines

**int ADIOS_File_seek (int fh, int offset, int whence)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | offset | absolute file offset |
| IN | whence | update mode |

**Description:** Updates the file pointer of a file according to *whence*, whereas following features are possible:

- SEEK_SET: pointer is set to *offset*
- SEEK_CUR: pointer is set to the current pointer position plus *offset*
- SEEK_END: pointer is set to the end of file

**Example:** `ADIOS_File_seek (fh1, 50, SEEK_SET);`

The file pointer is set to position 50 of the file denoted by the file identifier.

**int ADIOS_File_seek_struct (int fh, int offset, int whence)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier assigned in ADIOS_File_open |
| IN | offset | absolute file offset |
| IN | whence | update mode |

**Description:** Updates the file pointer of a file according to *whence* within a predefined file access pattern rather than merely in a contiguous way.

**Example:** `ADIOS_File_seek_struct (fh1, 50, SEEK_SET);`

The file pointer is set to position 50 of the file according the file access pattern, i.e. file view set by *ADIOS_File_set_view ()*.

**int ADIOS_File_get_position (int fh, int *pos)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier |
| OUT | pos | position of file pointer |

**Description:** Returns the current position of the individual file pointer in bytes relative to the beginning of the file.

**int ADIOS_File_get_count (int fh, int *count)**

| | | |
|---|---|---|
| IN | bh | buddy handle id |
| IN | fh | file identifier |
| OUT | count | number of bytes or entities read/written |

**Description:** Returns the number of bytes (no view) or entities (if there is file view set on *fh*) of the last read/write operation on the file handle *fh*.

**int ADIOS_Get_hostbyname (const char *hostname, GA_ID bh)**

| | | |
|---|---|---|
| IN | hostname | hostname |
| OUT | bh | buddy handle id |

**Description:** Returns the buddy server id *bh* (the GA_ID) by resolving the real world hostname *hostname*. Works rather similar to DNS.

**Example:** `ADIOS_Get_hostbyname ("adios.tuwien.ac.at", &bh);`

## 6.2 ADFS, the Filesystem Interface

ADFS is a filesystem on top of ADIOS. It provides a set of the common file system (POSIX standard) calls mapping them transparently to respective ADIOS calls. This allows on one hand the persistent storage of distributed files viewed in a logical canonical form, on the other hand the use of ADIOS inherent parallelism to speed up file accesses.

Summing up ADFS is aiming at

- providing tools to manage files on ADIOS similar to the Unix commands e.g. cp, mv, rm, ls, ...
- delivering a C-Interface for application development similar to existing IO-functions e.g. open, write, read, close, fprintf, ...
- viewing files as continuous data - at the file layer - and hiding the physical distribution from the user. The user can however specify the physical distribution at file creation and change the distribution of an existing file,
- taking advantage of parallelism due to the underlying physical distribution

However ADFS does not support logical file views at the problem layer. Thus files are always handled as continuous data at the file layer. Low level services such buffering and caching, prefetching, synchronization, and data distribution are not provided by ADFS itself, but by the functionality of the underlying ADIOS. ADFS is only an interface that allows users to use easily and efficiently services provided by ADIOS in a well-known standardized environment.

### 6.2.1 Design of ADFS

ADFS implements a command-line interface and a C language interface providing basic functionality similar to the equivalent Unix commands or Unix C-interface. Further it delivers extended functionality, allowing the user or application to make use of special features provided only by ADIOS, as choosing the data layout, giving hints etc.

ADFS consists basically of a library, which maps the well-known POSIX file routines (as open(), write(), read() etc.) to equivalent ADIOS calls if applicable. Thus programs linked with this library use ADIOS transparently bypassing the conventional POSIX calls. Thus it is simple to realize a command line interface to manage files on ADIOS similar to the Unix Commands. The programs (e.g. for cp, mv, etc.) have to be simply re-linked with the new library. In case of a dynamic loadable library this is done during the call of the respective command by the operating system automatically.

Even more the library can be linked to any application using the POSIX calls, which accesses ADIOS files automatically.

### Command-line Interface.

The following commands are supported by ADFS:

cp (copy files to ADIOS, copy files from ADIOS, copy files within ADIOS), mv (move files to ADIOS, move the files from ADIOS, move the files within ADIOS), rm (remove files from ADIOS), ls (list ADIOS files), cat (concatenate ADIOS files), more (list the contents of a file), od (octal dump), vi (edit a file)

All file management commands can be called with additional parameters to define or change the disk layout of the file in focus.

When installing ADIOS, a Unix directory (default: /ADIOS) is specified which contains the ADFS file space. Files copied to this directory are transparently distributed and managed by ADIOS.

### C-language Interface.

ADFS provides a POSIX-type C library which can be linked to applications. Concerning the base functionality, the ADFS function calls for accessing files show the same synopsis as standard C function calls. Thus the programmer has only to replace stdio.h by the ADFS header file, compile the program, link it to the ADFS library and run the new program with ADIOS parallel reads and writes.

The native interface base functionality is derived from the POSIX standard (and the ANSI standard which is a subset of the POSIX standard). The following functions will be supported:

- fclose, feof, ferror, fflush, fgetc,fgetpos, fgets, fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, fsetpos, ftell
- getc, putc, rewind, setbuf, setlinebuf, setbuffer, setvbuf, open, close, read, write

We start now with the description of xDGL, the underlying technique behind ADFS and will continue with examples for ADFS at the end of this chapter.

### 6.2.2 Introduction

Meta information in the context of Grid computing has to describe not only the logical part of the data (semantical information) but also specific structural information on the physical distribution of the data (syntactical information). Thus we propose an approach for an XML based language to act as a notational tool to describe all this information for data stored, administered, searched and processed on the Grid. Any information stored on the Grid (from a conventional text file to a structured database relation) is attributed with a semantic description expressed by the XML notation. In the most simple case the XML description is stored together with the file.

Only a few similar approaches exist, but these are in a early state (e.g. [36]) or target mostly very specific application domains (e.g. [9] [41]).

In the next section we present xDGDL [27], the XML-based Data Grid Description Language, and give several examples for the usage of the language. Then we introduce shortly the Meta-ADIOS system [33], which is a client server based I/O system supporting distributed applications on the Grid. Finally we present an prove-of-concept implementation of the xDGDL language within the ADFS, the distributed file system component of the ADIOS system.

### 6.2.3 xDGDL - the XML Data Grid Description Language

We propose the XML Data Grid Description Language (xDGDL) which aims to provide a convenient XML framework for the specification of meta information of data stored on the Grid. xDGDL is a derivative of PARSTORAGE [6], which was specifically designed as meta language for parallel IO data.

The xDGDL descriptor consists of a logical and a physical view to the file. The logical view describes the semantical information and the physical view the syntactical information (the physical layout) of the file.

Focusing the Grid we have to specify a very general Grid architecture hosting our framework. From our point of view the Grid consists of an arbitrary number of *collaborations*, which are defined by an organizational domain [30], interconnected by WAN technology. In practice such a collaboration will be usually (but must not be) a coherent IT infrastructure represented by a cluster like system, which consists of a number of execution nodes. These *nodes* are *processing nodes* and/or *data (server) nodes.* The latter type provides data storage resources by a number of storage *devices* (e.g. disks, tapes, etc.). It is to note that a single data node can host an arbitrary number of devices.

### The goals of xDGDL

The basic idea of the XML based approach is quite simple: Together with any "chunk" of data a xDGDL description of the meta information of the data is stored, in other words, any arbitrary number of bytes stored within our framework is attributed with its describing information, delivering the following properties:

**Semantics of data** Applications write results to files. There are lots of applications, there are lots of formats, there are lots of files. But what can be found in these files? Generally applications do not write simple bytes into a file. They write integers, real numbers, characters, records of arbitrary types etc. So the contents of a file is not just a sequence of bytes, but it is a sequence of typed elements. Without the knowledge of the semantics of the applications, we have no clue about its contents. Further the application that created it, used its own format, a format that is known to this application only. Today we have the urge for analyzing and processing data found on the Grid (as in typical OLAP applications), thus there is an undeniable need for semantic description. Simply said, data without semantics is dead, data with semantics lives. This statement leads naturally to the next issue, persistency of data.

**Persistency of data** Data stored without semantic information is lost (can not be reused), because the semantics is originally only in the program code of the application producing the data. Without the program the data is just a sequence of bytes without meaning. With the usage of a framework like xDGDL the data can be reused easily by any application understanding the meaning of the data. A practical Java-based example is given in [6].

**Portability** In a distributed environment parts of data can migrate from one node/system/environment to another. On different hosting environments naturally the data formats change. However when moving data from one system to another, applications must still be able to read the data. By the description of the format the data can be interpreted and can be easily transformed to any proprietary format of the target machine [36].

**Performance and efficiency** To enhance the bandwidth of the IO media (to fight the famous IO bottleneck) it is the most common technique to distributed the data among different nodes and/or devices and perform the accesses in parallel. If the user has knowledge about the available nodes or the application behavior she can describe the distribution of the file to her needs. This can lead to performance improvements especially if the user is aware of node's performance, the given network latency, the network bandwidth to each server, etc.

### The xDGDL specification

The Extensible Markup Language (XML) is the universal format for structured documents and data on the Web. It describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markups. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

The structure of XML is fundamentally tree oriented. Therefore a document can be modeled as an ordered, labeled tree, with a document vertex serving as the *root* vertex and several *child* vertices. Without the document vertex, an XML document may be modeled as an ordered, labeled forest, containing only one root element, but also containing the XML declaration, the doctype declaration, and perhaps comments or processing instructions at the root level.

To define the legal building blocks of an XML document, a DTD (Document Type Definition) can be used. It defines the document structure with a list of legal elements.

A DTD can be declared inline in your XML document, or as an external reference.

It was a clear decision to choose XML as the basis for our framework due to its undeniable success within the Internet community and its acceptance as basis for beneath any standard movement in the Grid community (e.g. WSDL [13]).

### The xDGDL document type definition

In our framework a typical xDGDL description consists of the following elements:

- **Document Root** The root of the document specifies the version and timestamp of the file of the XML description.
- **Island** Defines a logical unit with several servers distributed worldwide. This element resembles the collaboration of our simple Grid architecture given above.
- **Server** Servers are physical machines identified by their host name. These servers denote data nodes.
- **Devices** Devices are the disks holding the data on the specific server.
- **View** The View element allows a specific distribution within the device.
- **Block** The Block element specifies the number of bytes to write to the specific disk.

The complete DTD of xDGDL can be found in the Appendix.

**Document root** The root of the document is described by the element `PARSTORAGE`. It has the attribute `VERSION` that contains the version of the document and the attribute `TIMESTAMP` that identifies the external name together with the logical file. Both attributes are mandatory.

The root element can contain several child elements. The `PROCESSORS` and the `ALIGN` children are optional. The following child elements are possible:

- PROCESSORS describes the named processor arrays. A document may contain zero or more processor array definition, which are normally derived from the HPF definition.
- TYPE describes the data types and variables stored in the logical file. Types enhance the quality of stored data. They allow to define the meaning of the information stored. This leads to the fact that not only the program that stored the data can use them. Every program that understands the type information of the data can use the stored bytes. Because of these meta information it is also possible to migrate data from one machine to another. There must be at least one TYPE element in the document.
- ALIGN describes the alignments of the variables.
- ISLAND describes the physical view of the file.

Example:
```
<PARSTORAGE VERSION="1.0" TIMESTAMP="testfile_twoserver">
    <TYPE>
        ...
    </TYPE>
    <ISLAND NAME="tuwien.ac.at">
        ...
    </ISLAND>
</PARSTORAGE>
```

**Island** The ISLAND describes several server interconnected together. These servers can be distributed across the Grid. The island is identified by an island name. The ISLAND consists of one or more servers. At least one server is needed to write the file sequential to that server. The number of servers are received from the number of child present. Example:

```
<ISLAND NAME="tuwien.ac.at">
    <SERVER HOST="adios.tuwien.ac.at">
    </SERVER>
</ISLAND>
```

**Server** The SERVER identifies uniquely a node. It has an attribute called HOST which mirrors the name of the server.

The SERVER element consists of one or more DEVICE elements. At least one must be present for each server to know how the file should be distributed on the several disks. For this purpose the number of available devices on a specific server should be known.

Example:

```
<SERVER HOST="adios.tuwien.ac.at">
    <DEVICE DEVICE_ID="/dev/vda1">
    </DEVICE>
</SERVER>
```

**Device** Devices are the disks holding the data on the specific server. On one SERVER there could be more than one physical device. The server can have a RAID system for example with several disks connected onto it. The devices need not be physical, even a mounted NFS device on another server could be a device which could be accessed from a processing node. Although there can be many devices on a specific server, in most cases there will be only one device available.

