Dealing with Massive Data: from Parallel I/O to Grid I/O

Master's Thesis

Rainer Hubovsky **Florian Kunz**

`rainer@hubovsky.net` `flo@post.com`

Advisor: ao.Univ.-Prof. Dr. Erich Schikuta

Institute for Applied Computer Science and Information Systems

Department of Data Engineering, University of Vienna

Rathausstr. 19/4, A-1010 Vienna, Austria

January 16, 2004

Acknowledgements

Many people have helped us find our way during the development of this thesis. Erich Schikuta, our supervisor, provided a motivating, enthusiastic, and critical atmosphere during our discussions. It was a great pleasure for us to conduct this thesis under his supervision. We also acknowledge Heinz and Kurt Stockinger who provided constructive comments. We would also like to thank everybody for providing us with feedback.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Increasing requirements in *High-Performance Computing (HPC)* led to improvements of CPU power, but bandwidth of I/O subsystems does not keep up with the performance of processors any more. This problem is commonly known as the *I/O bottleneck.*

Additionally, new and stimulating data-intensive problems in biology, physics, astronomy, space exploration, human genom research arise, which bring new high-performance applications dealing with *massive data* spread over globally distributed storage resources.

Therefore research in HPC focuses more on I/O systems: all leading hardware vendors of multiprocessor systems provided powerful concurrent I/O subsystems. In accordance researchers focus on the design of appropriate programming tools and models to take advantage of the available hardware resources.

Numerous projects about this topic have appeared, from which a large and unmanageable quantity of publications have come. These publications concern themselves to a large extent with very special problems. Due to the time of their appearance the few overview papers deal with *Parallel I/O or Cluster I/O* [184, 306, 333].

Substantial progress has been made in these research areas since then. *Grid Computing* has emerged as an important new field, distinguished from conventional *Distributed Computing* by its focus on large-scale resource sharing, innovative applications and, in some cases, high-performance orientation. Over the past five years, research and development efforts within the Grid community have produced protocols, services and tools that address precisely the challenges that arise when we try to build Grids, I/O being an important part of it.

Similar to Heinz Stockinger's work *Dictionary on Input/Output* [333], which is based on the *Parallel I/O Archive* by David Kotz [266], we give an overview of I/O in HPC. Query optimization is not part of our work. Apart from few exceptions, only sources were con-

sidered which are not older than 2000.
Classification has been made and is presented in [type]. More details about the classification are given in chapter *Classification*. To be sure that the information is not wrong or outdated, we gave the project leaders, named in [contact], the opportunity to make corrections or additions to the entries before we included them in the final version. The corresponding webpage follows in [url]. The [description] section gives a short overview, [motivation] and [features] are stated. [application] names projects in which the software or idea gets used. After presenting [related work], citation names the sources of information for the respective entry. Most of the text and figures are taken out of these papers. [details] describe the entry. To accommodate the fact that the Grid is an all dominating topic in the Distributed Computing community nowadays, we dedicate chapter *Grid Projects* to this topic.

Layout and Text Style

Text in Helvetica 16 presents keywords followed by a body of explanation. Text written in *italics* is used to emphasize. Within the text, two kinds of references can be found:

▷*keyword* refers to a project which is explained in chapter *Projects*

▶*keyword* refers to an entry in chapter *Dictionary*

Chapter 2

Classification

In order to get an overview of the very heterogeneous range of topics, we structured the material. The entries were assigned to the following groups:

Access Anticipation Method: tries to foresee the I/O access characteristics of the application based on programmer's hints, anticipated knowledge or reaction to identified behaviour.

Application Level Method: organizes the mappings of the application's main memory objects to respective disk objects to make disk accesses more efficient by exploiting data locality.

Data Access System: provides mechanisms for tools and applications that require high-performance access to data located mostly in remote, potentially parallel file systems or Grids.

Data Format: provides a schema for efficiently storing, transferring and accessing data by defining physical data structures for images, multidimensional arrays, tables etc.

Data Transfer Protocol: mechanisms for reliably transferring data in HPC and Grid environments.

Definition: presents and identifies terms.

Device Level Method: reorganizes the disk access operations according to the application's request to optimize the performance of the I/O system.

File System: for organizing directories and files stored on a given drive (single partition, shared disk, etc.), generally in terms of how it is implemented in the disk operating system.

Filter System: provides the possibility to place processes near data to filter and process data close to the source thereby reducing the amount of data transferred over the network. With some systems it is even possible to push down processing as far as into the hard disks.

General Method: basic I/O method.

Intelligent I/O System: software architecture which hides the physical disk accesses from the application developer by providing a transparent logical I/O environment.

Library: for the application developer and consists basically of a set of highly specialized I/O functions.

Mass Storage System: high-capacity, external, large-scale data archive, such as disk or tape, more intelligent than storage system, global view.

Other: miscellaneous.

Replication: used for managing copies of file instances, or replicas, within specified storage systems.

Standardization: effort to define standard interfaces and behaviours of components.

Storage System: computer system that provides storage for one or more hosts.

Toolkit: integrated set of software routines or utilities used to develop and maintain applications or whole systems and to provide core functionalities to users of these systems.

Chapter 3

Grid Projects

CCTK (Cactus Computational Toolkit)

contact:

`cactusmaint@cactuscode.org`

url:

`http://www.cactuscode.org`

description:

CCTK is an open source problem solving environment designed for scientists and engineers. Its modular structure easily enables highly modular, multi-language, parallel applications to be developed by single researchers and large collaborations alike. *CCTK* is not an application in itself, but a development environment in which an application can be developed and run.

motivation:

The *CCTK* was originally developed to provide a framework for the numerical solution of Einstein's Equations [316]. The large and varied computational requirements of solving these equations for scenarios such as black hole or neutron star collisions, make them a good example for demonstrating the need for Grid Computing, and an ideal testbed for developing new techniques. In developing the *CCTK* infrastructure to make full use of the Grid for such problems these advances are then immediately available for all applications.

features:

- highly portable
- powerful application programming interface

- advanced computational toolkit
- collaborative development
- exhaustive numerical relativity and astrophysical applications

application:

Science applications in fields such as numerical relativity, climate modelling, chemical engineering, fusion modelling and astrophysics.

related work:

▷*Condor*, ▷*Globus*, ▷*Legion*

citation:

[10], [11], [12], [14], [15], [155]

details:

The code is structured as a core, the *flesh* and modules which are referred to as *thorns*.

The *flesh* is independent of all thorns and provides the main program, which parses the parameters and activates the appropriate thorns, passing control to thorns as required. It contains utility routines which may be used by thorns to determine information about variables, which thorns are compiled in or active, or perform non-thorn-specific tasks.

A thorn is the basic working module within *CCTK*. They are organized into logical units referred to as *arrangements*. Once the code is built these have no further meaning - they are used to group thorns into collections on disk according to function or developer or source. All user-supplied code goes into thorns, which are, by and large, independent of each other. Thorns communicate with each other via calls to the flesh API, plus, more rarely, custom APIs of other thorns.

The connection from a thorn to the flesh or to other thorns is specified in configuration files which are parsed at compile time and used to generate glue code which encapsulates the external appearance of a thorn.

When the code is built a separate build tree, referred to as a *configuration* is created for each distinct combination of architecture and configuration options. Associated with each configuration is a list of thorns which are actually to be compiled into the resulting executable, this is referred to as a *thornlist*.

At run time the executable reads a parameter file which details which thorns are to be active, rather than merely compiled in, and specifies values for the control parameters

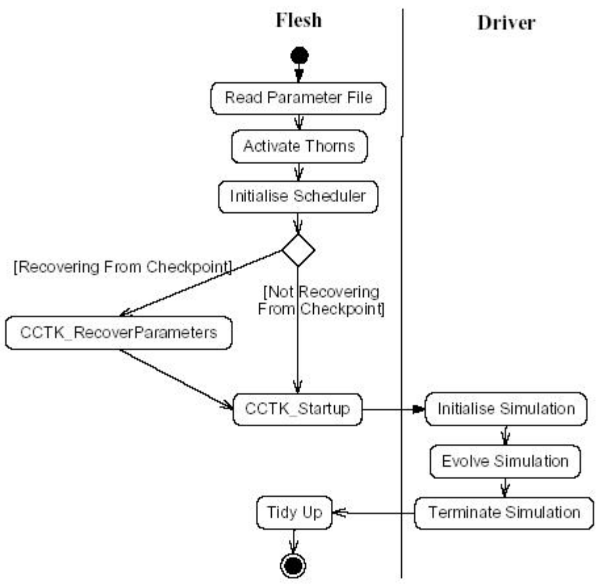


Figure 3.1: CCTK: Main Program Flow

for these thorns. Non-active thorns have no effect on the code execution. The main program flow is shown in Figure 3.1.

The flesh has minimal knowledge of the I/O, but provides a mechanism so that thorn authors can call for output of their variables and the appropriate I/O routines will be called. All thorns providing I/O routines register themselves with the flesh, saying they provide an I/O method, which is a unique string identifier. Associated with an I/O method are three functions: one to output all variables which need output, one to output a specific variable, and one to determine if a variable requires output. The first two of these routines then have analogues in the flesh which traverses all I/O methods calling the appropriate routines, or call the routine corresponding to a specific I/O method. Once per iteration the master evolution loop calls the routine to output all variables by all I/O methods. Thorns providing I/O methods typically have string parameters which list the variables which should be output, how frequently and where the output should go. The *IOUtil* thorn provides various default parameters such as the output directory, the number of iterations between output of variables, various down-sampling parameters and other parameters which may need to be used by thorns providing output of the same data but in different formats. It also provides common utility routines for I/O.

CoG (Globus COmmodity Grid Toolkit)

contact:

Gregor von Laszweski, `gregor@mcs.anl.gov`

url:

`http://www.globus.org/cog/`

description:

CoG toolkits provide the middleware for accessing the functionality of the Grid from a variety of commodity technologies, frameworks, and languages. Technologies and frameworks of interest currently include CORBA [267], Java [385, 389], Perl [360], Python [187], and Web Services [392].

motivation:

- enabling developers of *Problem Solving Environments (PSEs)* [289] to exploit commodity technologies wherever possible
- exporting Grid technologies to commodity computing for easy integration in PSEs

features:

CoGs should be:

- problem-oriented
- integrated
- distributed
- persistent
- open, flexible, adaptive
- graphical, visual

application:

Active Thermochemical Table Framework [391]

related work:

►*GASS*, ▷*Globus*, ►*GridFTP*, GSI [68]

citation:

[187], [267], [360], [377], [385], [386], [387], [388], [389], [390]

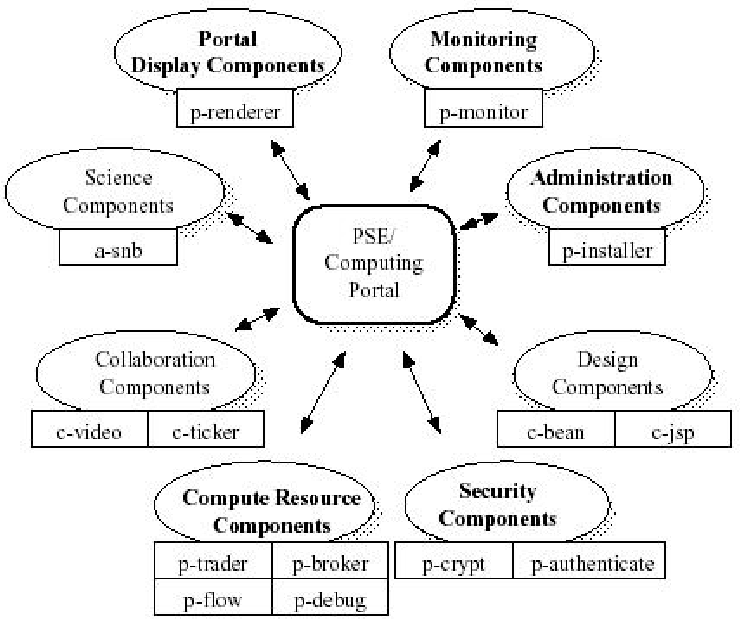


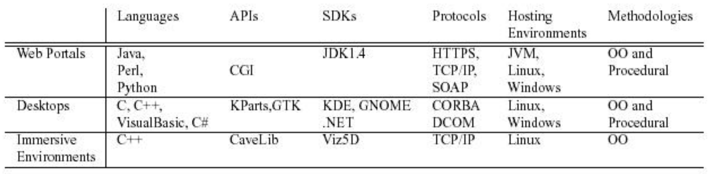
Figure 3.2: CoG: A Computing Portal

details:

A portal toolkit includes a set of services exposed via APIs that can be used to assemble a point solution for a problem. Figure 3.2 outlines the various groups of services that must be integrated into a portal toolkit. Each portal component may have several subcomponents that support the tasks performed as part of the computing portal for problem solving environments. The components in bold text of Figure 3.2 are developed as part of the *CoG* toolkit. Other components are provided either by commodity software or the application programmers. The flexible design makes it possible to integrate new components into the framework or replace existing modules.

Because of the use of different commodity technologies as part of different application requirements, a variety of *CoG* toolkits must be supported. In Table 3.1 a subset of commodity technologies are shown that are useful to developing *Grid Computing Environments (GCEs)*.

Since the implementations of the *CoG* toolkits are based on ▷*Globus*, components like ▶*GASS*, ▶*GridFTP*, and GSI [68] are used.



	Languages	APIs	SDKs	Protocols	Hosting Environments	Methodologies
Web Portals	Java, Perl, Python	CGI	JDK1.4	HTTPS, TCP/IP, SOAP	JVM, Linux, Windows	OO and Procedural
Desktops	C, C++, VisualBasic, C#	KParts,GTK	KDE, GNOME .NET	CORBA DCOM	Linux, Windows	OO and Procedural
Immersive Environments	C++	CaveLib	Viz5D	TCP/IP	Linux	OO

Table 3.1: CoG: Technologies Used to Develop Grid Computing Environments

Condor

contact:

`condor-admin@cs.wisc.edu`

url:

`http://www.cs.wisc.edu/condor/`

description:

Condor is a specialized workload management system for compute-intensive jobs. Like other full-featured batch systems, *Condor* provides a job queuing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management.

motivation:

The goal of the *Condor* Project is to develop, implement, deploy, and evaluate mechanisms and policies that supports HTC on large collections of distributively owned computing resources.

features:

- checkpoint and migration
- remote system calls
- no changes necessary to user's source code
- pools of machines can be hooked together
- jobs can be ordered
- *Condor* enables Grid Computing
- sensitive to the desires of machine owners
- ClassAds [94]

related work:

▷*Globus*

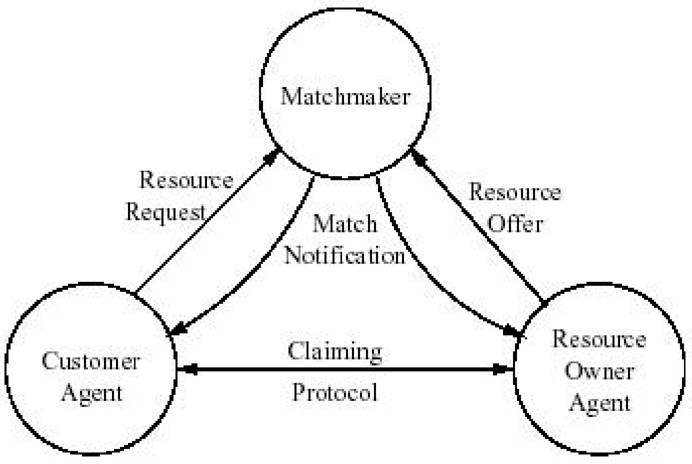


Figure 3.3: Condor: Matchmaker Architecture

citation:

[218], [285], [299], [348]

details:

In *Condor*, each user is represented by a customer agent, which manages a queue of application descriptions and sends resource requests to the matchmaker. Each resource is represented by a resource agent, which implements the policies of the resource owner and sends resource offers to the matchmaker. Using the ClassAds [94] mechanism the matchmaker is responsible for finding matches between resource requests and resource offers and notifying the agents when a match is found. Upon notification, the costumer agent and the resource agent perform a claiming protocol to initiate the allocation. This architecture is illustrated in Figure 3.3. Resource requests and offers contain constraints which specify if a match is acceptable. The matchmaker implements systemwide policies by imposing its own set of constraints on matches.

The resource management architecture of *Condor* benefits from a layered system design. This approach yields a modular system design, where the interface to each system layer is defined in the resource management model, allowing the implementation of each layer to change so long as the interface continues to be supported. This architecture separates the advertising, matchmaking, and claiming protocols. The agents advertise resource offers and requests asynchronously to the matchmaker, and the matchmaker notifies the agents when a match is found.