The DEVICE element consists of the attribute DEVICE_ID only, which specifies the physical device on the system. To describe the structure of file parts to be written to disk, a VIEW is used. If there is no VIEW defined we expect that the file should be written sequential by the "first" logical server and the "first" logical disk on this server.

Example:

```
<DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="7">
    </VIEW>
</DEVICE>
```

**View** The VIEW element is the link between logical, physical and application view. It is responsible for transforming the internal structure of the data layout to application programs.

A specific distribution is expressed by a VIEW element. The VIEW needs to correspond to the servers available. The NOVIEW elements marks that there is no VIEW element available. If NOVIEW is the only available child, the pointer to the access-descriptor is set to NULL and therefore the file will be written sequentially onto the disk. At least a VIEW or a NOVIEW element has to be present.

The VIEW consists of the SKIP_HEADER attribute that describes how many header bytes are skipped at the beginning of the data block and the SKIP attribute that defines the number of bytes to be skipped viewer units.

The VIEW element consists of one or more BLOCK elements. Theoretically there can be an infinite number of BLOCK elements, but at least one is needed. The BLOCK itself can have another VIEW element within itself.

Example:

```
<VIEW SKIP_HEADER="0" SKIP="7">
    <BLOCK OFFSET="0" REPEAT="3" COUNT="5" STRIDE="7">
        <BYTEBLOCK/>
    </BLOCK>
</VIEW>
```

**Block** The BLOCK element can have two types of childs. It can have a BYTEBLOCK element, which means, that either there are no more VIEW elements or it can consist of VIEW elements which have one or more BLOCK elements themselves. This leads to a recursive structure which allows arbitrary distribution. At least one has to be present.

The BLOCK element consists of the following attributes:

- **OFFSET** describes how many bytes should be skipped from the starting point of the current BLOCK.
- **REPEAT** describes how often the BLOCK should be read/written.
- **COUNT** number of bytes to read/write at each BLOCK operation.
- **STRIDE** describes the number of bytes to skip at each BLOCK operation.

Example of a regular distributed file onto 2 servers. The definition on server 1

```
<BLOCK OFFSET="0" REPEAT="3" COUNT="5" STRIDE="7">
    <BYTEBLOCK/>
</BLOCK>
```

corresponds to the definition on server 2:

```
<BLOCK OFFSET="5" REPEAT="3" COUNT="7" STRIDE="5">
    <BYTEBLOCK/>
</BLOCK>
```

### xDGDL examples

The following three examples show several possibilities that the xDGDL description provides. To depict the mapping between the internal structure and the xDGDL description two figures are attached to each example. The first figure shows a graphical tree representation of the underlying XML structure and the second figure the data distributed onto different servers.

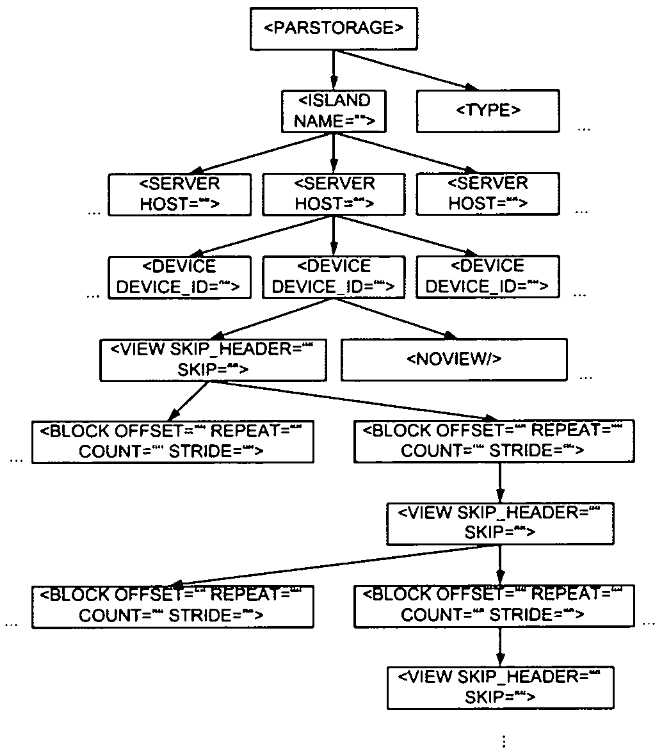

Figure 6.1: Example of a xDGDL tree

**A regularly distributed, two-server example** The first example introduces the structure of the xDGDL description. It uses two servers and writes data in round robin fashion to the local disks on each server: adios.tuwien.ac.at and clus9.tuwien.ac.at.

It is also possible to use more than one block. We would call this an interleaved distribution. The interleaved distribution divides the file into two parts. The first part is distributed on block one on server one and block one on server two. The second part is distributed on block two on server one and block two on server two.

The finer the granularity of the distribution gets, the more complex the structure grows.[1]

We suppose that server one writes more data to the disk. The factor is 5:7. (Please note it is an artificial example of minor practical relevance!)

The xDGDL representation of the regular, two-server example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PARSTORAGE SYSTEM "XDGDL.dtd">
<PARSTORAGE VERSION="1.0" TIMESTAMP="testfile_regular">
 <TYPE>
  <ETYPE TYPE="CHAR" LENGTH="1"/>
 </TYPE>
 <ISLAND NAME="island1.tuwien.ac.at">
  <SERVER HOST="adios.tuwien.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="7">
     <BLOCK OFFSET="0" REPEAT="3"
            COUNT="5" STRIDE="7">
      <BYTEBLOCK/>
     </BLOCK>
    </VIEW>
   </DEVICE>
  </SERVER>
```

[1]Beside this it is not wise to use a fine granularity for small files as the overhead of parsing the descriptor gets to large. In case of small files it would also lead to the situation that the description file is probably bigger than the files to write.

```
  <SERVER HOST="adclus9.tuwien.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="0">
     <BLOCK OFFSET="5" REPEAT="3"
            COUNT="7" STRIDE="5">
      <BYTEBLOCK/>
     </BLOCK>
    </VIEW>
   </DEVICE>
   </SERVER>
 </ISLAND>
</PARSTORAGE>
```

A graphical view of the regular distributed, two server example can be seen in Figure 6.2

**A regular distributed, nested three-server example** The last example handles three server. Beside the extension to three servers it is also the one that shows a nested description. The recursion depth itself is not limited.

The nested description gives the user an unrestricted flexibility to express any data distribution.

The xDGDL description of a regular distributed, nested three-server distribution:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PARSTORAGE SYSTEM "XDGDL.dtd">
<PARSTORAGE VERSION="1.0" TIMESTAMP="regular_multilevel">
 <TYPE>
  <ETYPE TYPE="CHAR" LENGTH="1"/>
 </TYPE>
 <ISLAND NAME="island3.tuwien.ac.at">
  <SERVER HOST="adios.tuwien.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="12">
     <BLOCK OFFSET="0" REPEAT="2"
            COUNT="1" STRIDE="12">
      <VIEW SKIP_HEADER="0" SKIP="0">
```

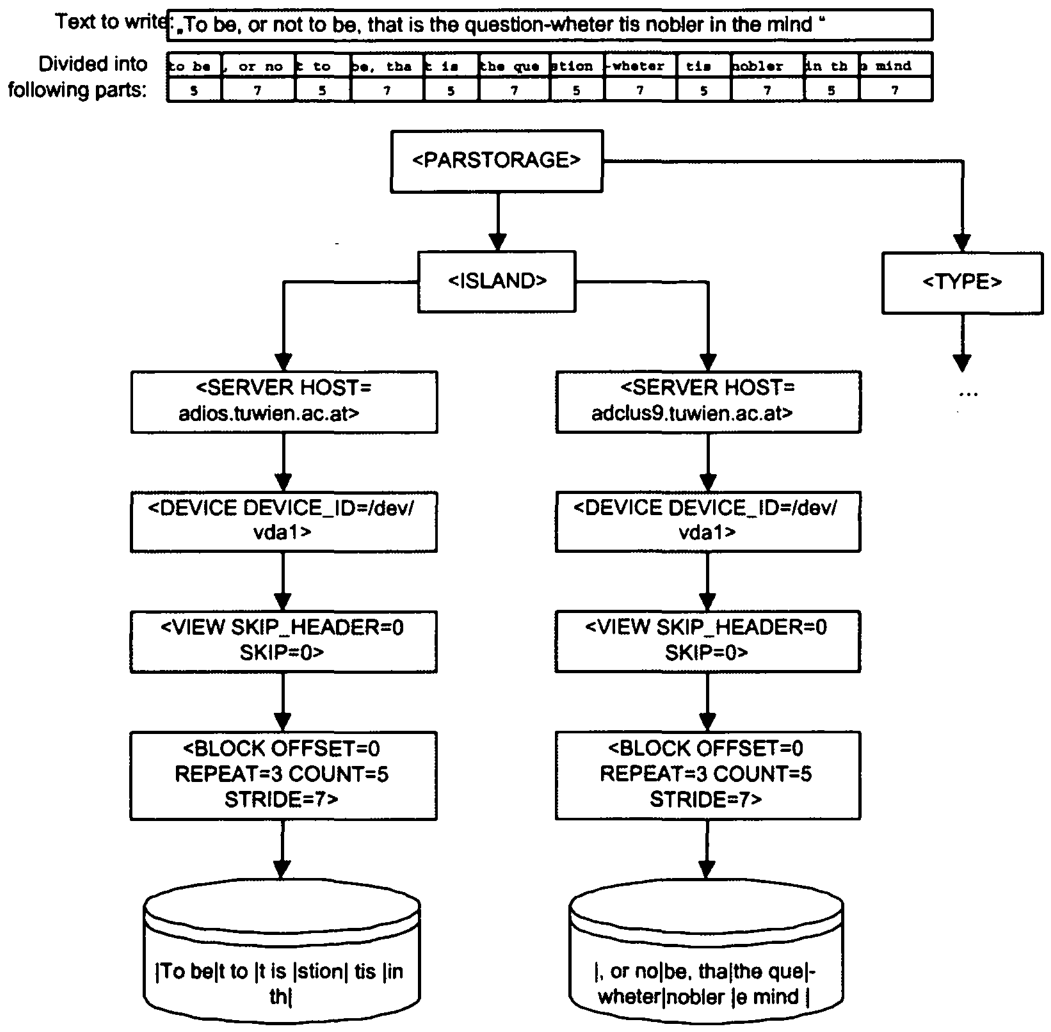
Text to write: „To be, or not to be, that is the question-wheter tis nobler in the mind "
Divided into following parts:
to be
, or no
t to
be, tha
t is
the que
stion
-wheter
tis
nobler
in th
e mind
5
7
5
7
5
7
5
7
5
7
5
7
<PARSTORAGE>
<ISLAND>
<TYPE>
...
<SERVER HOST= adios.tuwien.ac.at>
<SERVER HOST= adclus9.tuwien.ac.at>
<DEVICE DEVICE_ID=/dev/ vda1>
<DEVICE DEVICE_ID=/dev/ vda1>
<VIEW SKIP_HEADER=0 SKIP=0>
<VIEW SKIP_HEADER=0 SKIP=0>
<BLOCK OFFSET=0 REPEAT=3 COUNT=5 STRIDE=7>
<BLOCK OFFSET=0 REPEAT=3 COUNT=5 STRIDE=7>
|To be|t to |t is |stion| tis |in th|
|, or no|be, tha|the que|-wheter|nobler |e mind |

Figure 6.2: Tree representation of a regular distributed, two-server xDGDL distribution

```
       <BLOCK OFFSET="0" REPEAT="3"
              COUNT="5" STRIDE="7">
        <BYTEBLOCK/>
       </BLOCK>
      </VIEW>
     </BLOCK>
    </VIEW>
   </DEVICE>
  </SERVER>
  <SERVER HOST="adclus9.tuwien.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="12">
     <BLOCK OFFSET="0" REPEAT="2"
            COUNT="1" STRIDE="12">
      <VIEW SKIP_HEADER="0" SKIP="0">
       <BLOCK OFFSET="5" REPEAT="2"
              COUNT="7" STRIDE="12">
        <BYTEBLOCK/>
       </BLOCK>
      </VIEW>
     </BLOCK>
    </VIEW>
   </DEVICE>
  </SERVER>
  <SERVER HOST="adclus10.tuwien.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="0">
     <BLOCK OFFSET="29" REPEAT="2"
            COUNT="12" STRIDE="29">
      <BYTEBLOCK/>
     </BLOCK>
    </VIEW>
   </DEVICE>
  </SERVER>
 </ISLAND>
</PARSTORAGE>
```

A graphical view of the regular distributed, nested three-server example can be seen in Figure 6.3