CrossGrid

contact:

Marian Noga, `marian.noga@cyfronet.krakow.pl`

url:

`http://www.crossgrid.org/`

description:

CrossGrid will develop techniques for large-scale Grid-enabled simulations and visualizations that require responses in real-time.

motivation:

The main objective of the *CrossGrid* project is to extend the Grid environment across Europe, and to a new category of applications.

features:

- distribution of source data
- simulation and visualization
- virtual time management
- interactive simulation and visualization rollback
- platform-independent virtual reality

application:

The four main application areas addressed by *CrossGrid* are:

- pretreatment planning in vascular intervention and surgery
- Grid-based support for flood defence and prevention
- analysis of simulations in physics
- weather forecasting and air pollution modelling

related work:

▷*EDG*, ▷*GGF*, ▷*GridStart*, ▷*GriPhyN*, ▷*PPDG*

citation:

[100]

details:

The *CrossGrid* project is divided into four work packages which deal with the technical aspects of the project, and one work package dealing with management, dissemination, and exploitation:

WP1 – *CrossGrid* Application Development

WP2 – Grid Application Programming Environments development, integration and testing of tools that facilitate the development and tuning of parallel, distributed, and interactive applications on the Grid.

WP3 – New Grid Services and Tools develops the new, generic *CrossGrid* services and software infrastructure to support the Grid users, applications and tools as defined in the work packages WP1 and WP2.

WP4 – International Testbed Organization collects all of the developments from the work packages WP1-3 and integrates them into successive software releases. It also gathers and transmits all feedback from the end-to-end application experiments back to the developers, thereby linking development, testing, and user experience.

WP5 Project Management ensures the professional management of the project and active dissemination and exploitation of its results.

DODS (Distributed Oceanographic Data System)

contact:

`support@unidata.ucar.edu`

url:

`http://www.unidata.ucar.edu/packages/dods/index.html`

description:

DODS is a framework that simplifies all aspects of scientific data networking.

motivation:

The goal of *DODS* is to build a highly distributed system that allows users to control the distribution of their own data and the way they access data from remote sites.

related work:

▹*ESG-II*

citation:

[113], [115]

details:

DODS is a software framework that simplifies all aspects of scientific data networking, allowing simple access to remote data. Local data can be made accessible to

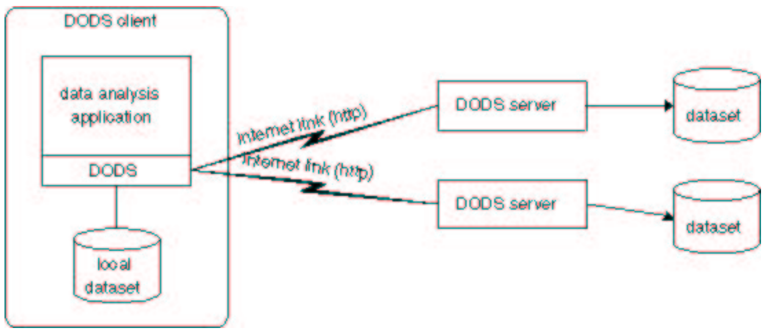


Figure 3.4: DODS: Architecture

remote locations regardless of local storage format by using *DODS* servers as shown in Figure 3.4. Existing, familiar data analysis and visualization applications can be transformed into *DODS* clients.

DODS is a protocol for requesting and transporting data across the web. The current *DODS Data Access Protocol (DAP)* uses HTTP to frame the requests and responses.

EDG (European DataGrid)

contact:

Alexia Augier-Bochon, `Alexia.Augier-Bochon@cern.ch`

url:

`http://www.edg.org`

description:

The *EDG* project will devise and develop scalable software solutions and testbeds to handle many Petabytes of distributed data, tens of thousand of computing resources including processors, disks, other devices and thousands of simultaneous users from collaborating research institutes.

motivation:

The objective of *EDG* is to enable next generation scientific exploration which requires intensive computation and the analysis of shared, large-scale databases.

features:

- job scheduling
- data management
- grid monitoring

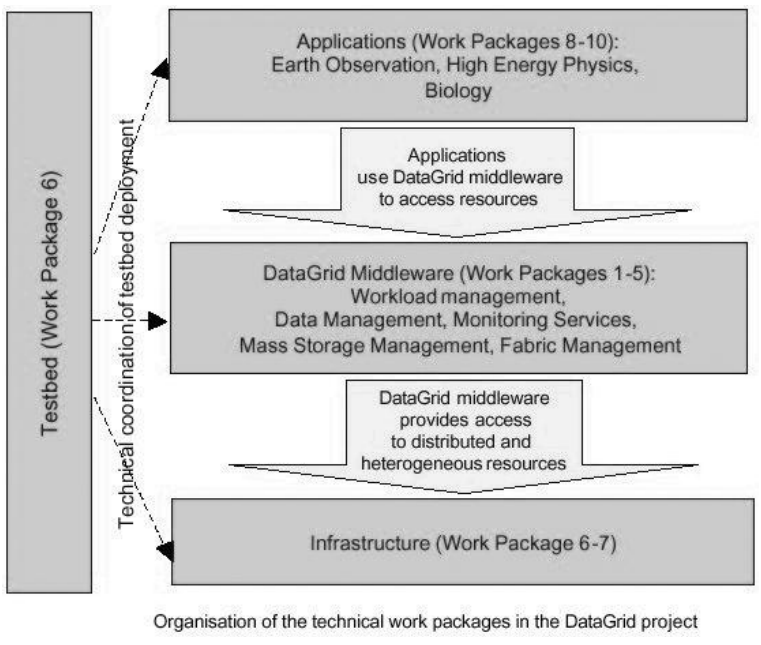

Testbed (Work Package 6)
Technical coordination of testbed deployment
Applications (Work Packages 8-10): Earth Observation, High Energy Physics, Biology
Applications use DataGrid middleware to access resources
DataGrid Middleware (Work Packages 1-5): Workload management, Data Management, Monitoring Services, Mass Storage Management, Fabric Management
DataGrid middleware provides access to distributed and heterogeneous resources
Infrastructure (Work Package 6-7)
Organisation of the technical work packages in the DataGrid project

Figure 3.5: EDG: Structure and Work Packages

- fabric management
- mass storage management

application:

- HEP
- biology and medical image processing
- earth observations led by the European Space Agency

related work:

▷*CrossGrid*, ▷*GGF*, ▷*GridStart*, ▷*GriPhyN*, ▷*PPDG*

citation:

[118], [315]

details:

The *EDG* project is divided into twelve work packages distributed over four working groups: Computational & EDG Middleware, Testbed and Infrastructure, Applications, Management and Dissemination. Figure 3.5 illustrates the structure of the project and the interactions between the work packages.

Work Packages:

Middleware

1. Grid Work Scheduling
2. Grid Data Management: ►*EDG Replica Manager*, ►*GDMP*, ►*Spitfire*
3. Grid Monitoring Services
4. Fabric Management
5. Mass Storage Management

Infrastructure

6. Testbed and Demonstrators
7. Network Services

Applications

8. HEP Applications
9. Earth Observation Applications
10. Biology Applications

Management

11. Dissemination
12. Project Management

Entropia

contact:

Andrew Chien, `achien@ucsd.edu`

url:

`http://www.entropia.com/`

description:

Entropia is a powerful and cost-effective PC Grid Computing platform that provides HPC capabilities by aggregating the unused processing cycles of networks of existing Windows-based PCs.

motivation:

Entropia enables a PC desktop environment for Grid Computing.

features:

- physical node management

- resource scheduling
- job management

related work:

▷*Globus*

citation:

[89], [331]

details:

The *Entropia* system aggregates raw desktop resources into a single logical resource. *Entropia's* approach to application integration, a process known as *sandboxing*, is to automatically wrap an application in the virtual machine technology. The *Entropia* system architecture is composed of three separate layers (Figure 3.6). At the bottom is the *Physical Node Management* layer that provides basic communication and naming, security, resource management, and application control. On top of this layer is the *Resource Scheduling* layer that provides resource matching, scheduling and fault tolerance. The Physical Node Management layer and Resource Scheduling layer span the servers and client machines. Users can interact directly with the Resource Scheduling layer through the available APIs or alternatively, users can access the system through the *Job Management* layer that provides management facilities for handling large numbers of computations and files. The Job Management layer runs only on the servers. Other job management systems can be used with the system.

ESG-II (Earth System Grid)

contact:

Arie Shoshani, `shoshani@lbl.gov`

url:

`http://sdm.lbl.gov/indexproj.php?ProjectID=ESG`

description:

ESG-II is a virtual collaborative environment that links distributed centres, users, models, and data.

motivation:

ESG-II will provide scientists with virtual proximity to the distributed data and resources that they require to perform their research.

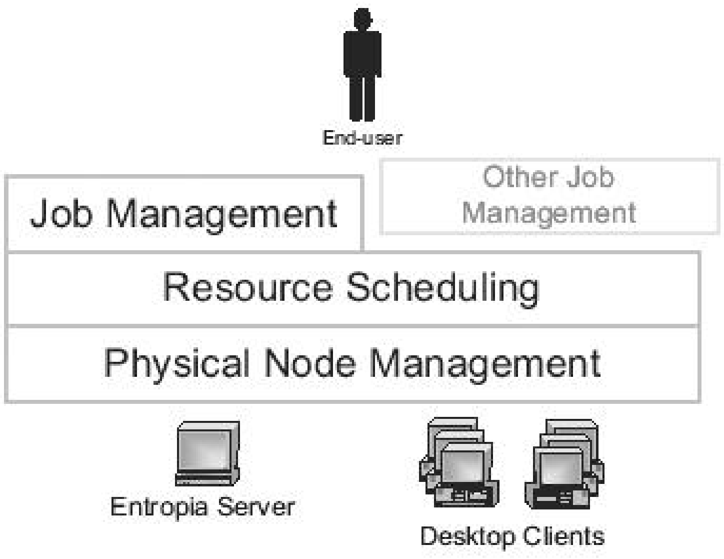


Figure 3.6: Entropia: Architecture

features:

- data publication
- data replication
- request specification
- multi-site request execution and monitoring
- data extraction and data assembly [124]

related work:

▶*DataCutter*, ▷*DODS*

citation:

[8], [123], [124], [171]

details:

ESG-II will integrate and extend a range of Grid and collaboratory technologies, including the ▷*DODS* remote access protocols for environmental data, ▷*Globus* Toolkit technologies for authentication, resource discovery, and resource access, and Grid technologies developed in other projects. The design of *ESG-II* is a layered system architecture with well-defined protocols and interfaces at each layer boundary. A layered design helps control the complexity of the overall system, simplifies development, deployment and maintenance tasks and maximizes the ability to reuse existing services and tools.

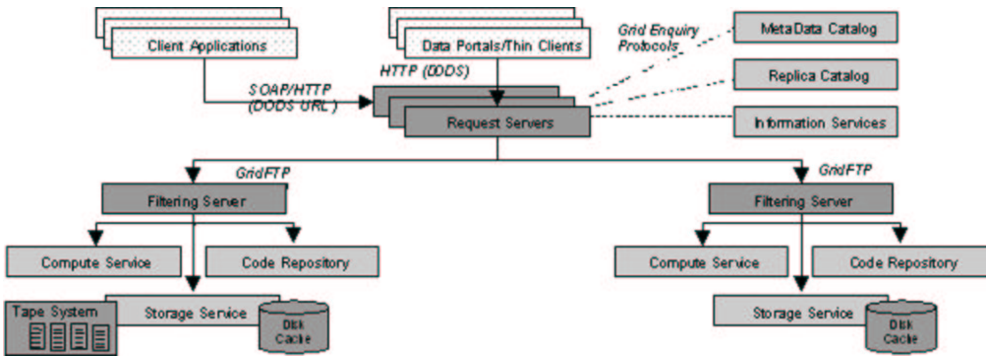


Figure 3.7: ESG-II: Component Level View

Figure 3.7 shows a component level view of the *ESG-II* architecture. A *request service* that mediates between user-level applications and the underlying Grid services and a *filtering server* that extracts specified data elements from files and performs some analysis on the data as part of responding to a data request are of special research and development interests.

EUROGRID

contact:

Daniel Mallmann, `d.mallmann@fz-juelich.de`

url:

`http://www.eurogrid.org`

description:

The *EUROGRID* project demonstrates the use of Grid technology in selected scientific and industrial communities. Core Grid software components were developed and integrated into an environment providing fast file transfer, resource brokerage, interfaces for coupled applications and interactive access.

motivation:

- support the e-Science concept
- integrate resources of leading European HPC centres into a European HPC Grid
- develop new software components for Grid Computing
- demonstrates the *Application Service Provider (ASP)* model for HPC access

application:

- Work Package 1 – Bio GRID [61]

- Work Package 2 – Meteo GRID
- Work Package 3 – CAE GRID

related work:

▷*UNICORE*

citation:

[126], [177]

details:

The systems is based on ▷*UNICORE*. The following additional Grid components are part of *EUROGRID*:

- efficient data transfer
- ASP infrastructure
- resource broker
- application coupling
- interactive access

The project ends on the 31st of January 2004.

GGF (Global Grid Forum)

type:

Standardization Effort

contact:

Charlie Catlett, `catlett@mcs.anl.gov`

url:

`http://www.gridforum.org/`

description:

The *GGF* is a community-initiated forum of 5000+ individual researchers and practitioners working on Distributed Computing, or Grid technologies.

motivation:

GGF's primary objective is to promote and support the development, deployment, and implementation of Grid technologies and applications via the creation and documentation of *best practices*-technical specifications, user experiences, and implementation guidelines.

Moreover *GGF's* goals are:

- to address architecture, infrastructure, standards and other technical requirements for Computational Grids and to facilitate and find solutions to obstacles inhibiting the creation of these Grids
- to educate the scientific community, industry, government and the public regarding the technologies involved in, and potential uses and benefits of, computational Grids
- to facilitate the application of Grid technologies within educational, research, governmental, healthcare and other industries

related work:

▷*Globus*, ▶*OGSA*, ▶*OGSA-DAI*, ▶*OGSI*

citation:

[152]

details:

GGF efforts are aimed at the development of a broadly based *Integrated Grid Architecture* that can serve to guide the research, development, and deployment activities of the emerging Grid communities. Defining such an architecture will advance the Grid agenda through the broad deployment and adoption of fundamental basic services and by sharing code among different applications with common requirements.

The work of *GGF* is performed within its various *working groups* and *research groups*. A *working group* is generally focused on a very specific technology or issue with the intention to develop one or more specific documents aimed generally at providing specifications, guidelines or recommendations. A *research group* is often longer-term focused, intending to explore an area where it may be premature to develop specifications.

The following working groups are concerned especially with I/O:

Database Access and Integration Services This group seeks to promote standards for the development of Grid database services, focusing principally on providing consistent access to existing, autonomously managed databases. ▶*OGSA-DAI*.

Data Format Description Language The aim of this working group is to define an XML-based language, the *Data Format Description Language (DFDL)*, for describing the structure of binary and character encoded (ASCII/Unicode) files and data streams so that their format, structure, and meta-data can be exposed [111].

GridFTP This group should focus on improvements of FTP and ▶*GridFTP* v1.0 protocol with the goal to produce bulk file transfer protocol suitable for Grid applications.

OGSA Replication Services This group intended to create, review and refine Grid service specifications for data replication services. ▶*OGSA*, ▶*OGSI*.

The research groups which work in the area of I/O are:

Data Transport The goal of this group is to provide a forum where parties interested in the secure, robust, high speed transport of data in the wide-area and related technologies can discuss and coordinate issues, and develop standards to ensure interoperability of implementations.

Persistent Archives The Persistent Archive Research Group of the *GGF* promotes the development of an architecture for the construction of persistent archives. ▶*Persistent Archives*.

Globus

contact:

Ian Foster, `foster@mcs.anl.gov`

url:

`http://www.globus.org/`

description:

The *Globus* Project provides software tools that make it easier to build computational Grids and Grid-based applications. These tools are collectively called the *Globus* Toolkit.

motivation:

The *Globus* system is intended to achieve a vertically integrated treatment of application, middleware, and network.

related work:

▷*Condor*

citation:

[132], [133]

details:

The open source *Globus* Toolkit version 2 (GT2) defined and implemented protocols, APIs, and services used in hundreds of Grid deployments worldwide. By providing solutions to common problems such as authentication, resource discovery, resource access, and data movement, GT2 accelerated the construction of real Grid applications. And by defining and implementing standard protocols and services, GT pioneered the creation of interoperable Grid systems and enabled significant progress on Grid programming tools. This standardization played a significant role in spurring the subsequent explosion of interest, tools, applications, and deployments.

As interest in Grids continued to grow, and in particular as industrial interest emerged, the importance of true standards increased. In particular, 2002 saw the emergence of the ▶*OGSA*, a community standard with multiple implementations–including the OGSA-based GT version 3 (GT3) [398] released in 2003. Building on and significantly extending GT2 concepts and technologies, ▶*OGSA* aligns Grid Computing with broad industry initiatives in service-oriented architectures and Web services [392].

A low-level *Globus* Toolkit provides basic mechanisms such as communication, authentication, network information, and data access. These mechanisms are used to construct various higher-level metacomputing services, such as parallel programming tools and schedulers.

The *Globus* Toolkit consists of the following components:

- security: GSI [68]
- data management: ▶*GridFTP*, Replica Catalog, Replica Management
- resource management: GRAM [159]
- information services: MDS [153]
- packaging technology

Globus I/O functions are implemented in the globus_io library [154].

GridLab

contact:

Jarek Nabrzyski, `naber@man.poznan.pl`

url:

`http://www.gridlab.org/`

description:

The *GridLab* project will develop an easy-to-use, flexible, generic, and modular *Grid Application Toolkit (GAT)*, enabling todays applications to make innovative use of global computing resources.

motivation:

- the co-development of infrastructure with real applications and user communities, leading to working scenarios
- dynamic use of Grids, with self-aware simulations adapting to their changing environment

related work:

▷*GridStart*

citation:

[13]

details:

The *GridLab* project consists of 12 core work packages, with additional work packages covering exploitation, dissemination and project management.

TB – Technical Board (TB)

WP1 – Grid Application Toolkit (GAT) The *GAT* provides a link between Grid middleware and applications.

WP2 – Cactus Grid Application Toolkit (CGAT) The *CGAT* provides an extended GAT interface for ▷*CCTK*.

WP3 – Work-Flow Application Toolkit (TGAT) The *TGAT* will develop Grid capabilities for Triana [368], a widely used data flow programming environment.

WP4 – Grid Portals

WP5 – Testbed Management

WP6 – Security

WP7 – Adaptive Grid Components

WP8 – Data Handling and Visualization

WP9 – Resource Management

WP10 – Information Services

WP11 – Monitoring

WP12 – Access for Mobile Users

WP13 – Exploitation and Dissemination

WP14 – Project Management

GridStart

contact:

Maureen Wilkinson, `m.wilkinson@epcc.ed.ac.uk`

url:

`http://www.gridstart.org/`

description:

GridStart is an initiative sponsored by the European Commission with the specific objective of consolidating technical advances in Europe, encouraging interaction amongst similar activities both in Europe and the rest of the world and stimulating the early take-up by industry and research of Grid-enabled applications. The initiative brings together technologists, scientists and industry in a multi-disciplinary approach to develop the Grid infrastructure. The clear goal is to develop sustainable, effective and universal solutions addressing the needs of science, industry and the public.

The following projects are part of *GridStart*:

AVO [39]
▷ CrossGrid
DAMIEN [105]
DataTag [106]
ESGO [125]
▷ EUROGRID
GRIA [160]

▷ Gridlab
GRIP [164]

GriPhyN (Grid Physics Network)

contact:

Paul Avery, `avery@phys.ufl.edu`

url:

`http://www.griphyn.org/`

description:

The *GriPhyN* Project is developing Grid technologies for scientific and engineering projects that must collect and analyze distributed, petabyte-scale datasets. *GriPhyN* research will enable the development of *Peta-Scale Virtual Data Grids (PVDGs)* through its ▷*VDT*.

motivation:

Driving the project are unprecedented requirements for geographically dispersed extraction of complex scientific information from very large collections of measured data.

application:

ATLAS [36], CMS [95], LIGO (Laser Interferometer Gravitational-wave Observatory) [213], SDSS (Sloan Digital Sky Survey) [311]

related work:

DataTAG [106], ▷*EDG*, ▷*Globus*, iVDGL [186], TeraGrid [342]

citation:

[132]

details:

GriPhyN is primarily focused on achieving the fundamental IT advances required to create PVDGs, but will also work synergistically on creating PVDG software systems for community use. *GriPhyN* will create a multi-faceted, domain-independent ▶*VDT*, and use this toolkit to prototype the PVDGs and to support the ATLAS (A Toroidal LHC ApparatuS) [36], CMS (Compact Muon Solenoid) [95], LIGO (Laser Interferometer Gravitational-wave Observatory) [213], and SDSS (Sloan Digital Sky Survey) [311] analysis tasks.

Legion

contact:

`legion@virginia.edu`

url:

`http://legion.virginia.edu`

description:

Legion is an object-based Grid OS charged with reconciling a collection of heterogeneous resources, dispersed across a wide-area, with a single virtual system image.

motivation:

The *Legion* OS solves the wide-area computing problem [163] by abstracting over a complex set of resources and providing high-level means for sharing and managing them.

features:

- single name space
- file system: ▶*LegionFS*
- security
- process creation and management
- interprocess communication
- input-output
- resource management
- accounting
- complexity management
- wide-area access
- heterogeneity management
- multi-language support
- legacy application support

application:

Centurion [76]

related work:

▷*Condor*, ▷*Globus*, ▶*LegionFS*

citation:

[163], [210], [246], [399]

details:

The single virtual machine view of the Grid provided by *Legion* enables users to access and use a Grid without necessarily facing the complexity of the components of the Grid. *Legion* is built around the following concepts::

Object-basedness In *Legion*, most important components of a Grid are first-class objects. Object-based design offers three advantages. First, it leads to a modular design wherein the complexity of a component is managed within a single object. Second, it enables extending functionality by designing specialized versions of basic objects. Third, it enables selecting an intuitive boundary around an object for enforcing security.

Naming & Transparency Every object in *Legion*, be it a machine, a user, a file, a subdirectory, an application, a running job or a scheduler, has a name. *Legion* unifies the multiple name spaces of traditional systems by providing a single name space for behaviourally-diverse and geographically-distributed components. Every *Legion* object has an ID associated with it - its *Legion Objects Identifier (LOID)*. The LOID of an object is a sequence of bits that identifies the object uniquely in a given Grid (and also across different Grids) without forcing subsequent accesses to violate transparency.

Service - Policy vs. Mechanism An important philosophical tenet in *Legion* is that mechanisms can be mandated but not policies. Users and administrators of a Grid must be free to configure a Grid and its components in any suitable manner by constructing policies over mechanisms.

Security Security in *Legion* is based on a PKI for authentication and *Access Control Lists (ACLs)* for authorization. *Legion* requires no central certificate authority to determine the public key of a named object, because the object's LOID contains its public key.

Extensibility Specialized objects can be constructed from basic objects for special functionality. New objects can be constructed and deployed in an existing Grid, thus extending the functionality of the Grid.

Interfaces *Legion* supports a variety of interfaces such as command-line tools, programmatic interfaces and access through familiar and traditional tools.

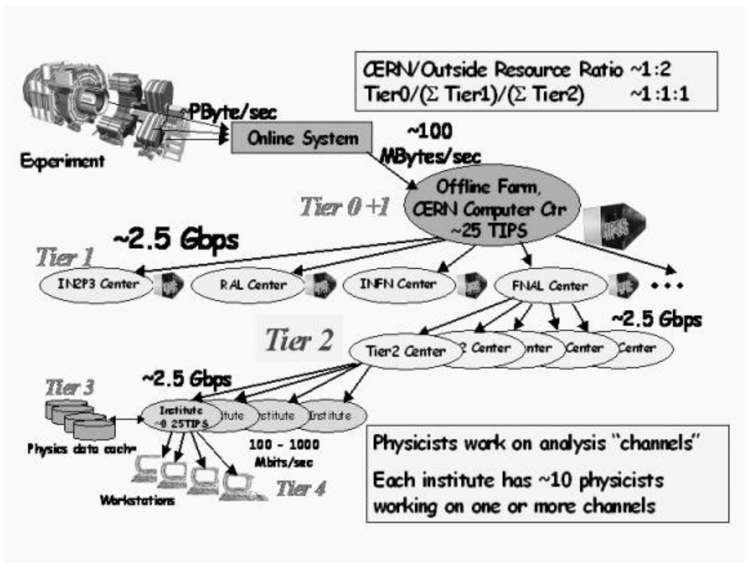


Figure 3.8: MONARC: The LHC Data Grid Hierarchy Model

Integration *Legion* is designed in order to mask complexity from the user. One of the ways in which *Legion* masks complexity is by providing an integrated system.

MONARC (Models of Networked Analysis at Regional Centres)

contact:

Michael Aderholz, `mia@mppmu.mpg.de`

url:

`http://monarc.web.cern.ch/MONARC/`

description:

The *MONARC* project has provided key information on the design and operation of the worldwide Distributed Computing models for the LHC experiments. The *MONARC* work led to the concept of a *Regional Centre* hierarchy, as shown in Figure 3.8, as the best candidate for a cost-effective and efficient means of facilitating access to the data and processing resources.

related work:

▷*EDG*, ▶*GIOD*, LCG [205]

citation:

[2], [67]

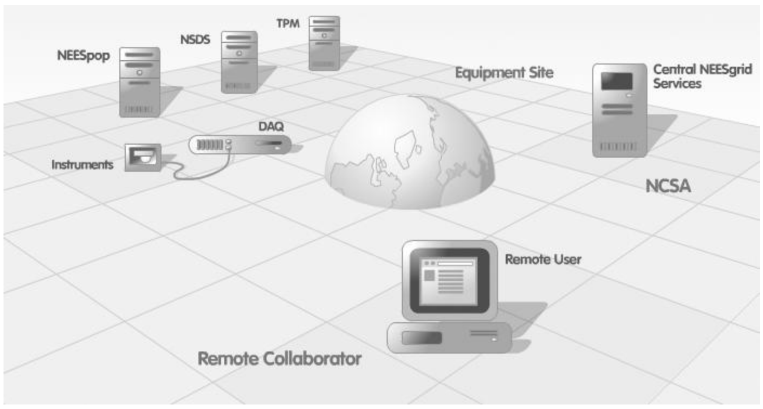


Figure 3.9: Neesgrid: System Overview

NEESgrid

contact:

Bill Spencer, `bfs@uiuc.edu`

url:

`http://www.neesgrid.org/`

description:

NEESgrid will link earthquake researchers across the USA with leading-edge computing resources and research equipment, allowing collaborative teams to plan, perform, and publish their experiments.

A system overview is depicted in Figure 3.9. The *NEESpop* server hosts the *Comprehensive Collaborative Framework (CHEF)* and other non-time-critical *NEESgrid* services. Streaming data services are managed through the *NEESgrid Streaming Data Server (NSDS)*. The *Telepresence Management (TPM)* server is responsible for video telepresence services etc. The Data Acquisition systems capture data and video from the actual experiment. *DAQ* is the LabView daemon. Remote sites communicate, primarily via a web browser interface, with the CHEF server on the NEESpop at the equipment site. A set of *Central NEESgrid Services* are hosted at NCSA.

related work:

▹*Globus*

citation:

[193]

PPDG (Particle Physics Data Grid)

contact:

`mailman-owner@lbnl2.ppdg.net`

url:

`http://www.ppdg.net/`

description:

The *PPDG* is a collaboration formed in 1999 because its members were keenly aware of the need for Data Grid services to enable the worldwide Distributed Computing model of current and future HENP experiments. It has provided an opportunity for early development of the Data Grid architecture as well as evaluating some prototype Grid middleware.

motivation:

The challenge of creating the vertically integrated technology and software needed to drive a data-intensive collaboratory for particle and nuclear physics is daunting. *PPDG* is providing a practical set of Grid-enabled tools that meet the deployment schedule of the HENP experiments.

related work:

▷*GriPhyN*

citation:

[150], [239]

details:

The *PPDG* work plan focuses on several distinct areas as follows:

- deployment, and where necessary enhancement or development of distributed data management tools:
  - distributed file catalog and web browser-based file and database exploration toolset
  - data transfer tools and services
  - storage management tools
  - resource discovery and management utilities
- instrumentation needed to diagnose and correct performance and reliability problems

- deployment of distributed data services (based on the above components) for a limited number of key sites per physics collaboration
- exploratory work with limited deployment for advanced services

The principal work areas:

- obtaining, collecting and managing status information on resources and applications
- storage management services in a Grid environment
- reliable, efficient and fault-tolerant data movement
- job description languages and reliable job control infrastructure for Grid resources

SETI@home (Search for Extraterrestrial Intelligence)

contact:

Diane Richards, `pio@seti.org`

url:

`http://www:seti.org`

description:

SETI@home uses computers in homes and offices around the world to analyze radio telescope signals.

motivation:

SETI is a scientific area whose goal is to detect intelligent life outside Earth [325].

citation:

[23], [318]

details:

The signal data is divided into fixed-size *work units* that are distributed, via the Internet, to a client program running on numerous computers. The client program computes a result (a set of candidate signals), returns it to the server, and gets another work unit. There is no communication between clients. *SETI@home* does redundant computation: each work unit is processed multiple times. As seen in Figure 3.10 after recording and splitting signal data a relational *database server* is used to store information about tapes, workunits, results, users, and other aspects of the project. A

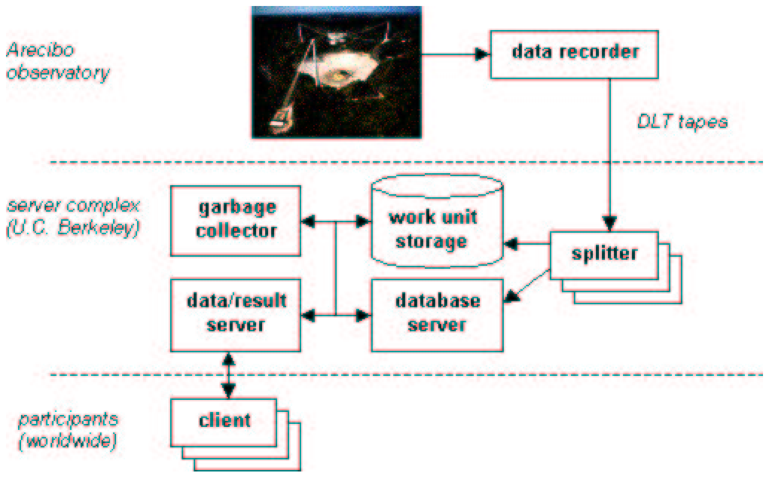


Figure 3.10: SETI@home: Data Distribution

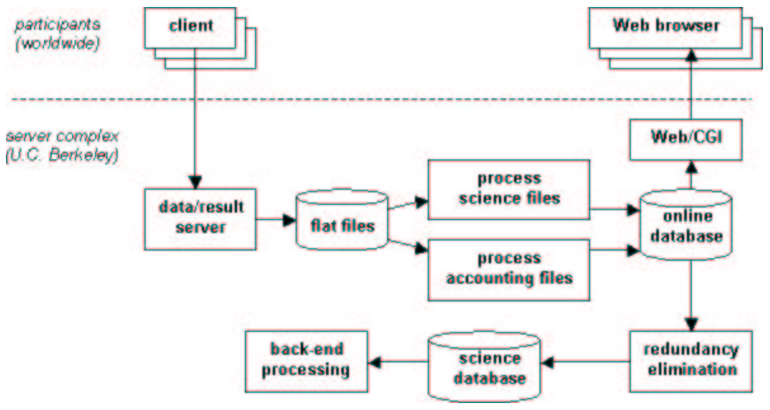


Figure 3.11: SETI@home: Collection and Analysis of Results

multi-threaded *data/result server* distributes work units to clients. Work units are 350 KB - enough data to keep a typical computer busy for about a day, but small enough to download over a slow modem in a few minutes. A *garbage collector* program removes work units from disk, clearing an on-disk flag in their database records. Results are returned to the *SETI@home server complex*, where they are recorded and analyzed (see Figure 3.11). Scientific and accounting results are processed and a *redundancy elimination* program examines each group of redundant results.

UNICORE (UNiform Interface to COmputing REsources)

contact:

Dietmar Erwin, `D.Erwin@fz-juelich.de`

url:

`http://www.unicore.org`

description:

UNICORE provides a science and engineering Grid combining resources of supercomputer centres and making them available through the Internet.

motivation:

The goal of *UNICORE* is to develop a Grid infrastructure together with a computing portal for engineers and scientists to access supercomputer centres from anywhere on the Internet.

related work:

▹*EUROGRID*, GRIP [164]

citation:

[122], [257]

details:

The system architecture of *UNICORE* is illustrated in Figure 3.12. The *UNICORE client* enables the user to create, submit and control jobs from any workstation or PC on the Internet. The client connects to a *UNICORE gateway* which authenticates both client and user, before contacting the *UNICORE servers*, which in turn manage the submitted *UNICORE jobs*. They incarnate abstract tasks destined for local hosts into batch jobs and run them on the native batch subsystem. Tasks to be run at a remote site are transferred to a peer *UNICORE gateway*. All necessary data transfers and synchronizations are performed by the servers. They also retain status information and job output, passing it to the client upon user request. The protocol between the components is defined in terms of *Java objects*. A low-level layer called the *UNICORE Protocol Layer (UPL)* handles authentication, SSL communication and transfer of data as inlined byte-streams and a high-level layer (the *Abstract Job Object (AJO)* class library) contains the classes to define *UNICORE jobs, tasks* and *resource requests*.