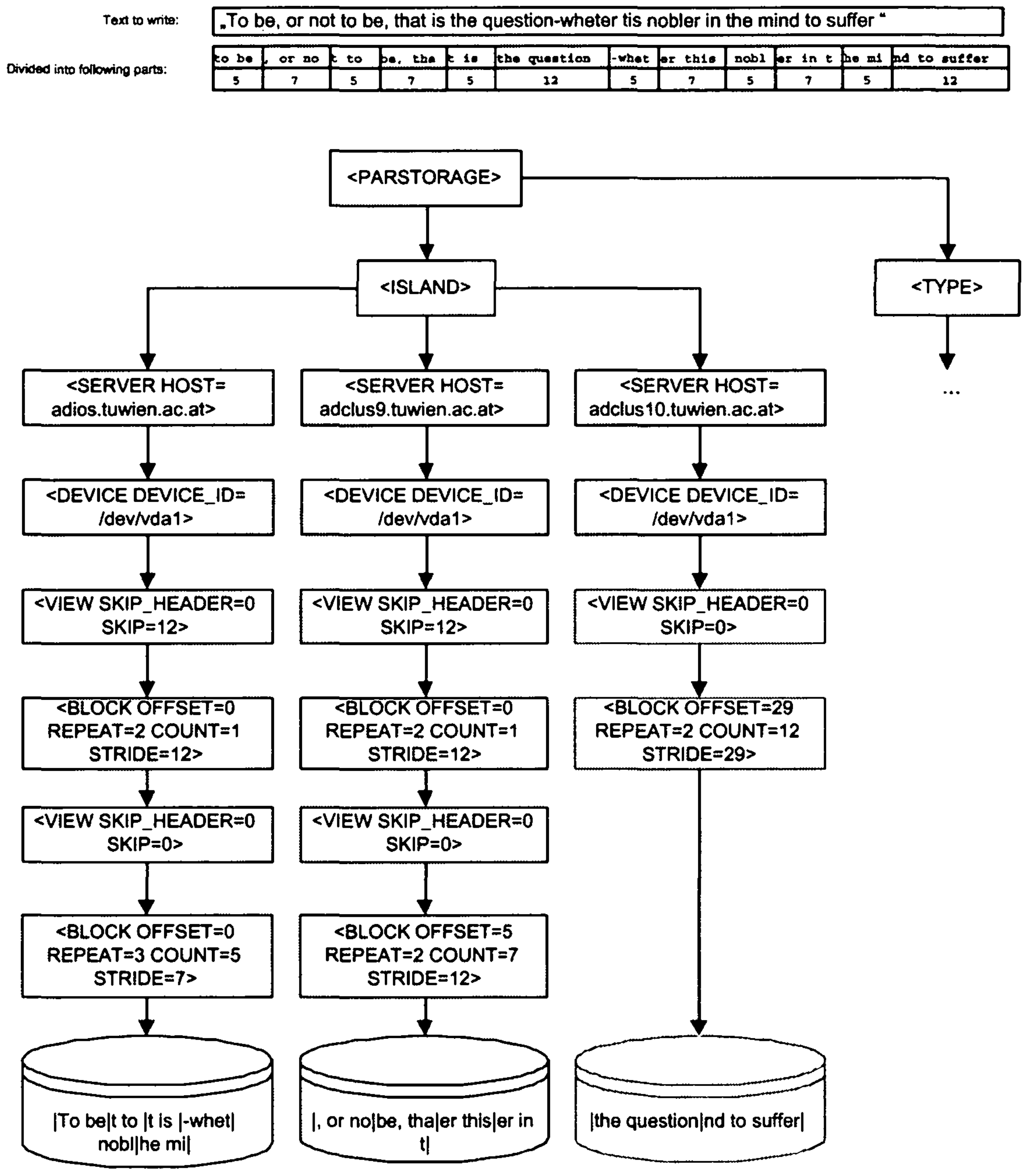
Text to write:
„To be, or not to be, that is the question-wheter tis nobler in the mind to suffer “
Divided into following parts:
to be | , or no | t to | be, tha | t is | the question | -whet | er this | nobl | er in t | he mi | nd to suffer
5 | 7 | 5 | 7 | 5 | 12 | 5 | 7 | 5 | 7 | 5 | 12
<PARSTORAGE>
<ISLAND>
<TYPE>
...
<SERVER HOST= adios.tuwien.ac.at>
<SERVER HOST= adclus9.tuwien.ac.at>
<SERVER HOST= adclus10.tuwien.ac.at>
<DEVICE DEVICE_ID= /dev/vda1>
<DEVICE DEVICE_ID= /dev/vda1>
<DEVICE DEVICE_ID= /dev/vda1>
<VIEW SKIP_HEADER=0 SKIP=12>
<VIEW SKIP_HEADER=0 SKIP=12>
<VIEW SKIP_HEADER=0 SKIP=0>
<BLOCK OFFSET=0 REPEAT=2 COUNT=1 STRIDE=12>
<BLOCK OFFSET=0 REPEAT=2 COUNT=1 STRIDE=12>
<BLOCK OFFSET=29 REPEAT=2 COUNT=12 STRIDE=29>
<VIEW SKIP_HEADER=0 SKIP=0>
<VIEW SKIP_HEADER=0 SKIP=0>
<BLOCK OFFSET=0 REPEAT=3 COUNT=5 STRIDE=7>
<BLOCK OFFSET=5 REPEAT=2 COUNT=7 STRIDE=12>
|To be|t to |t is |-whet| nobl|he mi|
|, or no|be, tha|er this|er in t|
|the question|nd to suffer|

Figure 6.3: Tree representation of a regular distributed, nested three-server xDGDL distribution

### 6.2.4 An Application of xDGDL

#### The ADIOS island

ADIOS - the Autonomous Distributed Input Output System - is an I/O system that tries to solve the well-known I/O bottleneck of high-performance computing [61]. ADIOS was originally designed as a client-server system satisfying parallel I/O needs of high performance applications. Due to the requirements of the Datagrid initiative ADIOS was extended to Meta-ADIOS, which harnesses distributed I/O resources [33].

A *ADIOS island* (resembling roughly a collaboration within our Grid architecture) can be seen as a logically independent system, residing on a defined set of processing nodes. Conventionally this is a typical cluster system, but it can also be an arbitrary set of world-wide distributed machines. An island comprises an arbitrary number of ADIOS servers processing the I/O requests of connected applications. To reach such an island the client needs to know the hostname (or IP-address) of a dedicated connection server responsible for that island (for more information see [60]).

An island provides several interfaces; beside the native interface, an MPI-IO interface (ADMPIOS), a HPF/VFC (Vienna Fortran Compiler) interface as well as a Unix file access interface (ADFS) are supported.

The system defines two modes to describe the distribution of a file. By default the automatic modes allows ADIOS to decide how to distribute the given file among the available servers. The user guided modus in contrast let the user decide how to distribute the file. In this modus a xDGDL file describes the distribution of a given file.

ADIOS provides a data independent view of the stored data to the application process. It is based on a three-tier model. The three specific ADIOS layers are the

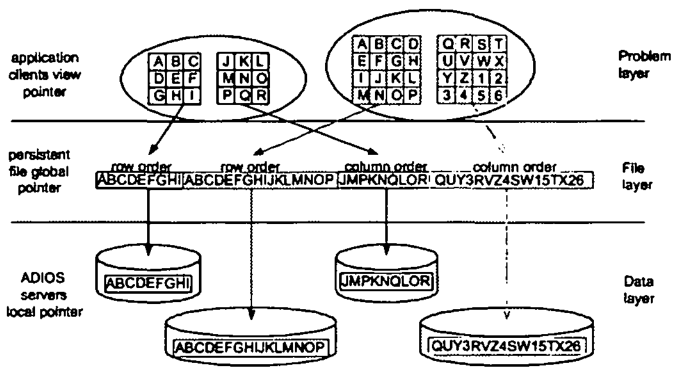

Figure 6.4: Different point of views: The ADIOS layers

following (see Figure 6.4):

- **Problem layer.** Defines the problem specific data distribution among the cooperating parallel processes (View file pointer).
- **File layer.** Provides a composed view of the persistently stored data in the system (Global file pointer).
- **Data layer.** Defines the physical data distribution among the available disks (Local file pointer).

The three tier architecture allows ADIOS to be completely logical data independent between the problem and the file layer as well as to be physical independent between the file and data layer.

## The ADIOS interfaces

ADIOS provides a range of interfaces to support a wide variety of applications. The interfaces are supported by interface modules to allow flexibility and extendibility. Up to now we implemented the following modules:

- HPF/VFC - High Performance Fortran interface based on the Vienna Fortran compiler
- ADMPIOS - a MPI-IO interface
- ADFS - ADIOS distributed file system
- ADIOS proprietary interface for some specialized modules

In the context of this paper we concentrate on the novel ADFS, that allows both the casual and the experienced user to use ADIOS in form of a distributed file system.

**ADFS**

Basically ADFS is a library which overloads the standard file calls in UNIX. This methods allows users easily and efficiently to employ transparently services provided by ADIOS. Thus all Unix tools for file accesses can be used without recompiling. The idea is to redirect the calls with "conventional" data files to the standard I/O library and to redirect the calls with ADFS data files to the ADIOS system. This approach is similar to PVFS [49].

Beside the overloaded Unix interface ADFS also provides a C-Interface, which can be linked with C-programs. This interface provides nearly the same functionality as the standard I/O interface.

For users it is very easy to define the meta information for the data file in focus. A respective xDGDL file has to be created and stored in the same directory as the data file, which has the same name as the data file, but with the prefix "`.vd.`"[2]. With an open statement the ADFS library checks if there is a corresponding xDGDL file for

[2]The prefix stands for *ADIOS description*

the given file. The prefixed dot is used because these files are not visible with the common `ls` command. It is also quite common to use the dot for configuration files and to a certain extent the ".vd.*" files can be seen as configuration files. When it is parsed, its is checked against the given data type definition (DTD). If the file is erroneous or does not exist the respective data file will be distributed with the standard distribution of ADFS which is a cyclic distribution among the available ADIOS servers.

**Copy Example** The copy command is a simple example to show the transparent usage of the ADFS file system. In this example it is the intent to copy a data file from a convention Unix file system to ADFS and back.

The preconditions for using ADFS are the following:

- Start of ADIOS
- Configuration of the ADIOS configuration file (`ADIOS.conf`) that was set up in the environment. In our example we used:

```
MAX_APP 5 MAX_SRV_FILE 32 DATA_BUFLEN 4096 SRV_GROUP_NAME
"adios_server" SRVR_DEVICE_LIST 3
  /home/fuerle/ADIOS/dev1/
  /home/fuerle/ADIOS/dev2/
  /home/fuerle/ADIOS/dev3/
ADIOS_DIR "/home/fuerle/adios"
```

- Setting of Unix environment variable that points to the ADIOS configuration file

```
fuerle@adios:~/adfstests > ls -al .vd.*
-rw-r-----   1 fuerle   users  1177 Oct 14  2001 .vd.testfile

fuerle@adios:~/adios > cp testfile /home/fuerle/adios   # copy in
fuerle@adios:~/adios > cp /home/fuerle/adios/testfile . # copy out

fuerle@adios:~/adios > ls -l /home/fuerle/adios
total 0
-rw-r--r--   1 fuerle   users   0   Oct 14  2001 testfile
```

Figure 6.5: ADFS copy of a data file

(e.g. `ADIOS_CONF=/home/fuerle/adios/ADIOS.conf`). The environment could be set up with the command `export`.

- Setting up the `LD_PRELOAD` environment variable. The variable must point to the `adfsinvoke.so` shared object. In our example we set it up as follows: `export LD_PRELOAD=/home/fuerle/adfs/adfsinvoke.so`

After these steps the ADFS can be used similar to an NFS mounted device. The user uses standard Unix calls only for writing and reading files. Internally all I/O calls on the specified directory (ADIOS_DIR) are passed to the ADFS library. Therefore all the Unix commands that use the standard I/O calls can be used with ADFS.

In case of the example above the user can copy a data file simply by the commands shown in figure 6.2.4

As we did not overload the `ls` command the user can only see a file with 0 bytes within the ADIOS_DIR. This is due to the fact that the file is not really copied into the directory. For transparency to the user ADFS generates a 0-byte file to provide the user with the information which files are currently distributed on the system.

In the first line we print out all .vd.* files. In our example only one distribution file is present. We used the distribution file presented in 6.3. That means, that the testfile was distributed among three servers with one device on each server. If we did not declare a .vd. file the testfile would have been written sequentially to the first disk on the current server.

### 6.2.5 Conclusion

We presented xDGDL, an XML language for storing meta information for distributed files on the Grid. The proposed XML approach acts in the system in two ways; on one hand it provides a user interface to specify the contents (semantical information) and the layout (physical information) of the file, on the other hand it is the expressive mechanism within the system to administer the distribution information of the files stored in the file system across several sites on the Grid. We showed a practical prove-of-concept implementation by the ADFS distributed file system.