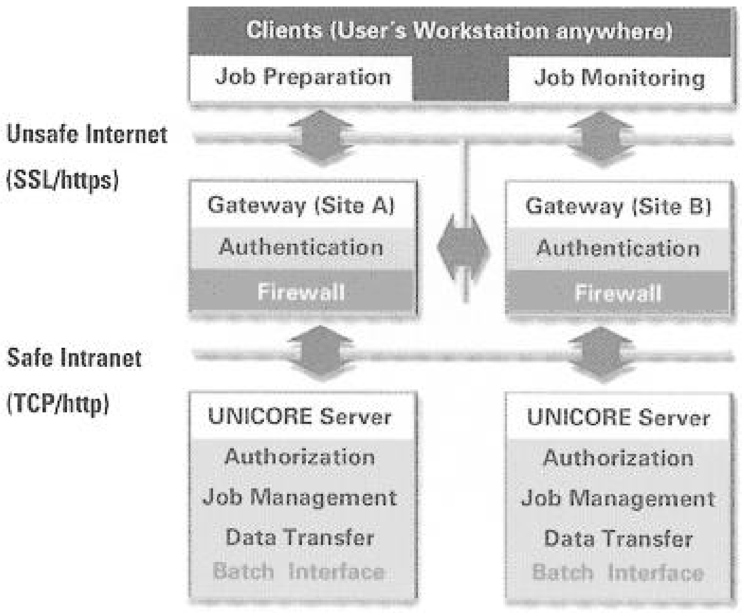


Figure 3.12: UNICORE: System Architecture

VDG (Virtual Data Grid)

contact:

Ian Foster, `foster@mcs.anl.gov`

description:

A *VDG* is the result of a more expansive view of a data system architecture based on an integrated treatment of not only data but also the computational procedures used to manipulate data and the computations that apply those procedures to data. In such a *VDG*, data, procedures, and computations are all first class entities, and can be published, discovered, and manipulated. It is called *virtual* because it allows for the representation and manipulation of data that does not exist, being defined only by computational procedures.

motivation:

Research communities require a scalable system for managing, tracing, communicating, and exploring the derivation and analysis of diverse data objects.

features:

- general but powerful abstractions for representing data and computation
- *VDG* architecture on top of ▶*OGSA*

related work:

▶*Giggle*, ZOO [183]

citation:

[142]

details:

The system has two components:

Virtual Data Schema The virtual data schema defines the data objects and relationships of the virtual data model. An implementation within a particular virtual data service instance might be a relational database, object-oriented database, XML repository, or even a hierarchical directory such as a file system or LDAP database.

Datasets A dataset is the unit of data managed within the virtual data model. This abstraction allows for the tracking of data in more general forms than files, tables, etc., and to insulate users from low-level data representations. A dataset is the unit of data manipulated by a *transformation* and the unit of data that may be stored in any of a variety of containers.

Types Each dataset has a type, which specifies various characteristics of the dataset, including how it is structured or represented on storage or data servers and what kind of data it contains.

Transformations A transformation is a typed computational procedure that may take as arguments both strings passed by value and datasets passed by reference. A transformation may create, delete, read, and/or write datasets passed as arguments. There are *simple transformations*, which act as a black box, and *compound transformations*, which compose one or more transformations.

Derivation A derivation specializes a transformation by specifying the actual arguments (strings and/or datasets) and other information (e.g., in some situations, environment variable values) required to perform a specific execution of its associated transformation. A derivation record can serve both as a historical record of what was done and as a recipe for operations that can be performed in the future.

Invocation An invocation specializes a derivation by specifying a specific environment and context (e.g., date, time, processor, OS) in which its associated derivation was executed.

Virtual Data System The purpose of the Virtual Data System is to maintain and provide access to the information in the Virtual Data Schema in a distributed, multi-user, multi-institutional environment, to address the larger goals of scalability, manageability, and support for discovery and sharing.

It consists mainly of two components:

Virtual Data Catalog (VDC) The term virtual data catalog denotes a service that maintains information defined by the virtual data schema. A VDC is, in general, an abstract notion: the VDC contents will typically be distributed over multiple information resources with varying degrees of authenticity and coherency.

Federated Indices Federated indices integrate information about selected objects from multiple VDCs. Presumably such federating indices would be differentiated according to their scope (user interest, all community data, community approved data, etc.), accuracy (depth of index, update frequency), cost, access control etc.

The realization of the concepts described above requires a variety of enabling infrastructure, including mechanisms for establishing inter-catalog references (and, in general, for naming *VDG* entities), establishing identity and authority, service discovery, virtualizing compute resources, and so forth. The current prototype builds on ▷*Globus* Toolkit v2 technology and the intention of the developers is to build future systems on the Grid infrastructure defined within ▶*OGSA* and implemented by the ▷*Globus* Toolkit v3.

VDT (Virtual Data Toolkit)

contact:

`vdt-support@ivdgl.org`

url:

`http://www.lsc-group.phys.uwm.edu/vdt/`

description:

VDT is a set of software that supports the needs of the research groups and experiments involved in the ▷*GriPhyN* project.

The *VDT* consists of three pieces, the *server*, the *client*, and the *SDK*. The *server* contains the ▷*Globus* gatekeeper, ▷*Condor*, ▶*Chimera*, etc. The *client* contains software for running jobs at a remote Grid site. The *SDK* contains libraries to develop new software. For a complete reference see [374].

application:

▷*EDG*, ▷*GriPhyN*, iVDGL [186]

related work:

▷*Condor*, ▷*Globus*, ▷*GriPhyN*

citation:

[375]

Chapter 4

Dictionary

A

Abacus

type:

Filter

contact:

Khalil Amiri, `amiri@cs.cmu.edu`

url:

`http://www.pdl.cmu.edu/Abacus/index.html`

description:

Abacus is a run-time system that monitors and dynamically changes function placement for applications that manipulate large datasets.

motivation:

Abacus aims to effectively utilize cluster resources.

features:

- migration and location-transparent invocation
- resource monitoring and management

application:

object-based distributed file system built for *Abacus* [1, figure 3]

related work:

River [32]

citation:

[21], [273], [299]

details:

Abacus consists of a programming model and a run-time system. The *Abacus* programming model encourages the programmer to compose data-intensive applications from small, functionally independent components or objects. These *mobile objects* provide explicit methods which *checkpoint* and *restore* their state during migration. Figure 4.1 represents a sketch of objects in *Abacus*. Objects have private state that is only accessible through their exported interface. The private state can contain references to *embedded objects*, or to *external objects*. *Anchored objects* include storage objects which provide persistent storage. A part of each application is usually anchored to the node where the application starts. The console is usually not data-intensive but serves for initialization and user/system interface functions.

The *Abacus* run-time system consists of a *Migration and Location-Transparent Invocation Component*, and a *Resource Monitoring and Management Component*. The first component is responsible for the creation of location-transparent references to mobile objects, for the redirection of method invocations in the face of object migrations, and or enacting object migrations. For example, Figure 4.2 shows a filter accessing a striped file. Functionality is partitioned into objects. Inter-object method invocations are transparently redirected by the location transparent invocation component of the *Abacus* run-time, which also updates the resource monitoring component on each procedure call, and return from a mobile object (arrows labeled ”U”). Clients periodically send digests of the statistics to the server. Resource managers at the server collect the relevant statistics and initiate migration decisions (”M”).

The *Resource Monitoring and Management Component* uses the notifications to collect statistics about bytes moved between objects and about the resources used by active objects. Moreover, this component monitors the availability of resources throughout the cluster. An analytic model is used to predict the performance benefit of moving to an alternative placement.

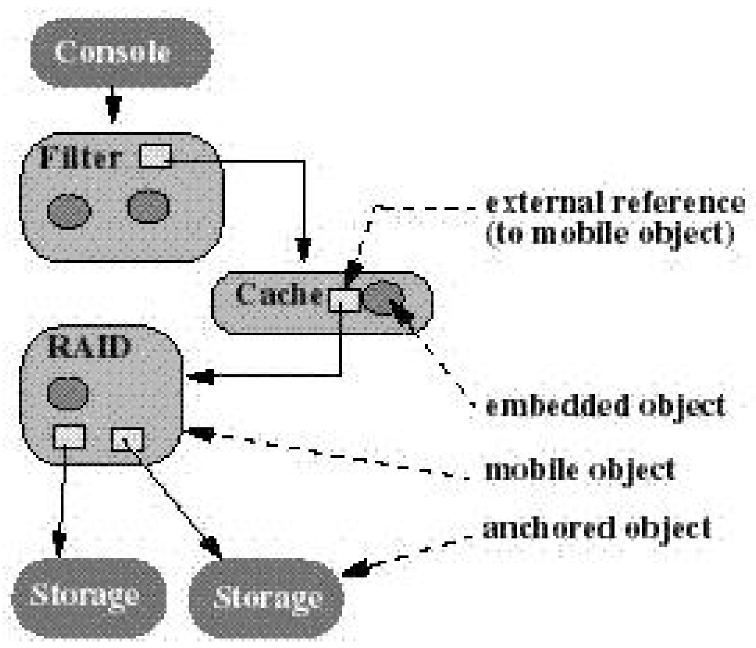


Figure 4.1: Abacus: Objects

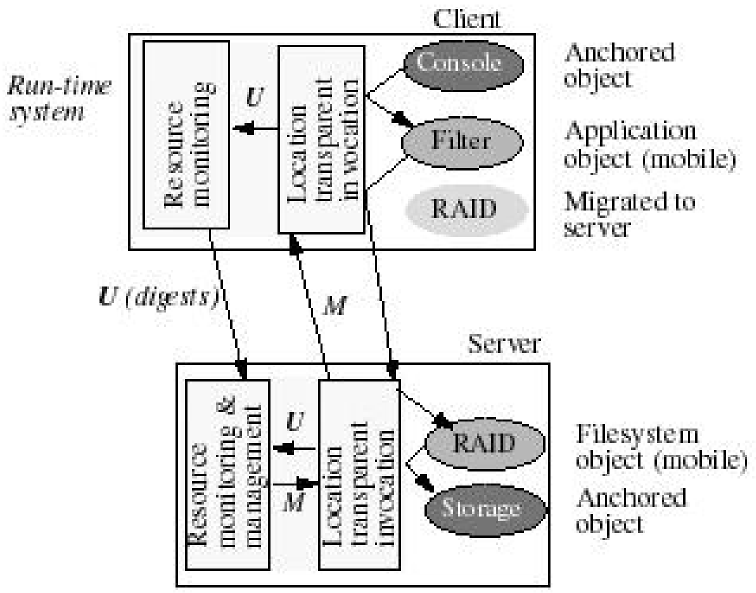

Client
Run-time system
Resource monitoring
U
Location transparent invocation
Console
Filter
RAID
Anchored object
Application object (mobile)
Migrated to server
U (digests)
M
Server
Resource monitoring & management
U
M
Location transparent invocation
RAID
Storage
Filesystem object (mobile)
Anchored object

Figure 4.2: Abacus: Run-time System

Active Buffering

type:

Application Level Method

contact:

Marianne Winslett, `winslett@uiuc.edu`

description:

Active Buffering is a method in which processors actively organize their idle memory into a hierarchy of buffers for periodic output data.

motivation:

Efficient transfer of output from main memory to secondary storage is very important to achieve high-performance for scientific applications.

features:

- the buffering scheme automatically adjusts to available memory space
- implementation in ►*Panda* available

application:

►*Panda*

related work:

[85] focuses on buffering for the purpose of aggregating small or non-continuous writes into long, sequential writes for better output performance. In contrast, *Active Buffering* tries to hide the cost of writing by increasing the overlap between I/O and communication, instead of trying to speed up the actual writes.
Active Buffering with Threads [220], *Active Buffering* plus Compressed Migration [207]

citation:

[219]

details:

To take full advantage of the parallelism between I/O activities and computation extra processors called *I/O processors* are used as dedicated writers which run the server executable of the ►*Collective I/O* library.

The clients periodically write out snapshot or checkpoint data using collective write calls. In such a call, the clients send a write request to the servers and exchange information with them on how to carry out the write operation. Then each client copies as much of its output data as possible into local buffers and sends overflowing data to the server(s).

The servers listen for requests from the clients. When one arrives, each server collects the data it is responsible for from the appropriate clients by explicit messages or using one-sided communication, and writes them to disk.

Both the servers and the clients utilize available local memory for *Active Buffering*. The servers can use most of their memory for this purpose. The clients can use idle memory not used by the compute application.

If client-side buffers have room for all the output, the clients will return to computation as soon as their data are copied to local buffers and the immediately visible I/O cost is only the cost of that copying. If the amount of output data exceeds local buffer capacity, the overflow will be sent to the servers using MPI [243] messages and the immediately visible I/O cost will include both copying and message passing costs. Further, if the total amount of overflow sent to a server exceeds its buffer capacity, it has to write out data to make room for new incoming data and the immediately visible I/O cost will include the costs of local copying, message passing, and file system requests.

However, because dedicated I/O servers are not always available or convenient to use, the authors investigate using *Active Buffering* with threads to hide ▶*Collective I/O* cost, instead of using dedicated I/O servers [220].

In [207] Lee et al. propose a novel execution environment that integrates *Active Buffering* and data migration with compression.

Active Disk

type:

Storage System

contact:

Christos Faloutsos, `christos@cs.cmu.edu`

url:

`http://www.pdl.cmu.edu/Active/`

description:

An *Active Disk* storage device combines on-drive processing and memory with software downloadability to allow disks to execute application-level functions directly at the device.

motivation:

Active Disks are able to leverage the processing power of tens or 100s of disks, which can more than compensate for the lower relative MIPS of single drives compared to host processors.

features:

- leverage the parallelism available in systems with many disks
- operate with a small amount of state, processing data as it streams off the disk
- execute relatively few instructions per byte of data

application:

►*ADFS*

related work:

►*ADFS*, NASD [151]

citation:

[290], [291], [371]

details:

Most current-generation drives fit all core drive-control and communications functions into a single ASIC. A specialized drive circuitry occupies approximately one-quarter of the chip, leaving sufficient area to include a 200-MHz ARM [29] core or similar embedded microprocessor. Processing power and memory inside disk drives currently optimize functions behind standardized interfaces such as SCSI or ATA, but limiting the interface to low-level, general-purpose tasks also limits the possible benefits of that processing power. With *Active Disks*, the rigid interface is broken and the excess computation power in drives is directly available for application-specific functions. The most compelling use of such processing power that scales with data size is large parallel scans.

AdaptRaid

type:

Storage System

contact:

Toni Cortes, `toni@ac.upc.es`

url:

`http://people.ac.upc.es/toni/AdaptRaid.html`

description:

AdaptRaid is a block-distribution algorithm that can be used to build disk arrays from a heterogeneous set of disks.

motivation:

Heterogeneous disk arrays are becoming a common configuration in many sites. To handle this kind of disk arrays, current systems do not take into account the differences between the disks. All disks are treated as if they had same capacity (the smallest one) and performance (the slowest one). This is not the best approach because improvements in both capacity and response time of heterogeneous arrays could be achieved if each disk were used accordingly to its characteristics.

features:

- works on any kind of disk array (hardware or software, tightly or loosely coupled)
- works with RAID0 (Striping) and RAID5 (Block-Interleaved Distributed-Parity) [83]

related work:

HP AutoRAID [402]

citation:

[96]

details:

Following the idea to place more data blocks in the larger disks than in the smaller ones, all D disks (as in a regular RAID0) are used for as many stripes as blocks can fit in the smallest disk. Once the smallest disk is full, the rest of the disks are used as if there is a disk array with $D-1$ disks. This distribution continues until all disks are full with data. A side effect of this distribution is that the system may have stripes with different lengths. Figure 4.3 shows the distribution of blocks in a five-disk array where disks have different capacities.

To evenly distribute the location of long and short stripes all over the array and reduce the variance between the accesses in the different portions of the disk array the concept of *pattern of stripes* is introduced. The algorithm assumes, for a moment, that disks are smaller than they actually are (but with the same proportions in size)

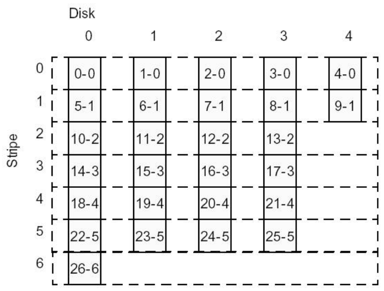


Figure 4.3: AdaptRaid: Distribution of Data Blocks

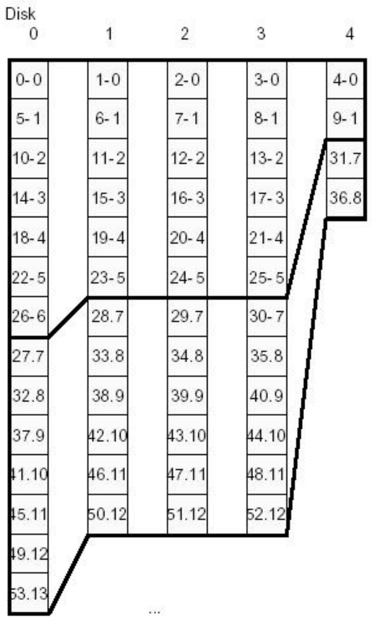


Figure 4.4: AdaptRaid: Example of Pattern Repetition

and distributes the blocks in this smaller array. This distribution becomes the pattern that is repeated until all disks are full. The resulting distribution has the same number of stripes as the previous version of the algorithm. Furthermore, each disk also has the same number of blocks as in the previous version. The only difference is that short and long stripes are distributed all over the array, which was the objective. An example of this pattern repetition can be seen in Figure 4.4.

A similar algorithm exists for RAID5. The only difference is that stripes will have one less data block because one block is needed to keep the parity for each stripe. A few small optimizations have also been included to avoid increasing the number of small writes.

ADFS (Active Disk-Based File System)

type:

File System

contact:

Hyeran Lim, `hrlim@cs.umn.edu`

description:

ADFS is a file system for the ▶*Active Disk*-based data server. All data files stored on ▶*Active Disks* are provided with operations, forming objects.

motivation:

ADFS was developed to reduce the application-processing overhead of the system by running application-specific operations by disk processors.

application:

CORBA simulation of the *Active Disk* based file system [214, page 111]

related work:

▶*Active Disk*

citation:

[214]

details:

The *Central File Manager, Clients*, and a *Group of* ▶Active Disks are attached to a high-speed network. The file manager will be involved only in system-wide decisions such as creation or deletion of classes and creation/deletion/replication of an object. ▶*Active Disks* take over the most of the functionality of the file manager and present the file system based interfaces to clients, and clients cooperate with the ▶*Active Disks* by embedding the part of file system interfaces. A set of file system client-server modules is shown in Figure 4.5.

In *ADFS*, each file is seen as a named object with operation codes combined. When some objects represent the same characteristics and share the same operations but their data components are different, the objects form a class of the type. The data stored in an ▶*Active Disk* is seen as objects to clients and the central file manager. A request is processed through several layers with libraries available to each layer. The layers are ▶*Active Disk, Object-Oriented Disk (OOD)*, and *Block Oriented Disk (BOD)*, where BOD is the abstract representation of the traditional disk that exports a block-oriented view to the other components. The OOD is implemented on top of

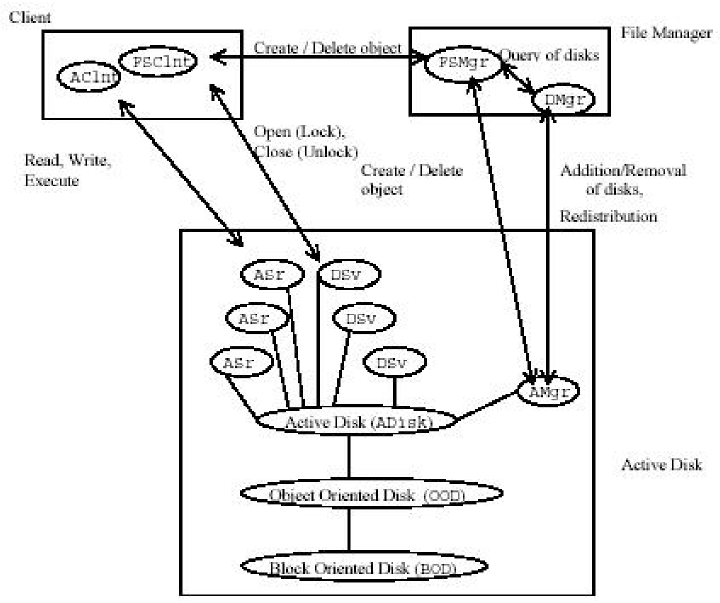


Figure 4.5: ADFS:Architecture

BOD, providing an object-oriented view of data stored on BOD. Clients are allowed to access data located by offsets with an *Object Identifier (OID)*. The Active module (*ADisk*) is implemented on top of OOD. All servers running on an *Active Disk* share ADisk to communicate with OOD and serve their duties to clients or the central file manager. On top of ADisk several *Active Disk Server (ASrv)* and *File System Server (DSrv)* are running. *ASrv* is responsible for serving data requests from clients. *DSrv* is a file system server also running on top of ADisk, which serves individual file system services to clients. The *Active Disk Manager (AMgr)* is responsible for maintaining storage space in an ▶*Active Disk* by creating and deleting objects on the ▶*Active Disk*.

The *File Manager* has two main functions: disk management and overall file system operations. The *Disk Manager (DMgr)* keeps track of the ▶*Active Disks* currently in the system. It is the responsibility of DMgr to support dynamic disk management. The *File System Manger (FSMgr)* needs to make system-wide decisions such as file creation. Two *ADFS* components running on the kernel of a client are *Active Disk Client (AClnt)* and *File System Client (FSClnt)*. *AClnt* is responsible for manipulating data stored on ▶*Active Disks*. *FSClnt* provides the file system functionality to client programs. It interacts with DSrv on the ▶*Active Disks* where the accessed objects lie. It contacts the File Manager when it creates a new file.

ADIO (Abstract Device Interface for I/O)

type:

I/O Library

contact:

Rajeev Thakur, `thakur@mcs.anl.gov`

url:

`http://www.mcs.anl.gov/~thakur/adio/`

description:

ADIO is a strategy for implementing any user-level Parallel I/O interface portably on multiple file systems.

motivation:

A limiting factor in I/O-intensive computing is the lack of of a standard, portable API for Parallel I/O. Instead of a single standard API, a number of different APIs are supported by different vendors and research projects. The goal of *ADIO* is to facilitate a high-performance implementation of any existing or new Parallel I/O API on any existing or new file-system.

features:

- *ADIO* uses MPI [243] for portability and high-performance wherever possible
- very low overhead

application:

►*RIO*

related work:

MPICH [165] uses a similar abstract-device interface approach.

citation:

[352]

details:

As illustrated in Figure 4.6 *ADIO* consists of a small set of basic functions for performing Parallel I/O. Any Parallel I/O API (including a file-system interface) can be implemented in a portable fashion on top of *ADIO*. *ADIO* in turn must be implemented in an optimized manner on each different file system separately. In other words, *ADIO* separates the machine-dependent and machine-independent aspects involved in implementing an API. The machine-independent part can be implemented

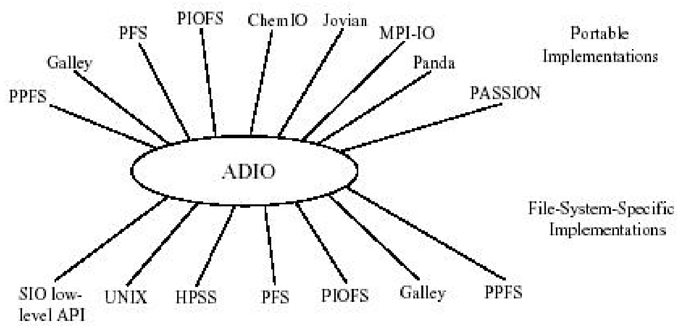


Figure 4.6: ADIO Concept

portably on top of *ADIO*. The machine-dependent part is *ADIO* itself, which must be implemented separately on each different system.

ADR (Active Data Repository)

type:

Mass Storage System

contact:

Joel Saltz, `saltz-1@medctr.osu.edu`

url:

`http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/adr/`

description:

ADR is an object-oriented framework designed to efficiently integrate application-specific processing with the storage and retrieval of multi-dimensional datasets on a parallel machine with a disk farm.

motivation:

ADR optimizes storage, retrieval and processing of very large multi-dimensional datasets.

application:

Bay and Estuary Simulation [50], Titan [366], Virtual Microscope [3], Volume Visualization [382]

related work:

►*DataCutter*

citation:

[78], [129], [202], [203]

details:

ADR consists of a set of modular services, implemented as a C++ class library, and a run-time system. Several of the services allow customization for user-defined processing. An application developer has to provide accumulator data structures, and functions that operate on *in-core* data to implement application-specific processing of *out-of-core* data (►*EM (External Memory) Algorithms and Data Structures*). An application implemented using *ADR* consists of one or more *clients, a front-end process,* and a customized *back-end*. See Figure 4.7. The shaded bars represent functions added to *ADR* by the user as part of the customization process. Client A is a sequential program while client B is a parallel program.

The *front-end* interacts with clients, translates client requests into queries and sends one or more queries to the parallel back-end. The *back-end* is responsible for storing datasets and carrying out application-specific processing of the data on the parallel machine. The customizable *ADR* services in the back-end include:

1. an *attribute space service* that manages the registration and use of user-defined mapping functions
2. a *dataset service* that manages the datasets stored in the *ADR* back-end and provides utility functions for loading datasets into *ADR*
3. an *indexing service* that manages various indices for the datasets stored in *ADR*
4. a *data aggregation* service that manages the user-provided functions to be used in aggregation operations, and functions to generate the final outputs

AJO (Abstract Job Object) see ▷**UNICORE**

April

type:

Data Access System

contact:

Gokhan Memik, `memik@ece.nwu.edu`

description:

April is a run-time library for tape-resident data.

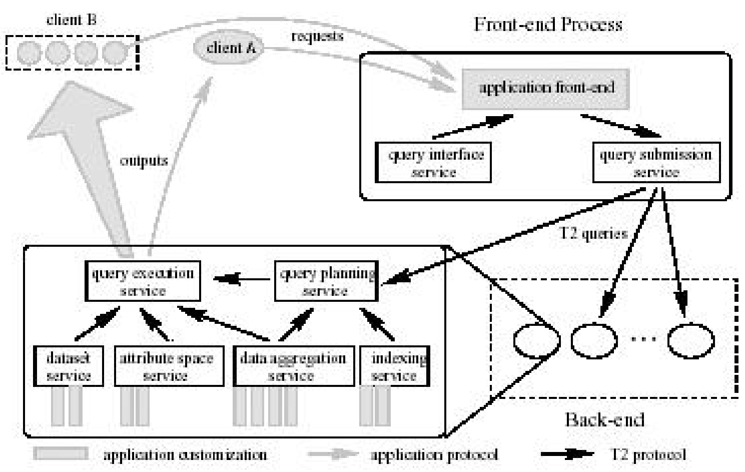


Figure 4.7: ADR: Application Suite

motivation:

April allows programmers to access data located on tape via a convenient interface expressed in terms of arrays and array portions (regions) rather than files and offsets.

features:

- prestaging
- migration
- ►*Sub-Filing*

related work:

►*Sub-Filing*

citation:

[229]

details:

The library provides routines to efficiently perform I/O required in sequential and parallel applications. It can be used for both *in-core* and *out-of-core* applications (►*EM (External Memory) Algorithms and Data Structures*).

It provides a portable interface on top of ►*HPSS* and ►*MPI-IO*. It can also be used by an optimizing compiler that targets programs whose datasets require transfers between secondary storage and tertiary storage. It might even be possible to employ the library within a database management system for multi-dimensional data.

The library uses an optimization technique called ►*Sub-Filing*, ►*Collective I/O* using a ►*Two-Phase I/O* method, data prestaging, ►*Prefetching*, and data migration.

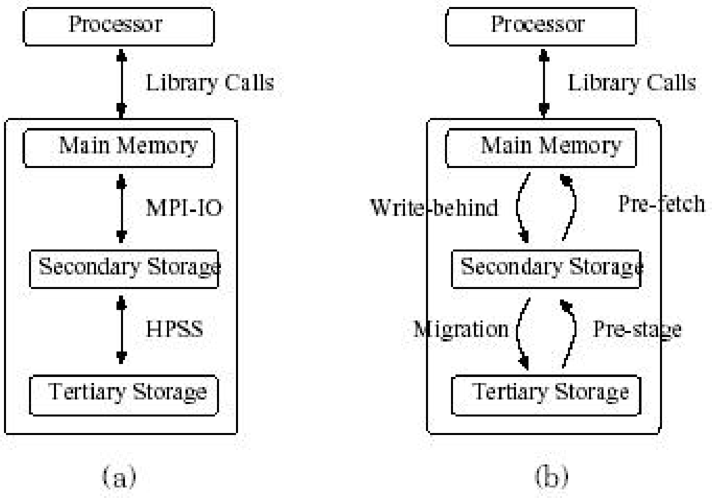


Figure 4.8: April: Library Architecture and Prefetching, Prestaging, Migration

▶*Sub-Filing* is invisible to the user and helps to efficiently manage the storage hierarchy which can consist of a tape sub-system, a disk sub-system and a main memory. The main advantage of the ▶*Collective I/O*, on the other hand, is that it results in high-granularity data transfers between processors and disks, and it also makes use of the higher bandwidth of the processor interconnection network. Computation and tape I/O is overlapped by prestaging by bringing the required data ahead of the time it will be used. It issues asynchronous read calls to the tape sub-system, which helps to overlap the reading of the next data portion with the computation being performed on the current dataset. The data ▶*Prefetching* is similar except that it overlaps the disk I/O time with the computation time. The connections between different components are shown in Figure 4.8.

Armada

type:

Intelligent I/O System

contact:

Ron Oldfield, `raoldfi@sandia.gov`

url:

`http://www.cs.dartmouth.edu/~dfk/armada/`

description:

The *Armada* framework for Parallel I/O provides a solution for data-intensive applications in which the application programmer and dataset provider deploy a network of application-specific and dataset-specific functionality across the Grid.

motivation:

The goal of *Armada* is to develop an I/O framework that allows data-intensive applications to efficiently access geographically distributed datasets.

features:

- ability to restructure the application graph
- flexibility
- remote processing of application code

application:

remote file copy [261, 5.1], seismic imaging [261, 5.2]

related work:

▶*DataCutter*, ▶*dQUOB*, HFS [200], PS [231], wrapFS [411]

citation:

[260], [261], [262], [263]

details:

Using the *Armada* framework [261, 262], Grid applications access remote data by sending data requests through a graph of distributed application objects called *ships*. From an operational perspective, requests flow (in a pipelined manner) from the client processors, through the ships, to the data servers. Data then flows back to the clients for reads, or toward the data servers for writes. A typical graph consists of two distinct portions: one that describes the layout of the data (as defined by the data provider), and one that describes the interface and processing required by the application. *Armada* appends the two portions to form a single graph, restructures the resulting graph to distribute computation and data flow, and assigns ships to Grid processors so that *data-reducing ships* execute near the data source and *data-increasing ships* execute near the data destination.

Figure 4.9 shows a typical *Armada* graph for an application accessing a replicated and distributed dataset. It consists of a portion from the data provider that describes the layout of the data and a portion from the application that describes required preprocessing. As depicted in Figure 4.10 *Armada* restructured the original graph to provide end-to-end parallelism. It moved the filter code close to the data source,

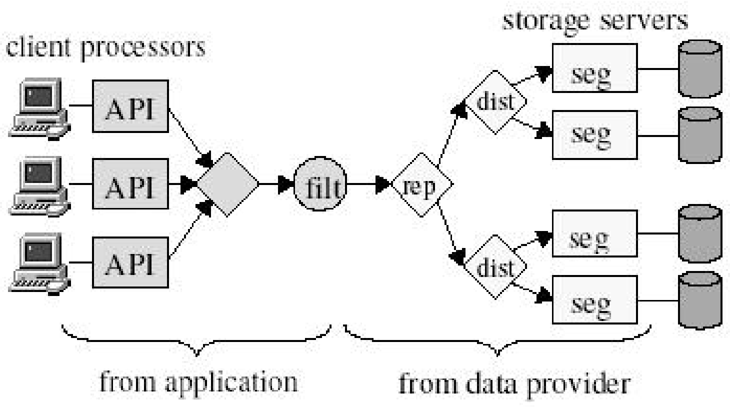


Figure 4.9: Armada: Typical Graph

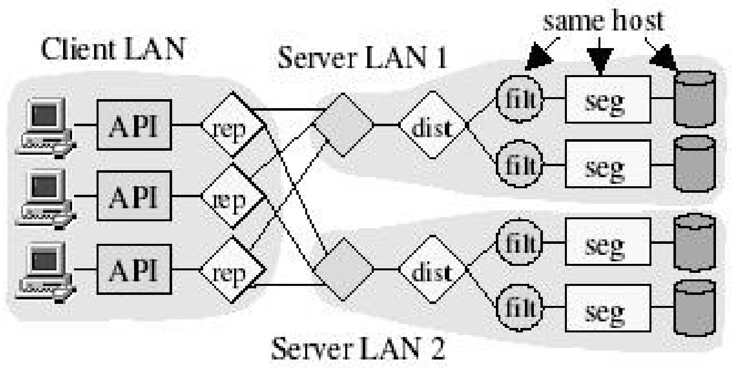


Figure 4.10: Armada: Restructured Graph

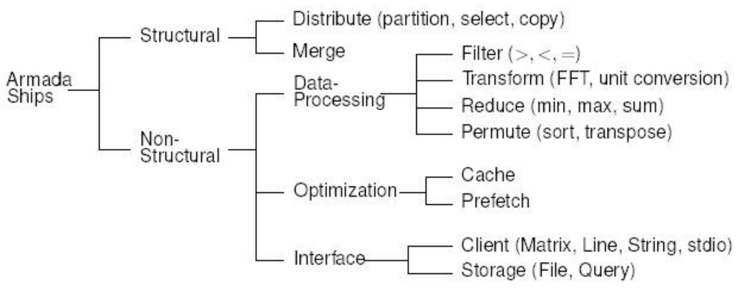


Figure 4.11: Armada: Hierarchy of Ships

and it partitioned the graph into appropriate administrative domains (the grey blobs represent the three LANs used by the application).

The *Armada* framework includes a rich set of extensible ship classes (shown in Figure 4.11) divided into two primary categories: *structural* and *non-structural*. *Structural ships* allow one-to-many and many-to-one connections in the *Armada* graph. They provide functionality to distribute or merge sequences of requests and data. *Non-structural ships* process and generate single sequences of requests and data. Non-structural ships provide functionality for data processing and I/O optimization, and are also used as interfaces for client applications and low-level storage servers.

A key feature of *Armada* is the ability to restructure the application graph [264]. Unlike existing database systems, that restructure graphs based on well-known relational operators, *Armada's* restructuring algorithm uses programmer-defined properties to allow restructuring of a wide range of application classes (not just relational databases).