## Appendix: xDGDL DTD

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT PARSTORAGE
  (PROCESSORS*,TYPE+,ALIGN*,ISLAND)>
<!ATTLIST PARSTORAGE VERSION CDATA #REQUIRED>
<!ATTLIST PARSTORAGE TIMESTAMP ID #REQUIRED>

<!-- processors -->
<!ELEMENT PROCESSORS (PROC_DIMENSION)+>
<!ATTLIST PROCESSORS NAME CDATA #REQUIRED>
<!ELEMENT PROC_DIMENSION EMPTY>
<!ATTLIST PROC_DIMENSION LOWER CDATA "1">
<!ATTLIST PROC_DIMENSION UPPER CDATA #REQUIRED>


<!-- hpf data structure -->
<!-- Intrinsic Data Types -->
```

```
<!ELEMENT TYPE (ETYPE|ARRAY|TYPE)+>
<!ATTLIST TYPE TYPENAME CDATA #IMPLIED>
<!ATTLIST TYPE NAME CDATA #IMPLIED>

<!ELEMENT ETYPE EMPTY>
<!ATTLIST ETYPE TYPE CDATA #REQUIRED>
<!ATTLIST ETYPE LENGTH CDATA #REQUIRED>
<!ATTLIST ETYPE NAME CDATA #IMPLIED>

<!-- Arrays -->
<!ELEMENT ARRAY (TYPE, DIMENSION+)>
<!ATTLIST ARRAY NAME CDATA #IMPLIED>
<!ATTLIST ARRAY MAJOR (ROW|COLUMN) "ROW">
<!ATTLIST ARRAY DISTRIBUTE_ONTO CDATA #IMPLIED>

<!ELEMENT DIMENSION EMPTY>
<!ATTLIST DIMENSION LOWER CDATA "1">
<!ATTLIST DIMENSION UPPER CDATA #REQUIRED>
<!ATTLIST DIMENSION DISTRIBUTE
  (BLOCK|CYCLIC|NO) #IMPLIED>
<!ATTLIST DIMENSION DIST_SKALAR CDATA "1">


<!-- Alignment -->
<!ELEMENT ALIGN EMPTY>
<!ATTLIST ALIGN WHAT CDATA #REQUIRED>
<!ATTLIST ALIGN WITH CDATA #REQUIRED>


<!-- data distribution in this file -->
<!-- Model Island-Descriptor -->
<!ELEMENT ISLAND (SERVER*)>
<!ATTLIST ISLAND NAME CDATA #REQUIRED>

<!-- Model Server-Descriptor -->
<!ELEMENT SERVER (DEVICE*)>
<!ATTLIST SERVER HOST CDATA #REQUIRED>

<!-- Model Device-Descriptor -->
<!ELEMENT DEVICE (VIEW|NOVIEW)>
```

```
<!ATTLIST DEVICE DEVICE_ID CDATA #REQUIRED>

<!-- Model Access-Descriptor -->
<!ELEMENT VIEW (BLOCK+)>
<!ATTLIST VIEW SKIP_HEADER CDATA #REQUIRED>
<!ATTLIST VIEW SKIP CDATA #REQUIRED>

<!ELEMENT BLOCK (VIEW|BYTEBLOCK)>
<!ATTLIST BLOCK OFFSET CDATA #REQUIRED>
<!ATTLIST BLOCK REPEAT CDATA #REQUIRED>
<!ATTLIST BLOCK COUNT CDATA #REQUIRED>
<!ATTLIST BLOCK STRIDE CDATA #REQUIRED>
<!ELEMENT BYTEBLOCK EMPTY>
```

# Chapter 7

# Practical Tests with ADIOS

## 7.1 Introduction

Before we show the actual performance data, we should discuss what maximum data we can achieve. We used standard PCs with a AMD CPU of 1600 MHz, 256 MB RAM, Gigabit Network Card D-Link DGE-500T on a 64 Bit PCI slot and a 1GBit switch, which supported only the standard frame MTU size (1500 bytes). So we started with the measurement of the raw network speed under different circumstances. For this purpose we used NetPipe [24].

We switched the network speed between 100 MBit/s and 1 GBit/s and measured on the one hand the resulting transfer rate for TCP traffic and on the other hand the transfer rate for PVM to get an idea of the overhead of the underlying message passing system of ADIOS. The transfer rate for PVM is our theoretical maximum for ADIOS.

| blocksize (KB) | transfer rate (Mbit/s) (MBit/s) | transfer rate TCP (MBit/s) | transfer rate PVM (MBit/s) | MTU (Bytes) |
|---|---|---|---|---|
| 4K | 100 | 64.0 | 36.0 | 1500 |
| 4K | 1000 | 78.0 | 78.0 | 1500 |
| 64K | 100 | 100.0 | 71.0 | 1500 |
| 64K | 1000 | 300.0 | 173.0 | 1500 |

We notice, that a blocksize of 64KByte is enough to saturate the 100MBit/s network. But surprisingly, the switch from 100MBit/s to 1GBit/s only triples the transfer rate instead of ten times. I.e the 1 GBit network is only utilized to a third. So we were looking for other possibilities to increase the network speed.

**Excursus: MTU (maximum transfer unit)** Current TCP networks use 1500 bytes for frame MTU sizes. New switch technology supports so called jumbo frames which range up to 9000 bytes. This settings are necessary for harnessing the full range of a Gigabit LAN (up to a 1GBit/s). Unfortunately our switch supported only a frame MTU size of 1500 bytes. To measure the network speed with a frame MTU size of 3000 bytes for comparison we used crossed jumper cables. Below are the results

| blocksize (KB) | transfer rate (MBit/s) | transfer rate TCP (MBit/s) | transfer rate PVM (MBit/s) | MTU (Bytes) |
|---|---|---|---|---|
| 64K | 1000 | 300.0 | 173.0 | 1500 |
| 64K | 1000 | 400.0 | 213.0 | 3000 |

We notice, that the frame MTU size has a massive impact on the network performance, i.e for setting up such an environment it is necessary to work with switches which support jumbo frames (frame size up to 9000 bytes) to harness almost the full network speed of Gigabit. Details on this issue can be found in [8].

**Excursus: Zero Copy Protocols** [53] Despite technological advantages in microprocessors and network technology over the last few years, commercially-available networks of workstations (NOWs [3]) contain inherent communication bottlenecks. Traditional layered network protocols will inevitably fail to achieve high throughput if they access data several times. As a result, applications on NOWs often fail to observe the performance speed-up that might be expected. Network protocols which avoid routing through the kernel can remove this limit on communication performance and support very high transmission speeds, turning NOWs into an attractive alternative to Massively-Parallel Processors.This kind of protocols improve the performance of an entire NOW system, but this needs to be supported by the message passing software. There are some popular hardware vendors like Myrinet [57] or Dolphin [42], which harness this technology. Myrinet supports PVM and reaches about 1853 MBit/s for TCP transfer rate, which is 5 times faster than plain Gigabit (see above) and also a small latency time of 32us, which is 2 to 5 times smaller than common Gigabit cards compare with [17].

Unfortunately, we didn't had that special hardware and even no jumbo frame aware switch, so for the sake of clarity we must live with the perception that our theoretical maximum transfer rate is 300Mbit/s for TCP and 173MBit/s for PVM. Figure 7.1 and figure 7.2 summarize these results.

## 7.2 The native system interface

To measure the performance of ADIOS, we wrote ad_cp, which works rather similar to the well known UNIX cp. See following example:

```
time ad_cp -v testfile.100M /home/fuerle/adios/testfile.100M
```

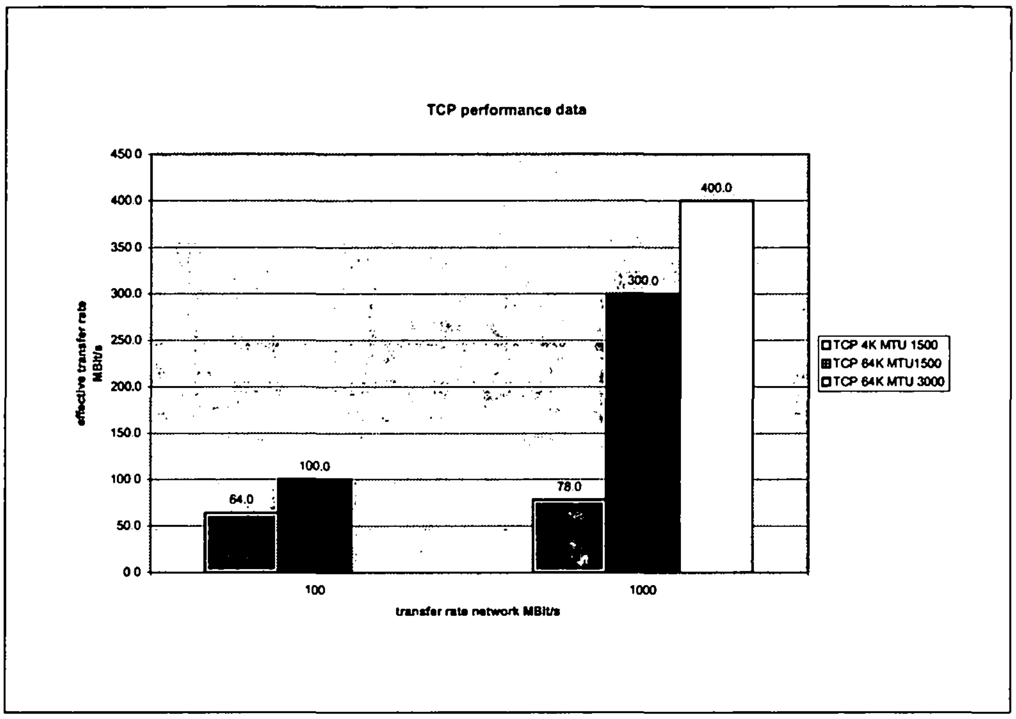

Figure 7.1: TCP performance data

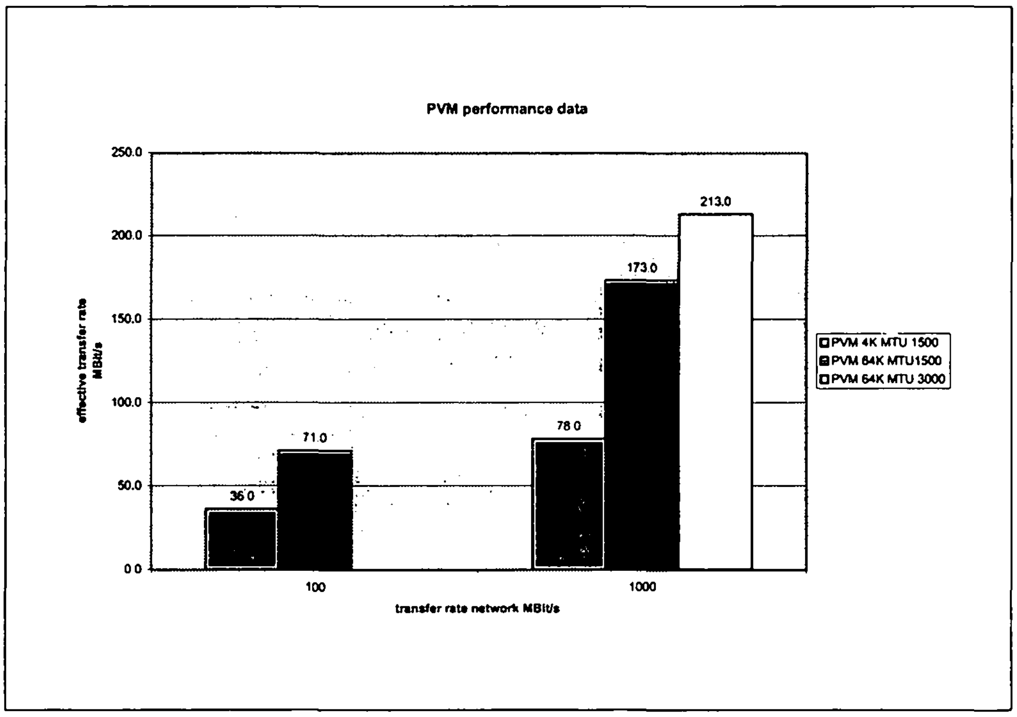

Figure 7.2: PVM performance data

This command takes testfile.100M, reads its distribution file .vd.testfile.100M and distributes it regarding to the entries there. The path /home/fuerle/adios/ tells ADIOS, that it should take care of this file (this path is defined in the environment file ADIOS.conf, which is sourced by the shell, see also 4.5.2). The time command measures the overall runtime of this process in seconds, i.e less means faster performance. Below are the results:

| Write: time ad_cp -v testfile.100M /home/fuerle/adios/testfile.100M<br>Read: time ad_cp -v /home/fuerle/adios/testfile.100M testfile.100M | | | | | | |
|---|---|---|---|---|---|---|
| blocksize (KB) | transfer rate (MBit/s) | write (sec.) | read (sec.) | server count | data (MB) | MTU (Bytes) |
| 64K | 100 | 27.8 | 23.9 | 1 | 100M | 1500 |
| 64K | 100 | 55.2 | 53.9 | 1 | 200M | 1500 |
| 64K | 100 | 109.0 | 110.0 | 1 | 400M | 1500 |
| 64K | 100 | 218.0 | 217.0 | 1 | 800M | 1500 |
| 64K | 1000 | 18.9 | 14.3 | 1 | 100M | 1500 |
| 64K | 1000 | 39.4 | 36.8 | 1 | 200M | 1500 |
| 64K | 1000 | 74.0 | 73.0 | 1 | 400M | 1500 |
| 64K | 1000 | 148.0 | 143.0 | 1 | 800M | 1500 |
| 64K | 1000 | 15.5 | 14.2 | 1 | 100M | 3000 |
| 64K | 1000 | 31.8 | 35.7 | 1 | 200M | 3000 |
| 64K | 1000 | 64.0 | 72.0 | 1 | 400M | 3000 |
| 64K | 1000 | 127.0 | 135.0 | 1 | 800M | 3000 |

| Write: time ad_cp -v testfile.100M /home/fuerle/adios/testfile.100M<br>Read: time ad_cp -v /home/fuerle/adios/testfile.100M testfile.100M | | | | | | |
|---|---|---|---|---|---|---|
| blocksize (KB) | transfer rate (MBit/s) | write (sec.) | read (sec.) | server count | data (MB) | MTU (Bytes) |
| 64K | 100 | 22.5 | 18.4 | 2 | 100M | 1500 |
| 64K | 100 | 44.8 | 35.6 | 2 | 200M | 1500 |
| 64K | 100 | 86.0 | 82.0 | 2 | 400M | 1500 |
| 64K | 100 | 176.0 | 162.0 | 2 | 800M | 1500 |
| 64K | 1000 | 16.6 | 12.4 | 2 | 100M | 1500 |
| 64K | 1000 | 35.1 | 24.9 | 2 | 200M | 1500 |
| 64K | 1000 | 68.0 | 58.0 | 2 | 400M | 1500 |
| 64K | 1000 | 135.0 | 116.0 | 2 | 800M | 1500 |

### 7.2.1 Network speed settings and Scaleup

Figure 7.3 for writing and figure 7.4 for reading shows similar results to the basic tests. The performance increases massively from 100MBit to 1GBit. To clarify, if we are still network bounded, we doubled the MTU size from 1500 bytes to 3000 bytes and got another nice performance gain. For our remaining tests we kept the MTU size at 1500 bytes because of the restrictions mentioned in 7.1 and continued only with 1GBit network speed, which results in transfer rate of 10.8MByte/s (1600MByte/148sec.). This results also shows, that we have an constant scaleup, i.e the overall runtime increases linear with the data size.

### 7.2.2 Speedup

Figure 7.5 for writing and figure 7.6 for reading shows results when switching from one to two servers. As the overall time decreases, this result shows clear speedup,

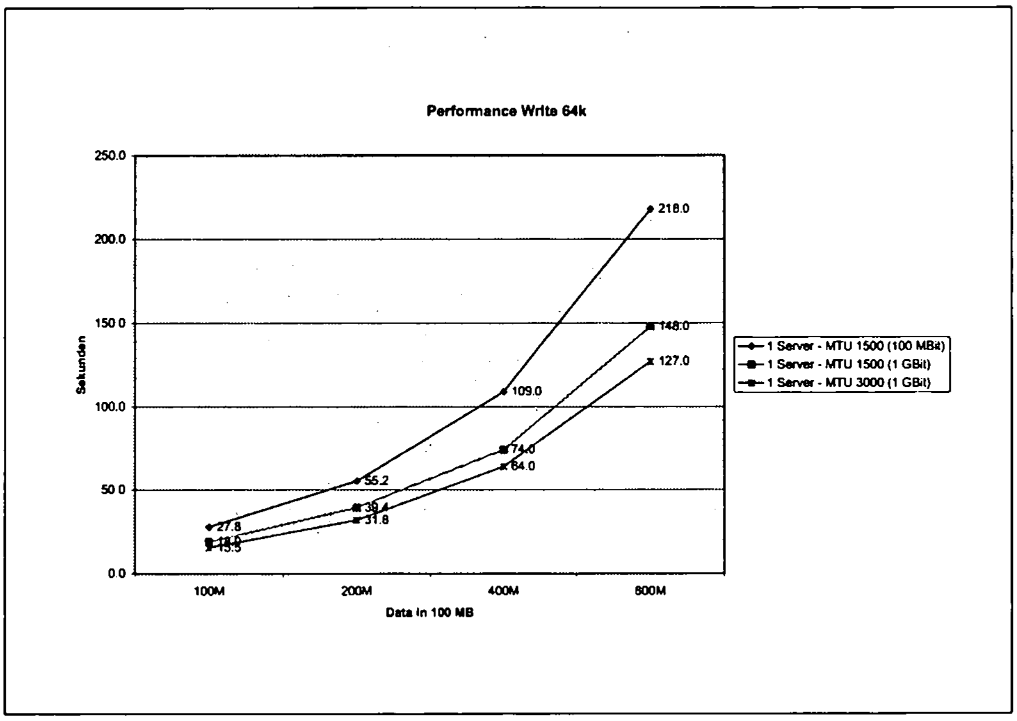

Figure 7.3: Performance write with different network settings

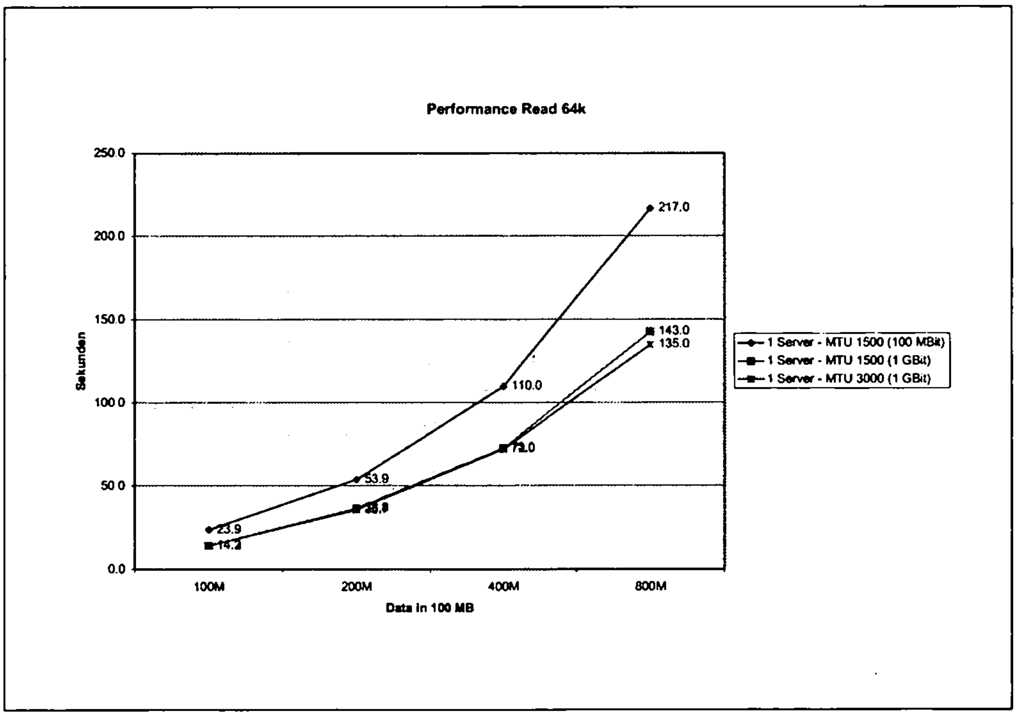

Figure 7.4: Performance read with different network settings

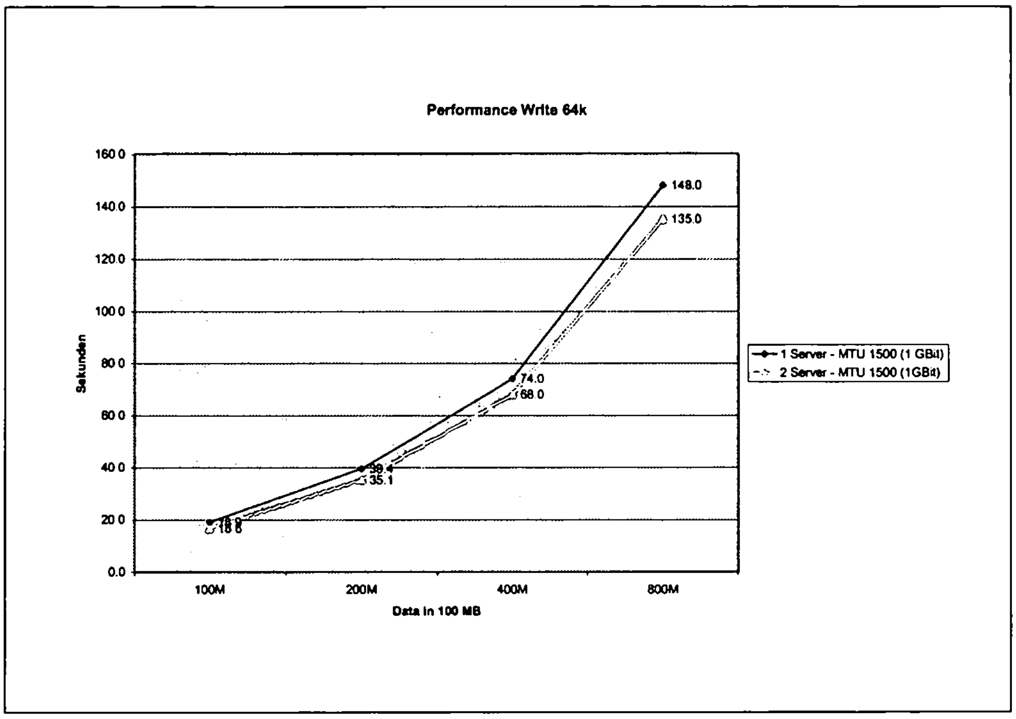

Figure 7.5: Performance write with one and two server

which proves this thesis, that clusters with a fast network can setup a distributed file system, which is faster than a single conventional file server with e.g. NFS.

### 7.2.3 Network traffic versus Payload

Based on this motivating results we are highly interested, how large is the influence of the network in those figures and so we made some tests where we first skipped the write part, so that we only read the file and then skipped the whole read and write part and reduced to the plain network traffic. Here are the results:

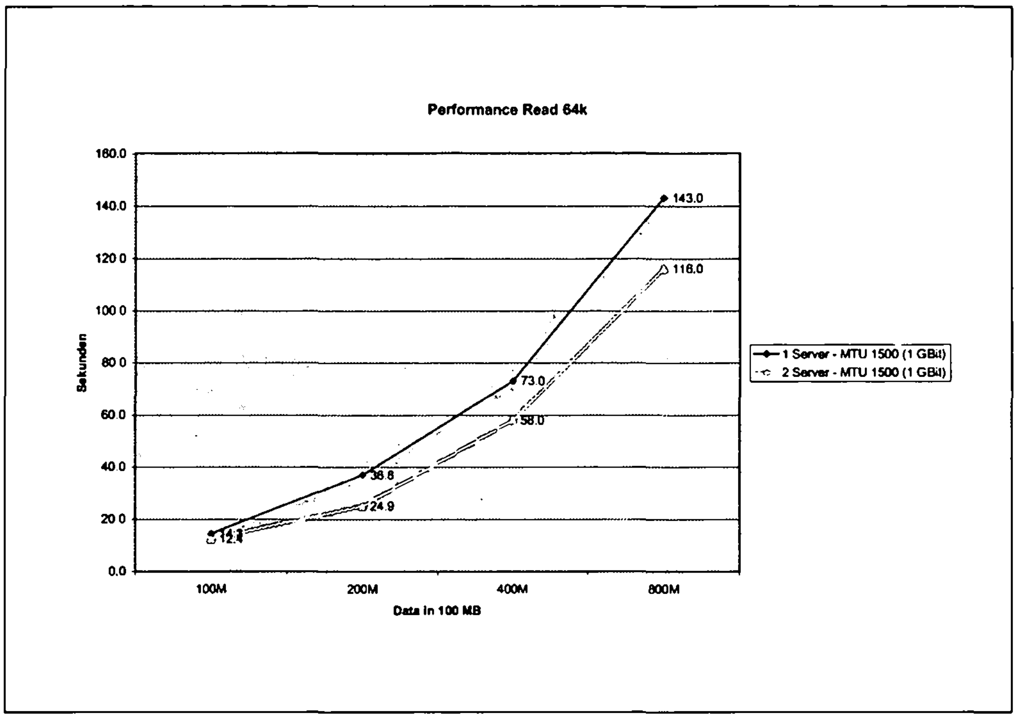

Figure 7.6: Performance read with one and two server

| blocksize (KB) | transfer rate (MBit/s) | write (sec.) | read (sec.) | server count | data (MB) | MTU (Bytes) | comment |
|---|---|---|---|---|---|---|---|
| 512K | 1000 | 15.3 | 11.8 | 1 | 100M | 1500 | no write |
| 512K | 1000 | 30.1 | 37.0 | 1 | 200M | 1500 | no write |
| 512K | 1000 | 60.0 | 77.0 | 1 | 400M | 1500 | no write |
| 512K | 1000 | 120.0 | 153.0 | 1 | 800M | 1500 | no write |
| 512K | 1000 |  | 11.5 | 1 | 100M | 1500 | network only |
| 512K | 1000 |  | 22.9 | 1 | 200M | 1500 | network only |
| 512K | 1000 |  | 45.8 | 1 | 400M | 1500 | network only |
| 512K | 1000 |  | 91.6 | 1 | 800M | 1500 | network only |

| blocksize (KB) | transfer rate (MBit/s) | write (sec.) | read (sec.) | server count | data (MB) | MTU (Bytes) | comment |
|---|---|---|---|---|---|---|---|
| 512K | 1000 | 13.9 | 6.9 | 2 | 100M | 1500 | no write |
| 512K | 1000 | 28.0 | 14.9 | 2 | 200M | 1500 | no write |
| 512K | 1000 | 52.0 | 37.0 | 2 | 400M | 1500 | no write |
| 512K | 1000 | 103.0 | 71.0 | 2 | 800M | 1500 | no write |
| 512K | 1000 |  | 6.8 | 2 | 100M | 1500 | network only |
| 512K | 1000 |  | 13.7 | 2 | 200M | 1500 | network only |
| 512K | 1000 |  | 26.2 | 2 | 400M | 1500 | network only |
| 512K | 1000 |  | 53.5 | 2 | 800M | 1500 | network only |

Figure 7.7 for writing and figure 7.8 for reading shows the gap between raw network traffic and reading and sending the payload (writing is skipped) with a larger block buffer size of 512KByte. The most impressive conclusion of this test is, that the network only part (compared to when reading and sending) is almost around 50 % of the overall runtime, which is much too high. See excursus 7.1, how to minimize this. On the other hand, the system still shows nice speedup and scaleup, i.e by reducing the network part could lead to an optimal system. Compared to theoretical maximum performance of 173MBit/s the two network results show also a convergence to this value.. With one server we have harnessed 70MBit/s or 40 % of the possible peak performance while with two servers we have increased the utilization to 120MBit/s or 70 %. So it can be assumed, that we can reach with a third server over 90 % utilization, which can be treated as a first approach to a local optimum. Unfortunately we had only three (which results in one client and two servers) boxes for testing available, so this hypothesis couldn't be verified.

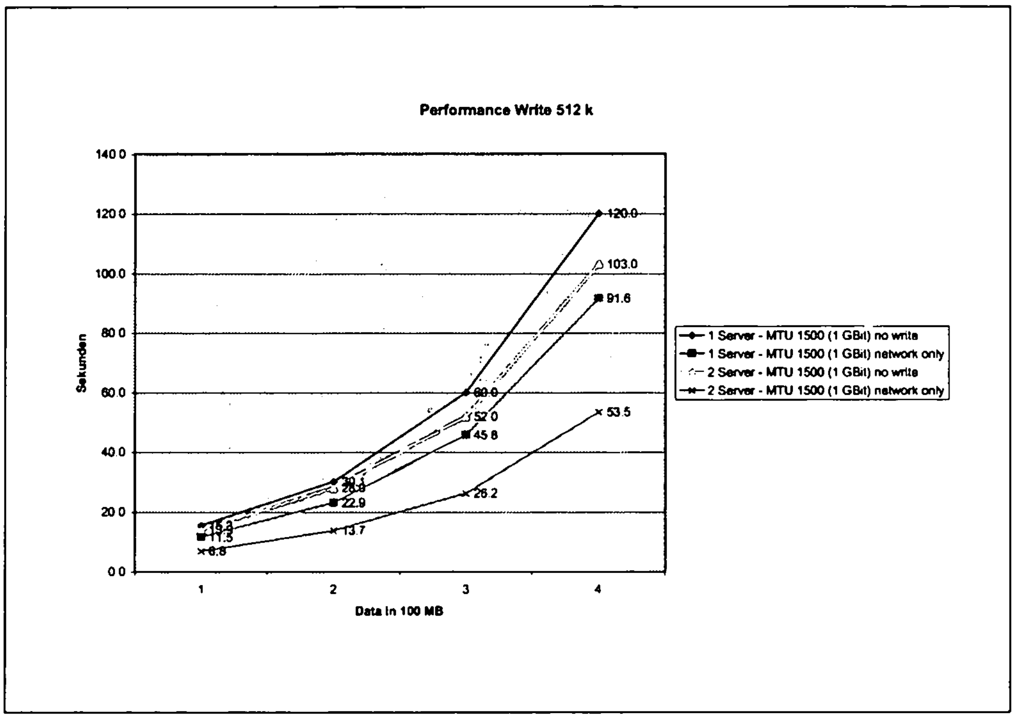

Figure 7.7: Performance write

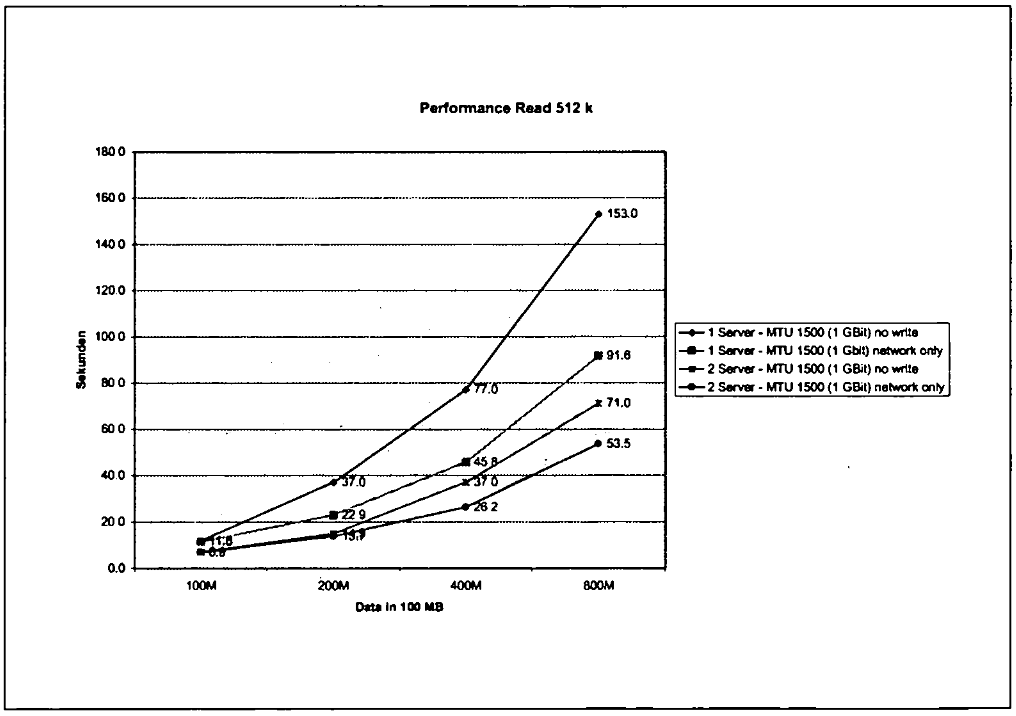

Figure 7.8: Performance read

### 7.2.4 ADIOS versus PVFS

Figure 7.9 for writing and figure 7.10 for reading shows ADIOS compared against his biggest competitor, namely PVFS. Both systems behave more or less equal performance. This is a very promising result, because it shows that a system without a centralized directory controller can work as sufficient than a comparable "commercial" system. Here are the results in detail:

| blocksize (KB) | transfer rate (MBit/s) | write (sec.) | read (sec.) | server count | data (MB) | MTU (Bytes) | comment |
|---|---|---|---|---|---|---|---|
| | 1000 | 11.8 | 20.7 | 2 | 100M | 1500 | PVFS |
| | 1000 | 32.0 | 38.2 | 2 | 200M | 1500 | PVFS |
| | 1000 | 58.5 | 76.0 | 2 | 400M | 1500 | PVFS |
| | 1000 | 98.0 | 182.0 | 2 | 800M | 1500 | PVFS |
| 512K | 1000 | 12.2 | 11.8 | 2 | 100M | 1500 | no write |
| 512K | 1000 | 36.2 | 27.7 | 2 | 200M | 1500 | no write |
| 512K | 1000 | 60.7 | 64.0 | 2 | 400M | 1500 | no write |
| 512K | 1000 | 115.0 | 178.0 | 2 | 800M | 1500 | no write |

## 7.3 The ADFS interface of ADIOS

As shown in 6.2 this interface enables a unix user to use ADIOS with the common unix tools like cp, mv, vi, etc. and it has only a minor overhead to the native ADIOS interface. But it has still a bad performance because the block size of the underlying operating system is only 4K, which directs into a very slow performance (see first measurements in 7.1). So we recommend to use ad_cp instead of the native unix cp in scripts or on the shell and be aware of the difference. Another recommendation

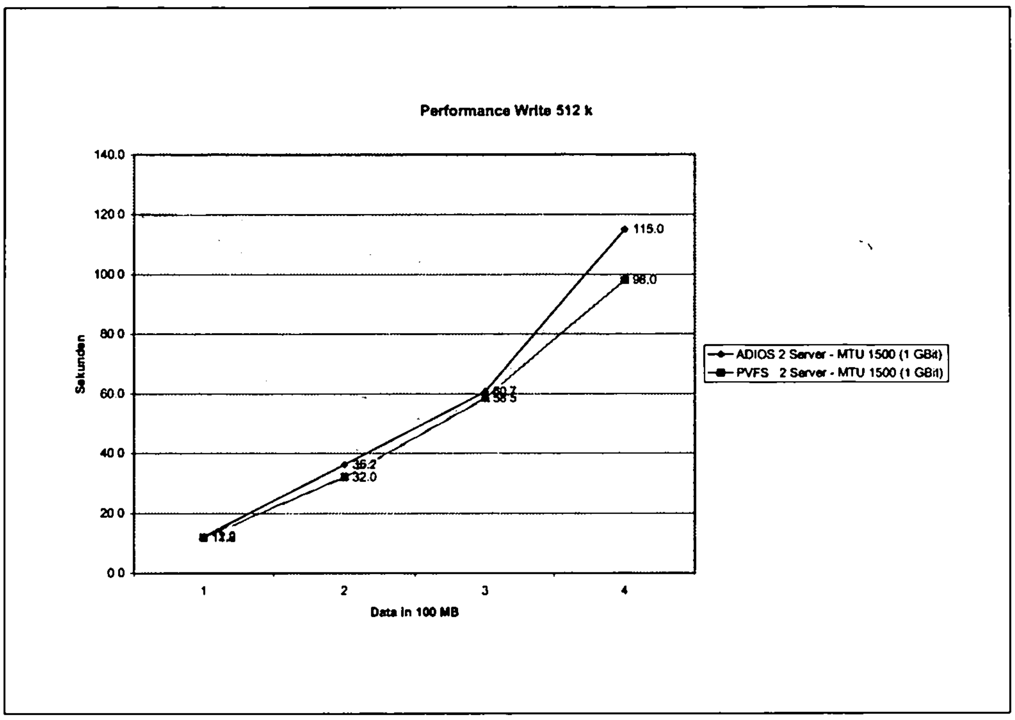