Another important feature of the *Armada* system is placement. Modules that make up the application graph execute on nodes near the client, nodes near the data, or intermediate network processors. Our approach is to treat placement as a hierarchical graph-partitioning problem. Using the Chaco graph partitioning software [172], the graph is first partitioned into administrative domains in an attempt to minimize data transferred between domains. Then application modules in each domain are partitioned to processors provided by domain-level resource managers. See [259] for details of the placement algorithm.

ASU (Active Storage Unit) see ***Active Storage***

Asynchronous Parallel Disk Sorting see ***greedyWriting***

Automatic GASS see ***GASS***

C

CASTOR (CERN Advanced STORage Manager)

type:

Mass Storage System

contact:

`castor.support@cern.ch`

url:

`http://cern.ch/castor`

description:

CASTOR is a *Hierarchical Storage Management (HSM)* system used to store physics production files and user files. *CASTOR* manages disk cache(s) and the data on tertiary storage or tapes.

motivation:

The goal of the *CASTOR* project is to handle LHC [211] data in a fully distributed environment.

application:

LHC [211]

related work:

►*DPSS*, ►*Enstore*, ►*HPSS*

citation:

[49]

details:

The *CASTOR* servers are all installed in the computer centre, while the *CASTOR* client software is installed on most of the desktops at CERN. The main access to data in *CASTOR* is through the use of *RFIO (Remote File I/O)*. RFIO provides access to local files, remote files not residing on the users' machine and HSM files.

The *STAGER* has the primary role of a disk pool manager whose functions are to allocate space on disk to store a file, to maintain a catalog of all the files in its disk pools and to clear out old or least recently used files in these pools when more free space is required, a process known as *garbage collection*. A *DISK POOL* is simply a collection of file systems - from one to many. The STAGER must be the only process that creates or deletes files in its disk pools so that it knows at all times the amount of free space in each of its disk pools. The STAGER may be remote from the disk pools and itself uses the RFIO mechanisms to handle file creation and deletion in the DISK POOLS. The principal role of the *NAME server* is to implement a hierarchical view of the *CASTOR* name space so that it appears that the files are in directories.

RTCOPY (Remote Tape COPY) is another important *CASTOR* component whose function is to transfer data from disk to tape or vice versa, normally as part of a *migration* of disk files to tape or as a recall from tape to disk. As a user you have no way to select a tape drive for your use. Instead a server called *Volume Drive Queue Manager (VDQM)* must be contacted. It knows about the available drives and their status and which tape volumes are currently in use.

Tape Management System (TMS), an old system, has been replaced by the *Volume Manager (VMGR)*. Before a request is sent to mount a tape, the RTCOPY client calls the VMGR to make sure the user has the rights to access the tape. The RTCPD child tells the *Tape Daemon (TPDAEMON)* to mount the tape volume and position. The tape daemon sends a message to the *Message Daemon (MSGDAEMON)* to tell the central computer operations staff to mount the tape or that there is a robotic tape request pending so that he can see if any error occurs. Figure 4.12 shows the complete *CASTOR* software layout as it currently exists.

Chaos

type:

Other / Project

contact:

Alan Sussman, `als@cs.umd.edu`

url:

`http://www.cs.umd.edu/projects/hpsl/chaos/`

description:

Chaos is a project which focuses on run-time support and programming models for data-intensive applications.

The origin of the name is a run-time library for parallelizing programs with completely irregular data distributions. That was the first work that really distinguished

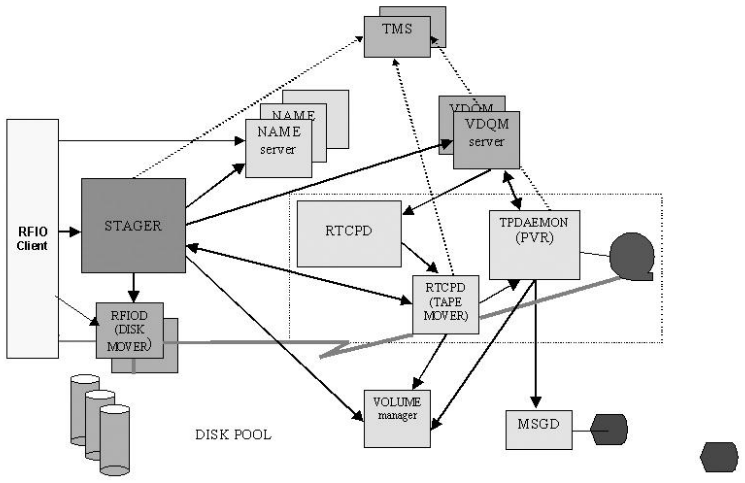


Figure 4.12: CASTOR: Layout

this group. At that time, and until a couple of years ago, the project leader was Joel Saltz, now at Ohio State University.

In particular the members concentrate on middleware for building data servers of various types, that store, retrieve and process large quantities of data.

Some of the projects are:
▶*ADR*
▶*Active Disk*
▶*DataCutter*

citation:
[79]

Chimera

type:

Other / Virtual Data System

contact:

Ian Foster, `foster@mcs.anl.gov`

description:

The *Chimera Virtual Data System (VDS)* provides a catalog that can be used by application environments to describe a set of application programs *transformations*, and then track all the data files produced by executing those applications *derivations*. *Chimera* contains the mechanism to locate the *recipe* to produce a given logical file, in the form of an abstract program execution graph.

motivation:

Much scientific data is not obtained from measurements but rather derived from other data by the application of computational procedures. In order to explore the benefits of data derivation tracking and virtual data management, *Chimera* was developed.

application:

analysis of data from the Sloan Digital Sky Survey [24]

related work:

ZOO [183]

citation:

[90], [141]

details:

The architecture of the *Chimera* VDS comprises two principal components:

- *Virtual Data Catalog(VDC)* This implements the *Chimera Virtual Data Schema*.
- *Virtual Data Language Interpreter* This implements a variety of tasks in terms of calls to virtual data catalog operations.

Applications access *Chimera* functions via a standard *Virtual Data Language (VDL)*, which supports both data definition and query statements. One important form of query returns a representation of the tasks that, when executed on a Data Grid, create a specified data product. Thus, VDL serves as a lingua franca for the *Chimera* ▷*VDG*, allowing components to determine virtual data relationships, to pass this knowledge to other components, and to populate and query the VDC without having to depend on the (potentially evolving) catalog schema.

The *Chimera* Virtual Data Schema defines a set of relations used to capture and formalize descriptions of how a program can be invoked, and to record its potential and/or actual invocations. The most important entities of the Virtual Data Schema are:

Transformation A transformation is an executable program. Associated with a transformation is information that might be used to characterize and locate it and information needed to invoke it.

Derivation A derivation represents an execution of a transformation. Associated with a derivation is the name of the associated transformation, the names of data objects to which the transformation is applied, and other derivation-specific information (e.g., values for parameters, time executed, execution time). While transformation arguments are formal parameters, the arguments to a derivation are actual parameters.

Data Object A data object is a named entity that may be consumed or produced by a derivation. In the applications considered to date, a data object is always a logical file, named by a *Logical File Name (LFN)*. A separate replica catalog or replica location service is used to map from logical file names to physical location(s) for replicas. However, data objects could also be relations or objects. Associated with a data object is information about that object: what is typically referred to as meta-data.

The VDL provides various commands for extracting derivation and transformation definitions. Since VDL is implemented in SQL, this query set is readily extensible. VDL query commands allow one to search for *transformations* by specifying a transformation name, application name, input LFN(s), output LFN(s), argument matches, and/or other transformation meta-data. One can search for *derivations* by specifying the associated transformation name, application name, input LFN(s), and/or output LFN(s).

Chirp

type:

Data Transfer Protocol

contact:

Dan Bradley, `dan@hep.wisc.edu`

details:

Chirp is a simple and lightweight remote I/O protocol. It may be used independently of ▷*Condor*, but one application of *Chirp* is to provide remote I/O services to Vanilla [372] and Java Universe [188] jobs in ▷*Condor*, since these types of jobs do not have direct access to ▷*Condor's* remote system call mechanism. Instead of accessing files directly, an application calls *Chirp* client functions which mimic the POSIX file operations, such as `open(), close()`, `read()`, and `write()`. The *Chirp* client communicates with a *Chirp* server, which access the requested files.

url:

`http://www.cs.wisc.edu/condor/chirp/`

related work:

▷*Condor*

Clusterfile

type:

File System

contact:

Florin Isaila, `florin@ira.uka.de`

url:

`http://www.ipd.uka.de/~florin`

description:

Clusterfile is a parallel file system which offers a high degree of control of the file layout over the cluster. It also allows applications to set arbitrary views on the files.

motivation:

Existing parallel file systems offer little control over matching the I/O access patterns and file data layout. Without this matching the applications may face the following problems: contention at I/O nodes, fragmentation of file data, false sharing, small network messages, high overhead of scattering/gathering the data. *Clusterfile* addresses some of these inefficiencies.

features:

- applications can physically partition a file in arbitrary patterns
- applications can set arbitrary views on files
- read and write operations are optimized by precomputing the direct mapping between access patterns and disks

related work:

►*CXFS*, ►*Frangipani*, ►*GPFS*

citation:

[185]

details:

Clusterfile has 3 main components:

Meta Data Manager There is one Metadata Manager running in the parallel file system. The Metadata Manager gathers periodically or by request information about a file from the I/O nodes and keeps them in a consistent state. It also offers per request services involving file metadata to the compute nodes. The Metadata Manager is not involved in the data transfer.

I/O Servers There is one I/O Server running on each I/O node in the parallel file system. The main task of the I/O Server is writing and reading the data to/from the subfiles

I/O Library Each compute node specifies operations on the file system by using an I/O Library. The I/O Library implements the Unix standard file system interface. The communication between the compute node and Metadata Manager or I/O Servers is hidden by the I/O Library from the applications.

The following changes will be incorporated into the next release of *Clusterfile*:

- ►*MPI-IO* interface and conversion of internal data types to MPI [243] data types
- ►*Two-Phase I/O* and ►*Disk-Directed I/O*
- cooperative caching [104]

Collective I/O

type:

Application Level Method

contact:

Rajesh Bordawekar, `rajesh@cacr.caltech.edu`

description:

Collective I/O is a technique in which system entities (e.g. compute clients, or I/O servers) share the data access and layout information and make coordinated, non-overlapping, and independent I/O requests in a conforming manner.

Global data access information may include array dimension and distribution information, while data layout information may include information about the file storage order and the file striping strategy. A conforming access pattern consists of one or more accesses, with each access reading or writing data from consecutive locations to/from a file or disk. In other words, the order in which data are fetched by each access matches the data storage pattern on file or disk.

There are three approaches to *Collective I/O*: ►*Two-Phase I/O*, ►*Disk-Directed I/O* and ►*Server-Directed I/O*.

In [228] Memik et al. introduce a new concept called *Multi-Collective I/O (MCIO)* that extends conventional *Collective I/O* to optimize I/O accesses to multiple arrays simultaneously. In this approach, as in ►*Collective I/O*, multiple processors coordinate to perform I/O on behalf of each other if doing so improves overall I/O time. However, unlike ►*Collective I/O*, MCIO considers multiple arrays simultaneously. It has a more global view of the overall I/O behaviour exhibited by application.

In [227] a *Compiler-Directed Collective I/O* approach is presented, which detects the opportunities for *Collective I/O* and inserts the necessary I/O calls in the code automatically.

related work:

Compiler-Directed *Collective I/O* [227], Multi-Collective I/O (MCIO) [228]

citation:

[63]

CXFS (Clustered Extended File System

type:

File System

url:

`http://www.sgi.com/products/storage/cxfs.html`

description:

SGI *CXFS* is a robust multi-OS shared file system for SANs. Based on the XFS file system [321, 337], *CXFS* provides a highly available, scalable, high-performance environment for sharing data between multiple OSs on a SAN. *CXFS* supports IRIX, Windows NT, Solaris, and will support other Unix platforms (including Linux).

motivation:

Traditional file serving and sharing methods such as NFS [302], CIFS/Samba [301] and FTP, which are still in use in SANs, have performance, availability, and complexity penalties. To completely remove these penalties and realize the promise of SAN technology, SGI has developed *CXFS*, a shared file system for SANs that allows multiple heterogeneous systems to simultaneously access all data on a SAN.

features:

- 64-bit scalability supports file sizes up to 9 million Terabytes, file systems up to 18 million Terabytes
- journaling for reliability and fast recovery
- POSIX compliant file locking
- supports memory-mapped files
- supports quotas
- dynamically allocated meta-data space
- centralized java-based management tools
- heterogenous client support

application:

Fleet Numerical Meteorology and Oceanography Center [131]

related work:

►*GPFS*, XFS [321, 337]

citation:

[319], [320]

details:

CXFS was designed as an extension to the XFS file system. *CXFS* has retained important XFS features while distributing I/O directly between disk and hosts.

CXFS disk volumes are constructed using the the SGI *Volume Manager (XFM)*. XVM augments the earlier volume manager by providing disk striping, mirroring, and concatenation in any possible combination.

While file data flows directly between systems and disk, *CXFS* meta-data is managed using a client-server approach. The *Metadata Server* acts as a central clearinghouse for meta-data logging, file locking, buffer coherency, and other necessary coordination functions.

To manage and control file meta-data and file access *CXFS* uses *tokens*. There are a number of tokens for each file being accessed via *CXFS*. These represent different aspects of the file – timestamps, size, extents, data, etc.

POSIX, BSD, and SVR4 file locks are implemented by forwarding client lock requests to the Metadata Server with RPCs. The server maintains information about all locks being held. Locks behave the same whether the locking processes run on a single host or on multiple hosts. No changes are required for an application to use locking with *CXFS*.

The health of all systems sharing a file system is monitored across the TCP/IP network using standard messaging and heartbeat. Should a system failure be detected, the Metadata Server will automatically take steps to recover. The Metadata Server inspects the XFS journal and rolls back any incomplete transactions the client had in progress. Should the Metadata Server on a particular host fail or be disabled, control will move to a host designated as a backup Metadata Server. This shift in control is transparent to user processes using *CXFS* files.

CXFS takes advantage of the resiliency of the XFS log-based file system (journaling). Metadata transactions are logged by the Metadata Server and mirrored for rapid and reliable recovery. XFS file system recovery voids the need for time-consuming file system checks during a recovery cycle after an ungraceful shutdown.

D

DAFS (Direct Access File System)

type:

Data Access System

contact:

Mark Wittle, `mwittle@netapp.com`

url:

`http://www.dafscollaborative.org/`

description:

DAFS is a new file-access protocol that is being designed to take advantage of new standard memory-to-memory interconnect technologies such as VI and InfiniBand [182] in data centre environments.

motivation:

DAFS is optimized for high-throughput, low-latency communication and for the requirements of local file-sharing architectures.

features:

- direct memory-to-memory transfer
- direct application access

citation:

[82], [108], [224]

details:

DAFS is implemented as a file access library, which will require a VI provider library implementation. Once an application is modified to link with *DAFS*, it is independent of the OS for its data storage. *DAFS* uses the underlying VI capabilities to provide direct application access to shared file servers. Local file sharing requires high-performance file and record locking to maintain consistency. *DAFS* allows locks to be cached so that repeated access to the same data need not result in a file server interaction, and when required by a node, a lock cached by another node is transferred without timeouts.

Direct memory-to-memory transfer and *direct application access* allows data to bypass the normal protocol processing. Applications can perform data transfer directly to VI-compliant network interfaces, without OS involvement. The data is transferred directly between appropriately aligned buffers on the communicating machines, and the VI host adapters perform all message fragmentation, assembly, and alignment in hardware and allow data transfer directly to or from application buffers in virtual memory.

DARC (Distributed Active Resource ArChitecture)

type:

Storage System

contact:

Craig J. Patten, `cjp@cs.adelaide.edu.au`

description:

DARC enables the development of portable, extensible and adaptive storage services in Grid environments which are independent of the underlying storage abstraction and end-user interface.

motivation:

DARC enables the production of distributed, cooperative data services.

application:

distributed file system prototype using *DARC* [269, 4.],
GMS-5 Satellite Imagery [269, 5.]

related work:

Strata is a distributed file system which has been developed using the *DARC* architecture [269].

citation:

[269]

details:

A *Node*, the central element of *DARC*, is a daemon on each host which wishes to participate in the architecture, and is the initial point of contact for establishing access to that host. Through the local Node, the *DataResource* provides remote access to some data storage and/or access mechanism. DataResources can represent some fixed local or higher-level mobile storage media. DataResources are non-server-centric providing a distributed presence at all data production and consumption points to improve flexibility and performance. When a client wishes to access some remote data service, its Node initiates a local instantiation of the specified DataResource. This entity then communicates with its remote peer instances, and potentially with other local DataResources.

The architecture provides a generic abstraction for DataResource communications through the Node. This abstraction which Nodes provide for bulk data transfers is called an *Xfer*, and for meta-data transfers, a *MetaXfer*. An Xfer provides a mechanism for a DataResource to transfer information to or from another. This information consists of the requested data's specification and source and destination host and resource identifiers.

Data Sieving

type:

Application Level Method

contact:

Rajeev Thakur, `thakur@mcs.anl.gov`

description:

Data Sieving is a technique that enables an implementation to make a few large, contiguous requests to the file system even if the user's request consists of several small, ▶*Noncontiguous Data Accesses*.