Figure 7.9: ADIOS write versus PVFS

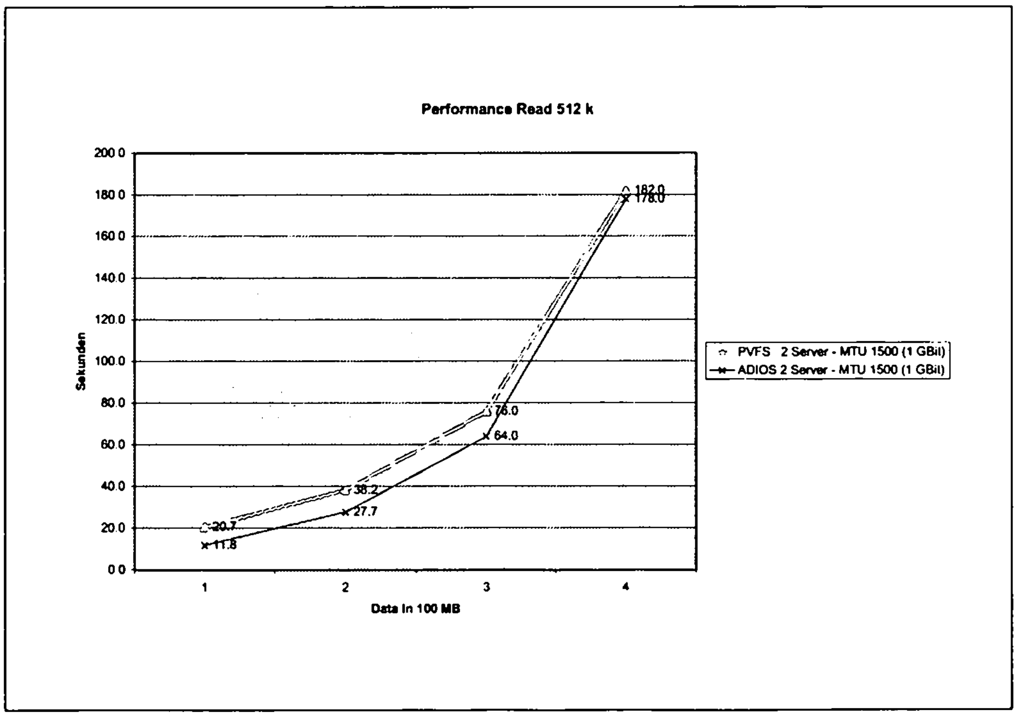

Figure 7.10: ADIOS read versus PVFS

would be to rewrite the programs to the ADIOS or MPI-IO interface (see below), if the user has source code access and performance is critical.

## 7.4 The MPI-IO interface of ADIOS

For this test we used the test programs, which are provided with ROMIO and compiled them against our MPI-IO Interface. The results didn't show any measurable difference to the native interface, i.e the overhead is negligible.

# Chapter 8

# Conclusions

## 8.1 Introduction

We have presented the design and implementation of ADIOS, an autonomous distributed raid system with a xml metadefinition language.

The growing popularity of parallel computers built from clusters of workstations has presented a number of challenges to file systems designers, which several research groups have tried to address with their file systems. ADIOS tries to satisfy the needs of all of this worlds with a best effort approach supporting e.g UNIX I/O as well as novel programming interface like distributed UNIX I/O with an underlying XML Meta language.

Nevertheless it supports basic file system features like sequential consistency, buffering, non blocking I/O and write behind by sitting on top of the native file system.

One of the basis ideas of ADIOS is to take the definition of RAID and extend it to a combination of multiple machines and even further to a group of multiple machines (see ADIOS Islands 5.2), but to do this on the file system level without any centralized directory server, but down to a very fine granularity (the user can even specify, how a single byte should be distributed).

The novel approaches compared to other available distributed systems are the

- *Autonomous Part.* We don't use any centralized directory servers (see 4.8 to find out, how we solved the major challenge). This is the central contribution of this thesis to the topic of Parallel I/O.

- *Failure Tolerance.* If a server fails, no operations are affected, as long as the failed server is not part of the requested data transfers. This is another main contribution of this thesis, which results out of the idea of the autonomous part.

Furthermore we have fulfilled all the requirements we have claimed in the beginning

- *Scalability.* Guarantees that the size of the used I/O system, i.e. the number of I/O nodes currently used to solve a particular problem, is defined by or correlated with the problem size. The *System Architecture* of ADIOS is highly distributed and decentralized. This leads to the advantage that the provided I/O bandwidth of ADIOS is mainly dependent on the available I/O nodes of the underlying architecture only. See the chapter 7 for details.

- *Efficiency.* The aim of optimization is to minimize the number of disk accesses for file I/O. This is achieved by a suitable data organization (section 3.5.1) by providing a transparent view of the stored data on disk to the 'outside world' and by organizing the data layout on disks respective to the static application problem description. The organization of this mapping is still the issue of the user, but he or she gets all necessary tools like the XML structure for distributing the data and the possibility to configure the system properly with the config file. And as it turned out in chapter 7, the system performed rather good, even compared to well known systems like PVFS.

- *Usability.* The application programmer must be able to use the system without big efforts. So she does not have to deal with details of the underlying hardware in order to achieve good performance and familiar *Interfaces* (section 3.5) are available to program file I/O.

- Use of widely accepted standards. ADIOS uses standards itself (e.g. PVM for the communication between clients and servers) and also offers standard interfaces to the user (for instance application programmers may use MPI-I/O or UNIX file I/O in their programs), which strongly enhances the systems portability and ease of use.

- *Portability.* The system is portable across multiple hardware platforms. This also increases the usability and therefore the acceptance of the system.

## 8.2 Conclusions

- *Network based distributed I/O systems do work.* Starting with the use of GigaBit networks our system does scale and shows speedups.
- *Autonomous based distributed I/O systems do work.* We don't use any centralized directory servers. This is the central contribution of this thesis to the topic of Parallel I/O.
- *No single point of failure.* If a server fails, no operations are affected, as long as the failed server is not part of the requested data transfers. This result is based on the idea of autonomous servers.
- *Heterogeneous systems are possible.* With the use of PVM, a protocol, which can operate on heterogeneous environments, our system can also work in heterogeneous environments transparent for the user .
- *GigaBit ethernet transfer rate with default settings is not effective.* It harnesses only a third of the available transfer rate. To increase this value, larger MTU sizes are needed, which must be supported by the network switches.
- *TCP protocol carries to much burden.* Network protocols which avoid routing through the kernel can remove this limit on communication performance and support very high transmission speeds, turning NOWs into an attractive alternative to Massively-Parallel Processors. This kind of protocols improve the performance of an entire NOW system, but this needs to be supported by the message passing software (PVM supports that).

## 8.3 Future Directions

Future directions for ADIOS could be

- *Full featured MPI-IO Interface.* We have only implemented a part of the MPI-IO Interface (which was necessary to run the test programs), a full featured MPI-IO Interface would round off our running prototype
- *Bullet proof System.* It would be interesting to turn our working prototype in a production system like PVFS and give it to the open source community for further development
- *Test Results with State of the Art Network Equipment and more Servers.* A lot of the results would have been much more interesting with network equipment like Myrinet or Jumbo frame enabled Switches and large amount of servers, especially how the overall performance would modify and where the real borders of the system are.

# Bibliography

[1] Sweeney Adam, Doug Doucette Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck, *Scalability in the xfs file system*, Proceedings of the Usenix Technical Conference, January 1996, pp. 1–14.

[2] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke, *Data management and transfer in high-performance computational grid environments*, Parallel Computing **28** (2002), no. 5, 749–771.

[3] Thomas E. Anderson, David E. Culler, and David A. Patterson, *A case for now (network of workstations)*, IEEE Micro, February 1995, pp. 54–64.

[4] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang, *serverless network file systems, describes berkeley's xfs file system for now*, ACM Transactions on Computer Systems, February 1996, pp. 41–79.

[5] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, and Udaya A. Ranawake, *Beowulf: A parallel workstation for scientific computation*, Journal of Parallel and Distributed Computing, June 1997, pp. 147–155.

[6] Andras Belokosztolszki, *An xml based language for meta information in distributed file systems*, Master's thesis, University of Vienna / ELTE University Budapest, 2000.

[7] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke, *GASS: A data movement and access service for wide area computing systems*, Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (Atlanta, GA), ACM Press, May 1999, pp. 78–88.

[8] Anthony Betz and Paul Gray, *Gigabit over copper evaluation*, http://www.syskonnect.com/syskonnect/performance/gig-over-copper.htm, April 2002.

[9] Nassem Bhatti, Jean-Marie Le Goff, Hassan Waseem, Zsolt Kovacs, Richard Martin, Peter McClatchey, Heinz Stockinger, and Ian Willers, *Object serialisation and deserialisation using xml*, 10th International Conference on Management of Data (COMAD 2000) (Pune, India), December 2000.

[10] T. Burns, E. Fong, E. Jefferson, R. Knox, L. Mark, C. Reedy, L. Reich, N. Roussopoulos, and N. Truszowski, *Reference model for dbms standardization. database architecture framework task group (daftg) of the ansi/x3/sparc database system study group*, ACM Sigmod Record **15** (1986), no. 1, 19–58.

[11] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur, *PVFS: A parallel file system for linux clusters*, Proceedings of the 4th Annual Linux Showcase and Conference (Atlanta, GA), USENIX Association, October 2000, pp. 317–327.

[12] Matthew M. Cettei, Walter B. Ligon III, and Robert B. Ross, *Support for parallel out-of-core applications on beowulf workstations*, Proceedings of the 1998 IEEE Aerospace Conference, March 1998, pp. 355–365.

[13] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana, *Web services description language (wsdl) 1.1*, http://www.w3.org/TR/wsdl, March 2001.

[14] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snir, Bernard Traversat, and Parkson Wong, *Overview of the MPI-IO parallel I/O interface*, Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems, April 1995, pp. 1–15.

[15] Peter F. Corbett and Dror G. Feitelson, *The Vesta parallel file system*, ACM Transactions on Computer Systems **14** (1996), no. 3, 225–264.

[16] Peter F. Corbett, Dror G. Feitelson, Jean-Pierre Prost, George S. Almasi, Sandra Johnson Baylor, Anthony S. Bolmarcich, Yarsun Hsu, Julian Satran, Marc Snir, Robert Colao, Brian Herr, Joseph Kavaky, Thomas R. Morgan, and Anthony Zlotek, *Parallel file systems for the IBM SP computers*, IBM Systems Journal **34** (1995), no. 2, 222–248.

[17] ACCS Corporation, *Lmbench results — tcp latency, http://www.accs.com/p_and_p/gigabit/results_lmbench2.html.*

[18] IBM Corporation, *The ibm aix parallel i/o file system: Installation,administration, and use*, 1995.

[19] Intel Corporation, *Paragon system user's guide, includes a chapter on using pfs but has little information on its underlying design*, April 1996.

[20] Cortes David, Arthur Evans, Wendy Ferguson, Jed Hartman, and Susan Thomas, *Topics in irix programming, contains information on high performance i/o programming for sgi systems*, Silicon Graphics, Inc., 1998.

[21] Erik DeBenedictis and Juan Miguel del Rosario, *nCUBE parallel I/O software*, Proceedings of the Eleventh Annual IEEE International Phoenix Conference on Computers and Communications (Scottsdale, AZ), IEEE Computer Society Press, April 1992, pp. 0117–0124.

[22] E.Gould and M.Xinu, *The network file system implemented on 4.3 bsd*, Proceedings of the USENIX Association Summer Conference, 1986.

[23] Jack Dongarra et al., *Mpi - a message passing interface standard*, International Journal of Supercomputer Applications and High Performance Computing **8** (1994), no. 3.

[24] A Network Protocol Independent Performance Evaluator, *http://www.scl.ameslab.gov/netpipe*.

[25] G. Fagg, J. Dongarra, and A. Geist, *Heterogeneous mpi application interoperation and process management under pvmpi*, Tech. report, University of Tennessee Computer Science Department, June 1997.

[26] Peter F.Corbett and Dror G.Feitelson, *The vesta parallel file system*, ACM Transactions on Computer Systems, August 1996, pp. 225–264.

[27] Rene Felder and Erich Schikuta, *Towards an xml based data grid description language*, PACT-SPDSEC-02, September 2002.

[28] Ian Foster and Editors Carl Kesselman, *The grid: Blueprint for a future computing infrastructure*, Morgan Kaufmann, 1999.

[29] Ian Foster and Carl Kesselman, *The globus project: a status report*, Proceeding of the Seventh Heterogeneous Computing Workshop, March 1998, pp. 4–18.

[30] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke, *The physiology of the grid*, draft, June 2002.

[31] Ian Foster, David Kohr, Jr., Rakesh Krishnaiyer, and Jace Mogill, *Remote I/O: Fast access to distant storage*, Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems (San Jose, CA), ACM Press, November 1997, pp. 14–25.

[32] Open Software Foundation, *Introduction to osf dce, brief description of all of dce, including a chapter on dfs*, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[33] Thomas Fuerle, Oliver Jorns, Erich Schikuta, and Helmut Wanek, *Meta-vipios: Harness distributed i/o ressources with vipios*, Iberoamerican Journal of Research "Computing and Systems", Special Issue on Parallel Computing 4 (2000), no. 2, 124–142.

[34] Thomas Fuerle and Erich Schikuta, *A transparent communication layer for heterogenous, distributed systems*, 2003.

[35] Thomas Fuerle, Erich Schikuta, Christoph Lffelhardt, Kurt Stockinger, and Helmut Wanek, *On the implementation of a portable, client-server based mpi-io interface*, EuroPVM/MPI98 (Liverpool, UK) (Springer-Verlag Lecture Notes in Computer Science, ed.), September 1998.

[36] Feitelson Dror G. and Klainer Tomer, *High performance mass storage and parallel i/o: Technologies and applications*, ch. XML, Hyper-media, and Fortran I/O, John Wiley and Sons, November 2001.

[37] A. Geist et al., *Pvm 3.0 user's guide and reference manual*, Oak Ridge National Labratory, 1994.

[38] David S. Greenberg, Ron Brightwell, Lee Ann Fisk, Arthur B. Maccabe, and Rolf Riesen, *A system software architecture for high-end computing*, Proceedings of Supercomputing, November 1997.

[39] Andrew S. Grimshaw, Michael J. Lewis, Adam J. Ferrari, and John F. Karpovich, *Architectural support for extensibility and autonomy in wide-area distributed object systems, overview of the legion run-time architecture*, Technial

Report CS-98-12, Department of Computer Science, University of Virginia, June 1998.

[40] R.L. Haksin, *Tiger shark - a scalable file system for multimedia*, IBM Journal of Research and Development, March 1998, pp. 185–197.

[41] *The ncsa hdf home page*, http://hdf.ncsa.uiuc.edu/.

[42] Dolphinics Interconnect Solutions Inc., *http://www.dolphinics.com.*

[43] Sun Microsystems Inc., *The nfs distributed file service*, Online whitepaper available at www.sun.com/software/white-papers/wp-nfs.sw, describes Version 3 of NFS, 1995.

[44] D.E. Jardine, *The ansi/sparc dbms model*, North-Holland, The Netherlands, 1977.

[45] Barkes Jason, Marecelo R.Barrios, Fancis Cougard, Paul G.Crumley, Didac Marin, Hari Reddy, and Theeraphong Thitayanun, *Gpfs: A parallel file system, documentation on gpfs for users and system administrator, http://www.redbooks.ibm.com*, IBM International Technial Support Organization, April 1998.

[46] J.H.Howard, *An overview of the andrew file system*, Proceedings of the USENIX Association Winter Conference, February 1988, pp. 213–216.

[47] Terry Jones, Richard Mark, Jeanne Martin, John May, Elsie Pierce, and Linda Stanberry, *An MPI-IO interface to HPSS*, Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems, September 1996, pp. I:37–50.

[48] Michale Kazar, Bruce W.Leverett, Owen T.Anderson, Vasilis Apostolides, Beth A.Bottos, Sailesh Chutani, Craig F.Everhart, W.Anthony Mason, Shu-Tsui Tu, and Edward R. Zayas, *Decorum file system architectural overview, introduces*

*the file system that became dce's dfs*, Proceedings of the USENIX Association Summer Conference, June 1990, pp. 151–163.

[49] W. B. Ligon and R. B. Ross, *Implementation and performance of a parallel file system for high performance distributed applications*, Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, August 1996, pp. 471–480.

[50] Johnson Lori, *Cxfs software installation and administration guide, available at techpubs.sgi.com*, Silicon Graphics, Inc., 1999.

[51] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler, sfs*: A parallel file system for the CM-5*, Proceedings of the 1993 Summer USENIX Technical Conference, 1993, pp. 291–305.

[52] John M. May, *Parallel i/o for high performance computing*, Morgan Kaufmann Publishers, October 2000.

[53] P. Melas and E.J. Zaluska, *Performance of message-passing systems using a zero-copy communication protocol*, International Conference on Parallel Architectures and Compilation Techniques, October 1998, p. 264 ff.