If a user has made a single read request for five noncontiguous pieces of data, instead of reading each noncontiguous piece separately, using *Data Sieving* the application reads a single contiguous chunk of data starting from the first requested byte up to the last requested byte into a temporary buffer in memory. It then extracts the requested portions from the temporary buffer and places them in the user's buffer.

The advantage of using *Data Sieving* to perform ▶*Noncontiguous Data Accesses* is that multiple ▶*Noncontiguous Data Accesses* can be described by a single I/O request. If the noncontiguous regions are nearby, the *Data Sieving* approach can eliminate many I/O requests. The *Data Sieving* approach can perform poorly, however, if the noncontiguous regions are far apart on disk. This access pattern will cause the single disk read to access a large amount of unused data that must move over the network. In general, using *Data Sieving* to perform noncontiguous I/O can benefit the user for ▶*Noncontiguous Data Access* patterns that have relatively densely packed regions of desired data.

application:

▶*ROMIO*

related work:

▶*Collective I/O*, ▶*Noncontiguous Data Access*, ▶*Two-Phase I/O*

citation:

[91], [349], [353], [355]

DataCutter

type:

Filter

contact:

Alan Sussman, `als@cs.umd.edu`

url:

`http://www.cs.umd.edu/projects/hpsl/ResearchAreas/DataCutter.htm`

description:

The *DataCutter* framework is a middleware infrastructure that enables processing of scientific datasets stored in archival storage systems across a wide-area network. It provides support for subsetting of datasets through multi-dimensional range queries, and application specific aggregation on scientific datasets stored in an archival storage system.

motivation:

The main design objective in *DataCutter* is to extend and apply the salient features of ▶*ADR* (i.e. support for accessing subsets of datasets via range queries and user-defined aggregations and transformations) for very large datasets in archival storage systems, in a shared Distributed Computing environment.

features:

- subsetting of very large datasets through multi-dimensional range queries
- application specific non-spatial subsetting

application:

Virtual Microscope [3]

related work:

▶*ADR*, ▶*dQUOB*, ▶*SRB*

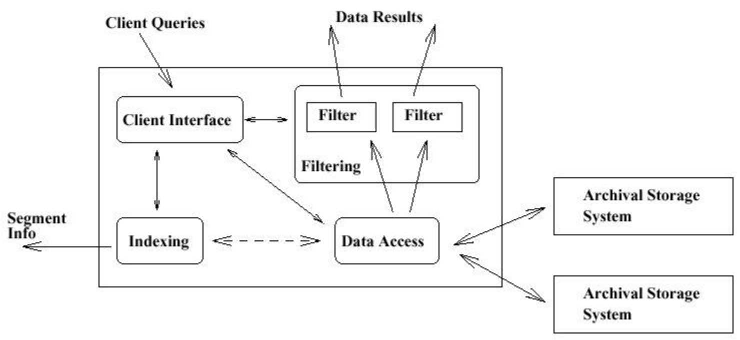


Figure 4.13: DataCutter: System Architecture

citation:

[59], [60], [299]

details:

DataCutter was developed following the *filter-stream programming model* [60]. This model represents the processing units of a data-intensive application as a set of *filters*, which are designed to be efficient in their use of memory and scratch space. In the filter-stream programming model, an application is represented by a collection of *filters*. A filter is a portion of the full application that performs some amount of work. Communication with other filters is solely through the use of *streams*. A stream is a communication abstraction that allows fixed sized untyped data buffers to be transported from one filter to another.

Architecture The *DataCutter* infrastructure consists of two major components: *Proxies* and *DataCutters*. A *Proxy* provides support for caching and management of data near a set of clients. The goal is to reduce the response time seen by a client, decrease the amount of redundant data transferred across the wide-area network, and improve the scalability of data servers.

The application processing structure of *DataCutter* is decomposed into a set of processes, called *filters*. They can execute anywhere but are intended to run on a machine close to the archival storage server or within a *Proxy*.

The architecture of *DataCutter* (Figure 4.13) is being developed as a set of modular services. The client interface service interacts with the clients an receives queries from them. The data service provides low-level I/O support for accessing the datasets stored on archival storage systems. Both the filtering and the indexing services use the data access service to read data and index information from files stored on archival storage systems. The indexing service manages the indices and indexing methods registered with *DataCutter*. The filtering service manages the filters for application-specific aggregation operations.

Indexing A *DataCutter* supported dataset consists of a set of data files and a set of index files. Data files contain the data elements of a dataset. Each data file is viewed as consisting of a set of *segments*, which are the units of retrieval from archival storage for spatial range queries. Because storing very large datasets results in a large set of data files, each of which may itself be very large, it may be expensive to manage the index and to perform a search to find segments using a single index file. To alleviate this problem, *DataCutter* uses a multi-level hierarchical indexing scheme implemented via *summary index files* and *detailed index files*.

Filters Filters are used to perform non-spatial subsetting and data aggregation. A filter is a specialized user program that preprocesses data segments retrieved from archival storage before returning them of the requesting client. They can be used for elimination of unnecessary data near the data source, preprocessing of datasets in a pipelined fashion before sending them to the clients, and data aggregation. Filters execute in a restricted environment and can execute anywhere, but are intended to run on a machine close to the archival storage server or within a *Proxy*.

Filters consist of an initialization function, a processing function, and a finalization function. A filter may optionally contain scratch space but cannot dynamically allocate and deallocate space, which allows the filtering service to better perform scheduling and enable execution in environments with limited resources.

DDS (Distributed Data Structure)

type:

Data Format

contact:

Steven D. Gribble, `gribble@cs.washington.edu`

description:

A *DDS* is a scalable, distributed data structure for Internet service construction not to be confound with ▶*SDDS*.

motivation:

DDS, is designed to simplify cluster-based Internet service construction.

features:

- incremental scaling of throughput and data capacity
- fault tolerance and high-availability
- high concurrency
- consistency
- durability

application:

Sanctio is an instant messaging gateway [161, Chapter 6],
Web server [161, chapter 6]

related work:

▶*LH*$*_{RS}$, ▶*SDDS*

citation:

[161]

details:

The architecture consists of the following components:

Client A client consists of service-specific software running on a client machine that communicates across the wide-area with one of many service instances running in the cluster.

Service A service is a set of cooperating software processes, each of which are called a service instance.

Hash Table API The Hash Table API is the boundary between a service instance and its *DDS library*. The API provides services with `put()`, `get()`, `remove()`, `create()`, and destroy operations on hash tables.

DDS Library The *DDS Library* is a Java class library that presents the hash table API to services. The library accepts hash table operations, and cooperates with the *Bricks* to realize those operations.

Brick Bricks are the only system components that manage durable data. Each Brick manages a set of network-accessible single node hash tables. A Brick consists of a buffer cache, a lock manager, a persistent chained hash table implementation, and network stubs and skeletons for remote communication.

All service instances in the cluster see the same consistent image of the *DDS*. As a result, any WAN client can communicate with any service instance.

A distributed hash table provides incremental scalability of throughput and data capacity as more nodes are added to the cluster. To achieve this, partition tables are horizontally partitioned to spread operations and data across bricks. Each brick thus stores some number of partitions of each table in the system, and when new nodes are added to the cluster, this partitioning is altered so that data is spread onto the new node.

To find the partition that manages a particular hash table key, and to determine the list of replicas in partitions' replica groups, the *DDS* libraries consult two *Metadata Maps* that are replicated on each node of the cluster. The first map is called the *Data Partitioning (DP) map*. Given a hash table key, the DP map returns the name of the key's partition. The second map is called the *Replica Group (RG) membership map*. Given a partition name, the RG map returns a list of bricks that are currently serving as replicas in the partition's replica group.

Dfs (Distributed File System)

type:

File System

url:

`http://www.microsoft.com/ntserver/nts/downloads/winfeatures/NTSDistrFile/AdminGuide.asp`

description:

The Microsoft *Distributed File System (Dfs)* is a network server component. It is a means for uniting files on different computers into a single name space. *Dfs* makes it easy to build a single, hierarchical view of multiple file servers and file server shares on the network.

motivation:

Historically, with the *Universal Naming Convention (UNC)*, a user or application was required to specify the physical server and share in order to access file information. As networks continue to grow in size mapping a single drive letter to individual shares scales poorly. *Dfs* solves these problems by permitting the linking of servers and shares into a simpler, more meaningful name space.

features:

- custom hierarchical view of shared network resources: by linking shares together, administrators can create a single hierarchical volume that behaves as though it were one giant hard drive. Individual users can create their own *Dfs* volumes.
- flexible volume administration: individual shares participating in the *Dfs* volume can be taken offline without affecting the remaining portion of the volume name space.
- higher data availability and load balancing: multiple copies of read only shares can be mounted under the same logical *Dfs* name to provide alternate locations for accessing data and permitting limited load balancing between drives or servers.
- name transparency: users navigate the logical name space without consideration to the physical locations of their data.

related work:

NFS [302]

citation:

[233]

details:

A *Dfs Root* is a local share that serves as the starting point and host to other shares. Any shared resource can be published into the *Dfs Name Space*. A *Dfs Volume* is accessed using a standard UNC name:

```
\\Server_Name\Dfs_Share_Name\Path\File
```

where `Server_Name` is the name of the host *Dfs* computer name, `Dfs_Share_Name` maps to any share that is designated to be the root of the *Dfs*, and `Path\File` is any

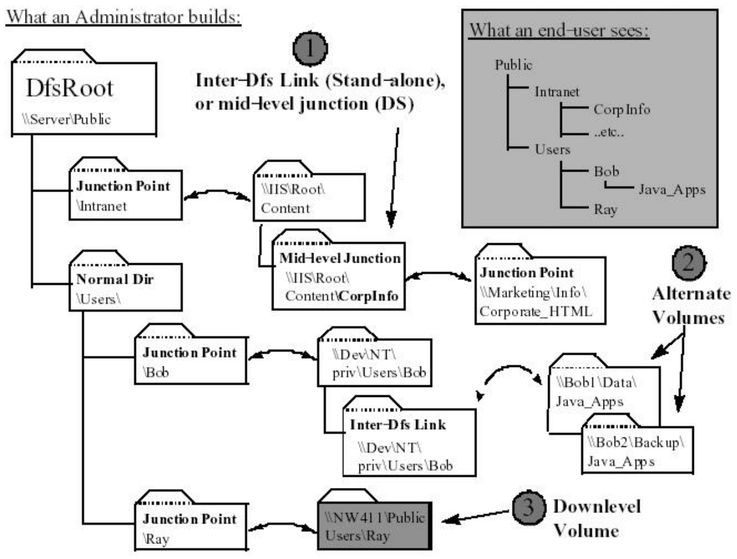


Figure 4.14: DFS: Typical Scenario

valid Win32 path name. Shares participating in a *Dfs* may be hosted by any server accessible by clients of the *Dfs*. This includes shares from the local host, shares on any Windows NT server or workstation, or shares accessible to Windows NT through client software.

Figure 4.14 shows a typical *Dfs* scenario. *Inter-Dfs Links* join one *Dfs* volume to another. *Midlevel Junctions* are a planned feature to support unlimited hierarchical junctioning that does not require Inter-Dfs Links.

The *Partition Knowledge Table (PKT)* holds knowledge of all junction points. It maps the Logical *Dfs* name space into physical referrals. When the *Dfs* Client attempts to navigate a junction, it first looks to its locally cached PKT entries. If the referral cannot be resolved, the client contacts the *Dfs* Root. If the referral still cannot be resolved, an error occurs. If the referral is properly resolved, the client adds the referral to its local table of entries. When a *Dfs* client obtains a referral from the PKT, that referral is cached for five minutes. If the client reuses that referral, the time to live is renewed. Otherwise, the cache expires. If alternate volumes exist for a particular referral, all alternates are handed to and cached by the client. The client randomly selects which referral to use.

DFSA (Direct File Access System) see ***MOSIX***

DGRA (Data Grid Reference Architecture)

type:

Standardization Effort

contact:

Ian Foster, `foster@mcs.anl.gov`

description:

The *DGRA* defines interoperability mechanisms rather than a complete vertically integrated solution.

motivation:

- to establish a common vocabulary for use when discussing Data Grid systems
- to document what is seen as the key requirements for Data Grid systems
- to propose a specific approach to meeting these requirements

related work:

▷*Globus*, ▷*GGF*, ▶*OGSA*

citation:

[132], [134], [137]

details:

Figure 4.15 illustrates some of the principal elements in a Data Grid architecture. *Data Catalogs* maintain information about data (meta-data) that is being manipulated within the Grid, the transformations required to generate derived data, and the physical location of that data. Resources can be storage systems, computers, networks, and code repositories. Application-specific *Request Formulation Tools* enable the end user to define data requests, translating from domain-specific forms to standard request formats, perhaps consulting application-specific ontologies. *Request Planning* and *Request Execution* is code that implements the logic required to transform user requests for virtual data elements into the appropriate catalog, data access and computational operations, and to control their execution. A clean separation of concerns exists between virtual *Data Catalogs* and the control elements that operate on that data *(Request Manager)*.

Figure 4.16 illustrates the primary functional elements, placing them in the context of a conventional Grid architecture to make clear how they relate to each other and to other Grid services. Shading indicates some of the elements that must be developed specifically to support Data Grids.

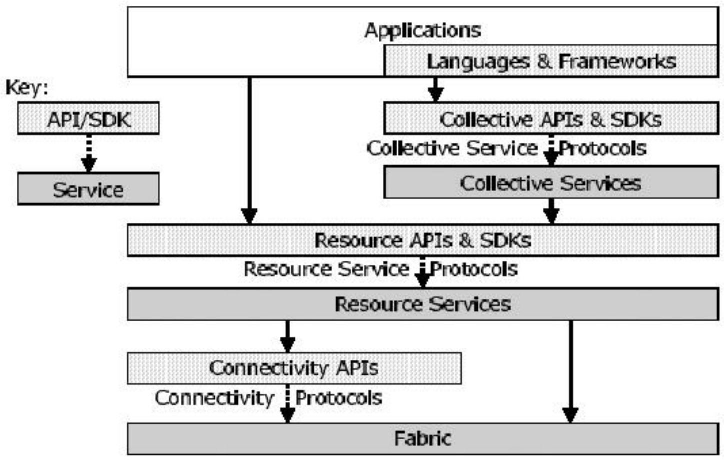


Figure 4.15: DGRA: Principal Data Grid Components

Fabric This lowest level of the Grid architecture comprises the basic resources from which a Data Grid is constructed. Some of these elements – eg. storage systems – must be extended to support Data Grid operation. In addition, new fabric elements are required, such as catalogs and code repositories for virtual data transformation programs.

Connectivity Services at this level are concerned with communication and authentication.

Resource Services at this level are concerned with providing secure remote access to storage, computing, and other resources.

Collective Services at this level support the coordinated management of multiple resources. Significant new development is required: in particular, catalog services, replica management services (►*EDG Replica Manager*, ►*Globus Replication Management Architecture*), community policy services, coherency control mechanisms for replicated data, request formulation and management functions for defining, planning and executing virtual data requests and replica selection mechanisms.

Application This level represents the discipline-specific Data Grid applications.