[54] Message-Passing Interface Forum, *MPI-2.0: Extensions to the message-passing interface*, ch. 9, MPI Forum, June 1997.

[55] D. Moody, *The intel ipsc/2 concurrent file system. high-level description of cfs, the predecessor of intel's pfs*, Software for Parallel Computers, 1992, pp. 229–241.

[56] *MPI-IO: a parallel file I/O interface for MPI*, The MPI-IO Committee, April 1996, Version 0.5.

[57] Inc. Myrinet, *http://www.myrinet.com.*

[58] Nils Nieuwejaar and David Kotz, *The Galley parallel file system*, Proceedings of the 10th ACM International Conference on Supercomputing (Philadelphia), ACM Press, May 1996, pp. 374–381.

[59] Paul Pierce, *A concurrent file system for a highly parallel mass storage system*, Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications (Monterey, CA), Golden Gate Enterprises, Los Altos, CA, March 1989, pp. 155–160.

[60] Erich Schikuta and Thomas Fuerle, *Vipios islands: Utilizing i/o resources on distributed clusters*, 15th International Conference on Parallel and Distributed Computing Systems (Louisville), September 2002.

[61] Erich Schikuta, Thomas Fuerle, and Helmut Wanek, *ViPIOS: The Vienna Parallel Input/Output System*, Proc. of the Euro-Par'98 (Southampton, England), Lecture Notes in Computer Science, Springer-Verlag, September 1998.

[62] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett, *Server-directed collective I/O in Panda*, Proceedings of Supercomputing '95 (San Diego, CA), IEEE Computer Society Press, December 1995.

[63] Kent E. Seamons, *Panda: Fast access to persistent arrays using high level interfaces and server directed input/output*, Ph.D. thesis, University of Illinois at Urbana-Champaign, May 1996.

[64] Ben Segal, *Grid computing: The european data project*, IEEE Nuclear Science Symposium and Medical Imaging Conference (Lyon), October 2000.

[65] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, and Udaya A. Ranawake, *Beowulf: A parallel workstation for scientific computation*, Proceedings, International Conference on Parallel Computing, 1995, vol. 1, August 1995, pp. 11–14.

[66] Heinz Stockinger, *Dictionary on parallel input/output*, Master's thesis, Department of Data Engineering, University of Vienna, February 1998.

[67] Kurt Stockinger, *ViMPIOS - a portable, client-server based implementation of MPI-IO on ViPIOS*, December 1998, Master's Thesis, Dept. of Data Engineering, University of Vienna.

[68] Ellis Susan and Steven Levine, *Irix admin: Disks and filesystems, sgi documentation for xfs and xlv, aimed at system administrator, available at techpubs.sgi.com*, Silicon Graphics, Inc., 1998.

[69] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi, *Passion: Optimized I/O for parallel applications*, IEEE Computer **29** (1996), no. 6, 70–78.

[70] Rajeev Thakur, William Gropp, and Ewing Lusk, *An abstract-device interface for implementing portable parallel-I/O interfaces*, Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation, October 1996, pp. 180–187.

[71] Rajeev Thakur, Ewing Lusk, and William Gropp, *Users guide for ROMIO: A high-performance, portable MPI-IO implementation*, Tech. Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

[72] Richard W. Watson and Robert A. Coyne, *The parallel I/O architecture of the high-performance storage system (HPSS)*, Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems, IEEE Computer Society Press, September 1995, pp. 27–44.

# LEBENSLAUF

Mag.Thomas Fürle
Löhrgasse 3/5
A-1150 Wien

Email: Thomas.Fuerle@gmx.at

| | | | |
|---|---|---|---|
| Persönliche Daten: | Familienstand: | ledig | |
| | Staatsangehörigkeit: | Österreich | |
| | Geburtsdatum: | 23.10.1969 | |
| | Geburtsort: | Mödling / NÖ | |
| Ausbildung: | Volksschule Südstadt | | 1976 - 1980 |
| | Gymnasium Mödling | | 1980 - 1982 |
| | Realgymnasium Mödling | | 1982 - 1984 |
| | HTL Mödling (Nachrichtentechnik) | | 1984 – 1989 |
| | Anmerkung: Abschluss HTL mit Vorzug | | |
| | 8 Monate Bundesheer (Maria Theresien Kaserne, Wien 13) | | 1989 - 1990 |
| | Studium der Wirtschaftsinformatik (Technische Universität und Universität Wien) | | 1990 - 1997 |
| | Sponsion zum Magister der Wirtschafts-informatik | | Februar 1997 |
| | Visiting Research Student at the Computer Department (ICL) of the University of Knoxville/Tennessee (USA) 1998 | | Aug. – Dez. |

## Berufserfahrung (Freelancer mit Gewerbeschein)

**Vermessungsbüro Dr. Palfinger (Mödling) 1991 - 1999**
**Vermessungsbüro GISTECH (Mödling) 2002**

- C, C++, Fortran, Java, X11, Motif u. Qt – Programmierung
- Erstellung eines AutoCad ähnlichen GUIs in Motif
- UNIX Administrator (HP-UX, Linux)
- UNIX Datenbankadministrator (Oracle für Linux, SQL, PL-SQL)
- UNIX-NT PowerUser (Shellscripts, Perl, KDE, VB Makros)
- UNIX Server Solutions with Linux (DHCP, NFS, NIS, Internet connectivity mit IP-Masquerading, ISDN-Access, Firewall), High Performance Solutions mit Linux (Beowulf Clusters), z.B. Channel Bonding (mehrere Netzwerkkarten pro Server)
- UNIX-NT Connectivity (Samba, telnet, ftp, NFS)
- Win95, -98, -NT, WinCenter (WinFrame), WTS Administrator
- IT-, Netzwerk- u. Hardware Planung, Einkauf

**BFI Wien 1991 - 1999**

- EDV-Trainer in der Erwachsenenbildung (EDV/PC-Einführungs-, DOS-, Windows-, Accesskurse, Internet, Linux, Unix, Perl, KDE)

**E + O Incentives + Conventions** (Wien) **1998 -**

- UNIX Server Solutions mit Linux (Internet Anschluss, File-, Fax- u. Print Server) mit Win98 als Clients

## Berufserfahrung (Angestelltenverhältnisse)

**Herold Business Data AG 1999**

- Aufsetzen, Wartung und technische Projektverantwortlichkeit der GelbenSeiten (http://www.gelbeseiten.at) unter Linux (Apache, perl, MySQL, Webalizer für Statistiken), Teamleader des Entwicklungsteam für Web u. CD

**Oracle** **2000 - 2002**

- B2B Integration mittels dem Oracle Integration Server
- DBA für Oracle Applications (seit Feb. 2001) auf Solaris, HP-UX;
- Tools: Oracle 8i (bzw. 9i), Apache, SQL, PL/SQL, Jserv
- Administrator für interne Solaris Machinen
- Netzwerkadministrator für interne Consulting Projekte (Anbindung von VPN wie Checkpoint, Wireless LAN, ...)
- Fortbildung (Oracle):

K1000 - Professioneller Einstieg in Oracle SQL
K1070 - PL/SQL und Datenbankprogrammierung
K1075 - PL/SQL Aufbau
K1080 - New Features für Developer 8i to 9i
K3308 - Oracle 8i Datenbankadminstration
K3811 - New Features 9i für Adminstratoren
K3851 - Oracle 8i Peformance Tuning
K3875 - Oracle 9i Real Application Cluster
K5700 - JAVA Programmierung mit Oracle Jdeveloper
K5720 - Entwicklung von DB - Anwendungen mit JAVA
K5760 - Entwicklung von Servlets und Java Server Pages

Sonstiges

**Fremdsprachen:**

Englisch ...................... in Wort und Schrift
Technisches Englisch ... in Wort und Schrift

**Führerscheine:**

A, B, C, F, G

**Hobbies:**

Schifahren, Snowboarding, Mountain Bike, Laufen,
Inline Skating, geprüfter Ski- u. Snowboardlehrer, Schach,
Adventure Games, Kartenspiele

**Allgemein:**

Beobachtung der Entwicklung u. professioneller
Einsatzmöglichkeiten von Linux, im speziellem SuSE Linux

Universtätslaufbahn

## Diplomarbeit an der Universität Wien

*PANNS (Parallel Artifical Neural Network Simulator)*

PANNS ist ein Werkzeug zum Erzeugen, Exekutieren und Analysieren von Neuralen Netzwerkmodellen. Dieser Simulator erlaubt jede beliebe Form von NN´s auf einem COW (Cluster von Workstations).
Das System basiert auf NeurDS von der Firma Digital© und wurde durch den Einsatz von PVM (Parallel Virtual Machine) zu PANNS erweitert.

PANNS läuft auf den meisten UNIX-Systemen, getestet wurde es unter SuSE Linux, Solaris u. HP-UX.

## Master Thesis at the Universitäty of Vienna

*PANNS (Parallel Artifical Neural Network Simulator)*

PANNS is a tool for creation, execution and analysing of neural network models. This simulator allows any desired form of NN's on a COW (Cluster von Workstations). The system is based on NeurDS, manyfactured by Digital© and was extended by the use of PVM (Parallel Virtual Machine) to PANNS.

PANNS runs on all UNIX platforms, which support PVM, it was tested on SUSE Linux, Solaris and HP-UX.