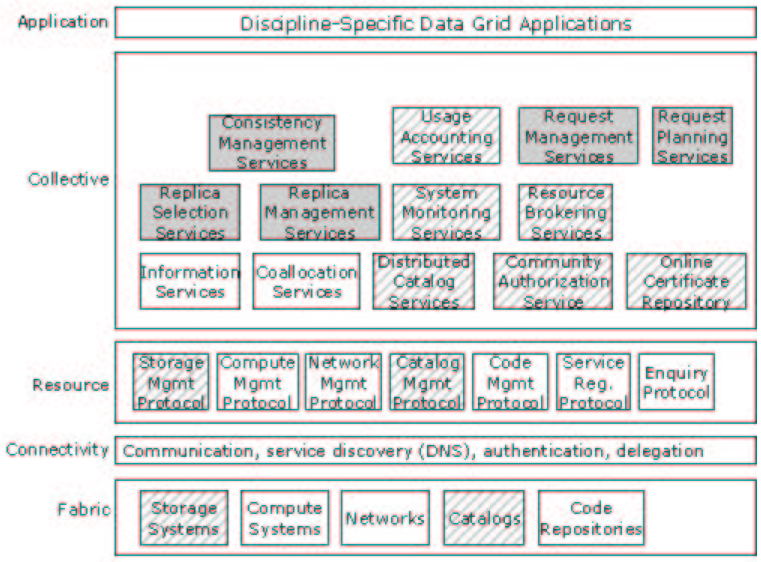

Application
Discipline-Specific Data Grid Applications
Collective
Consistency Management Services
Usage Accounting Services
Request Management Services
Request Planning Services
Replica Selection Services
Replica Management Services
System Monitoring Services
Resource Brokering Services
Information Services
Coallocation Services
Distributed Catalog Services
Community Authorization Service
Online Certificate Repository
Resource
Storage Mgmt Protocol
Compute Mgmt Protocol
Network Mgmt Protocol
Catalog Mgmt Protocol
Code Mgmt Protocol
Service Reg. Protocol
Enquiry Protocol
Connectivity
Communication, service discovery (DNS), authentication, delegation
Fabric
Storage Systems
Compute Systems
Networks
Catalogs
Code Repositories

Figure 4.16: DGRA: Major Components

DIOM (Distributed I/O Management)

type:

Intelligent I/O System

contact:

Martin Schulz, `schulzm@in.tum.de`

url:

`http://www.csl.cornell.edu/~schulz/projects.html`

description:

DIOM is a data management infrastructure allowing for a consistent and transparent handling of distributed data.

motivation:

DIOM enables data-intensive applications to take full advantage of the I/O potential in cluster environments by efficiently exploiting local resources present in most current cluster architectures.

application:

PET Image Reconstruction [309, 4.]

related work:

Galley [252], PIOUS [240], ►*PPFS II*, ►*PVFS2*

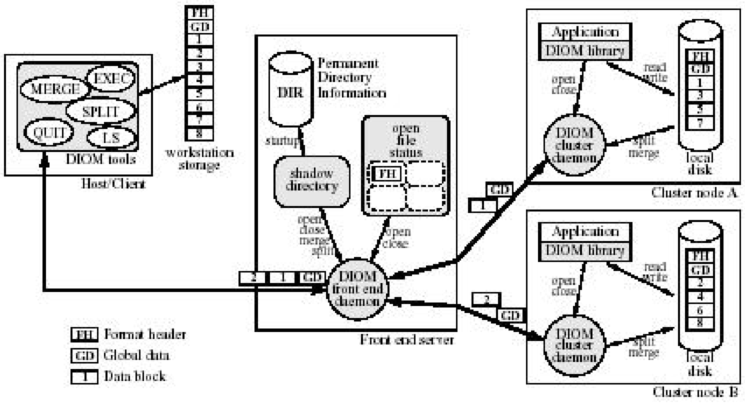


Figure 4.17: DIOM: General Architecture

citation:

[309], [310]

details:

Figure 4.17 shows the three main tiers of the overall system – *DIOM Tools,* a cluster entrance component located on a *Front-End Server,* and the software on the individual *Cluster Nodes* themselves supporting the local resource access. The most important user tools are the ones that control the data transfer to and from the cluster (`split, merge`). Any tool accesses the *DIOM* services through a *DIOM* daemon running on the cluster Front-End server. This daemon maintains a global directory of all *DIOM* controlled data and also executes the actual work during the `split` and `merge` operations. These operations are guided by additional semantic information about both the data and the target application. On the cluster side the front-end daemon communicates with the node local daemons, which are responsible for the actual storage operations on the individual nodes. They also include support for the maintenance of several storage directories on potentially different disks. Applications written for the *DIOM* system can access the distributed data with the help of a separate library *DIOMLIB,* which offers the data in a highly structured manner based on the data format contained within a self-describing file format.

One major challenge in this concept is to distribute the data among Cluster Nodes in a manner useful for the I/O pattern of the applications, as this requires semantic information about both data and application. Therefore, two separate descriptions are introduced: one describing the data layout and one describing an application specific splitting pattern. During the transfer of data to the cluster both definitions are then combined resulting in a data dependent and application specific distribution.

Direct I/O

type:

Application Level Method

description:

Direct I/O is I/O to files which bypasses the OS's buffer cache. It allows a file system to copy data directly to and from a disk and a user buffer. This eliminates a memory copy, as normal buffered I/O reads data off the disk into the buffer cache and then copies it into the user buffer. *Direct I/O* can offer significant speed improvements in accesses to large files, saves memory and boosts the performance of applications that cache their own data independently.

related work:

►*Active Buffering*

citation:

[114]

Discretionary Caching

type:

Application Level Method

contact:

Murali Vilayannur, `vilayann@cse.psu.edu`

description:

Discretionary Caching is a selective caching method for clusters based on ►*PVFS1*.

motivation:

While caching is useful in some situations, it can hurt performance if one is not careful about what to cache and when to bypass the cache. The introduced system addresses this problem.

features:

- builds on the architecture of ►*PVFS1*
- compiler-directed techniques for identifying what data should be brought into the cache
- run-time techniques for dynamic bypassing decisions

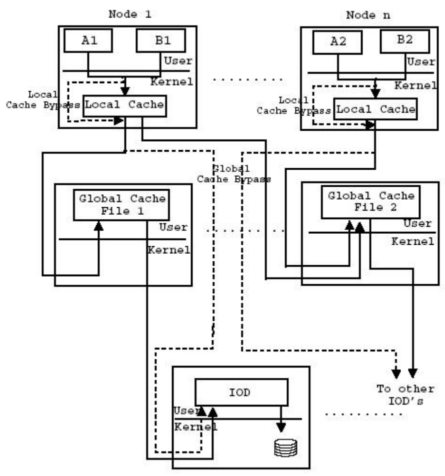


Figure 4.18: Discretionary Caching: System Architecture

related work:

▶*PPFS II*, ▶*PVFS1*

citation:

[378]

details:

The system provides two levels of caching, a *Local Cache* at every node of the cluster where an application process executes, and a *Global Cache* that is shared by different nodes (and possibly different) applications across the cluster. The Local Cache is implemented within the Linux kernel (a dynamically-loadable module) and can be shared across all the processes running on that node. Each Global Cache in the system is a user-level process serving requests to a specific file running on a cluster node, to which explicit requests are sent by the Local Caches, and is shared by different applications.

Figure 4.18 shows the system architecture. Nodes 1..n are the clients where one or more application processes run, and have a Local Cache present. Upon a miss, requests are either directed to the Global Cache (one such entity for a file), or are sent directly to the data server daemon *IOD* node(s) containing the data in the disk(s).

The original ▶*PVFS1* library on the client aggregates the requests to a particular IOD, before making a socket request (kernel call) to the node running that IOD. The Local Cache intercepts this call and if the entire request can be satisfied without a network message, then the data is returned to the ▶*PVFS1* library and the application proceeds. Otherwise, a subsequent message is sent to the Global Cache with this request. If it can satisfy the request completely from its memory, it returns the data to the requesting Local Cache. Otherwise, it sends a request message to each of the IODs holding corresponding blocks, stores the blocks in its memory when it gets responses from the IODs, and then returns the necessary data to the requesting Local Cache.

As it is very important to be very careful in deciding what data to place in these caches and when to avoid/bypass them, the system provisions mechanisms for bypassing the Local and/or Global Caches for a read or write on a segment level. For deciding what to cache there exist two *compiler-based* strategies and a *run-time* technique.

Disk-Directed I/O

type:

Device Level Method

contact:

David Kotz, `dfk@cs.dartmouth.edu`

description:

Disk-Directed I/O is one approach to ►*Collective I/O*. It is a technique for optimizing data transfer given a high-level, ►*Collective I/O* interface. In this scheme, the complete high-level, collective request is passed to the I/O processors, which examine the request, make a list of disk blocks to be transferred, sort the list, and then use double-buffering and special remote-memory `get` and `put` messages to pipeline the transfer of data between compute-processor memories and the disks. Compared to a traditional system with caches at the I/O processors, this strategy optimizes the disk accesses, uses less memory (no cache at the I/O processors), and has less CPU and message-passing overhead.

related work:

►*Collective I/O*

citation:

[198], [199]

DPFS (Distributed Parallel File System)

type:

File System

contact:

Alok Choudhary, `choudhar@ece.nwu.edu`

description:

DPFS collects locally distributed unused storage resources as a supplement to the internal storage of parallel computing systems to satisfy the storage capacity requirement of large-scale applications. In addition, like parallel file systems, *DPFS* provides striping mechanisms that divide a file into small pieces and distributes them across multiple storage devices for parallel data access. The unique feature of *DPFS* is that it provides three different striping methods.

motivation:

One of the challenges brought by large-scale scientific applications is how to avoid remote storage access by collectively using enough local storage resources to hold huge amount of data generated by the simulation while providing high-performance I/O. *DPFS* is designed and implemented to address this problem.

features:

- *DPFS*-API provides a hint structure for choosing a suitable striping method
- adopts ►*MPI-I/O* derived data type approach

related work:

►*CXFS*, ►*GFS*

citation:

[323]

details:

DPFS adopts a client-server architecture: the client (compute node) sends requests to the server (I/O node) whenever it needs to perform input or output. The server which resides on a specific storage device is responsible for sending the requested data to the client or storing data from the client on local storage.

Figure 4.19 shows the layered architecture of *DPFS*. At the top is the parallel computing resource, which could be a distributed memory system such as IBM SP2, a NOW or shared memory systems such as SGI Origin 2000.

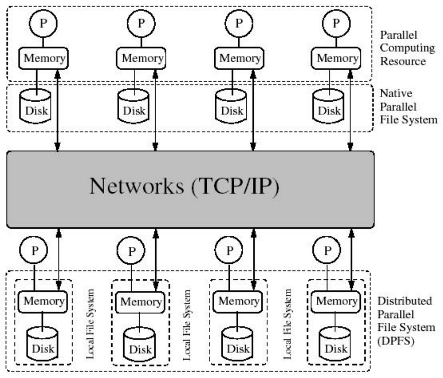

P
Memory
Disk
Parallel Computing Resource
Native Parallel File System
Networks (TCP/IP)
Local File System
Distributed Parallel File System (DPFS)

Figure 4.19: DPFS: System Architecture

Under the computing resources is the storage subsystem for parallel systems. This layer makes use of the local disk storage associated with the computing resource and forms a native parallel file system to achieve high performance.

At the bottom is the *DPFS*). It utilizes unused storage resources distributed over networks. These resources can be found on various commodity workstations and personal computers. Since the main storage space of a user is located at a NFS [302], much of the local disk space of these machines is unused. Aggregating these disjointed storage by *DPFS*, it is possible to have a very large storage space to satisfy the storage requirements of large-scale data-intensive applications.

As a repository for *DPFS* meta-data a database was chosen. Using a database solution has many advantages. It can save programming efforts since SQL is a very high-level and reliable interface compared to manipulating low level file directly. Moreover, the transaction mechanism provided by database systems can help maintain meta-data consistency easily, especially in a distributed environment. The *DPFS* meta-data keeps such information as what servers are available for I/O, what *DPFS* directories and files are currently maintained by *DPFS* and so on.

Like traditional Unix file systems, *DPFS* also provides a user interface which provides users with commands that can help manage files and directories in the file system. These commands include `cp`, `mkdir`, `rm`, `ls`, `pwd` and so on. *DPFS* also allows data transfer between sequential files and *DPFS*.

DPSS (Distributed-Parallel Storage System)

type:

Data Access System

contact:

Brian L. Tierney, `BLTierney@lbl.gov`

url:

`http://www-itg.lbl.gov/DPSS`

description:

DPSS is a network striped disk array allowing client applications complete freedom to determine optimal data layout, replication and/or coding redundancy strategy, security policy, and dynamic reconfiguration.

motivation:

Architecture and implementation of *DPSS* are intended to provide for easy and low-cost scalability.

features:

- provides high-speed parallel access to remote data
- similar to a striped RAID [83] system, but tuned for WAN access
- data is striped across both disks and servers
- on a high-speed network, can actually access remote data faster than from a local disk
- data replication
  - load balancing across servers
  - fault tolerance
- automatic TCP buffer tuning

application:

Kaiser: medical imaging [189], Terravision [344], Wide-Area Large Data Object Architecture (WALDO) [393]

related work:

►*GridFTP*. ►*SRM* borrowed several *DPSS* ideas.

citation:

[361], [362], [363], [364], [365]

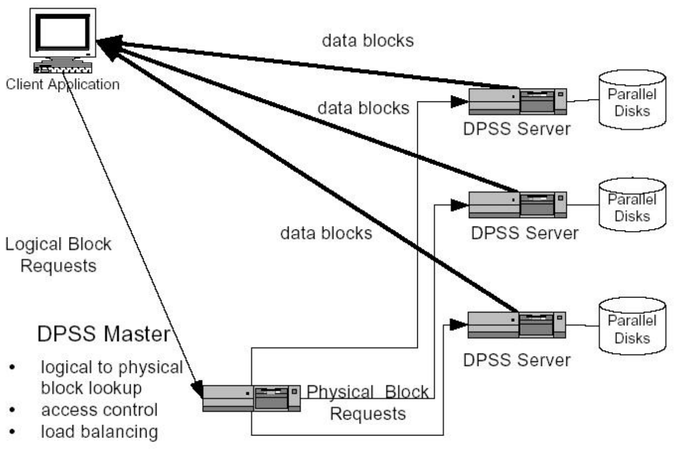


Figure 4.20: DPSS: Architecture

details:

The *DPSS* is a collection of disk servers which operate in parallel over a wide-area network to provide logical block level access to large datasets, as shown in Figure 4.20.

The *DPSS* is essentially a *block* server that can supply data to applications located anywhere in the network in which it is deployed. Multiple low-cost, medium-speed disk servers use the network to aggregate their data streams. Data blocks are declustered (dispersed in such a way that as many system elements as possible can operate simultaneously to satisfy a given request) across both disks and servers. This strategy allows a large collection of disks to seek in parallel, and all servers to send the resulting data to the application in parallel.

At the application level, the *DPSS* is a semi-persistent cache of named data-objects, and at the storage level it is a logical block server. The overall data flow involves *third-party* transfers from the storage servers directly to the data-consuming application. Thus the application requests data, these requests are translated to physical block addresses (server name, disk number, and disk block), and the servers deliver data directly to the application.

Operated primarily as a network-based cache, the architecture supports cooperation among independently owned resources to provide fast, large-scale, on-demand storage to support data handling, simulation, and computation in a wide-area high-speed network-based Internet environment.

dQUOB (dynamic QUery OBject)

type:

Filter

contact:

Beth Plale, `plale@cs.indiana.edu`

url:

`http://www.cs.indiana.edu/˜plale/projects/dQUOB/`

description:

dQUOB is a middleware system providing continuous evaluation of queries over time sequenced data. The system provides access to data in data streams through SQL queries.

motivation:

The *dQUOB* system addresses two key problems in large-scale data-streams:

- unmanaged data streams often over- or under-utilize resources
- it can be difficult to customize computations for certain streams and stream behaviours and the potential exists for duplication of processing when multiple computations are applied to a single data source

features:

- declarative query language
- query optimization for partial evaluations
- queries as compiled code
- dynamic and continuous query re-optimization

application:

Scientists coupled a parallel and distributed global atmospheric transport model [275] and a parallel chemical model. The primary client of the models is a visAD visualization [173].

related work:

►*ADR*, ►*DataCutter*, Continual Query System [215]. Run-time detection in data streams has been addressed in [4] and in [288].

citation:

[276], [277]

details:

The system architecture consists of a *client* and a *server*. The *client* is located close to the user and is used for creating queries. It is a compiler which accepts a variant of SQL queries, compiles the query into an intermediate form, optimizes the query and generates a Tcl [341] script version of the code that is moved to the server. Queries are portable entities that are embedded into the data streams at run-time, and managed remotely during application execution.

The *server* is a centralized, application-wide service and query repository that accepts queries from clients and deploys them at *Quoblets*. The Quoblet consists of an interpreter to decode the script, and the *dQUOB* library to dynamically create efficient representations of queries at run-time. The resulting queries are stored in compiled form. At startup the empty Quoblet contacts the server to receive the URL for the data repository and the query in script form. Once compiled by a resident Tcl interpreter the query starts receiving events from the data repository and executes the queries.

During run-time, a re-optimizer gathers statistical information and periodically triggers re-optimization to generate a more optimal version of a query, if eg. the data stream behaviour has significantly changed or a new query has initially been optimized based only on historical trace data.

DRA (Disk Resident Arrays)

type:

I/O Library

contact:

Jarek Nieplocha, `j_nieplocha@pnl.gov`

description:

DRA is a model and library that extends the distributed array library called *Global Arrays (GA)* to support I/O. This library allows parallel programs to use essentially the same mechanisms to manage the movement of data between any two adjacent levels in a hierarchical memory system.

motivation:

In *out-of-core* (►*EM (External Memory) Algorithms and Data Structures*) computations, disk storage is treated as another level in the memory hierarchy, below cache, local memory, and (in a parallel computer) remote memories. However, the tools used to manage this storage are typically quite different from those used to manage access to local and remote memory. This disparity complicates implementation of out-of-core algorithms and hinders portability.

features:

- extends the GA model to another level, namely secondary storage
- forms a stand-alone I/O package and is also part of a larger system called ChemIO [139]
- collective and asynchronous read and write operations

application:

ChemIO [139]

related work:

Global Arrays Library [251]

citation:

[140]

details:

The *DRA* model extends the *GA* model to another level in the storage hierarchy, namely, secondary storage. It introduces the concept of a *DRA*, a disk-based representation of an array, and provides functions for transferring blocks of data between GA and *Disk Resident Arrays*. Hence, it allows programmers to access data located on disk via a simple interface expressed in terms of arrays rather than files. The benefits of GA (in particular, the absence of complex index calculations and the use of optimized, blocked communication) can be extended to programs that operate on arrays that are too large to fit into memory.

The *DRA* library encapsulates the details of data layout, addressing and I/O transfer. *DRA* read and write operations can be applied both to entire arrays and to sections of arrays (disk and/or GA); in either case, they are collective and asynchronous.

The *DRA* library distinguishes between temporary and persistent *DRA*. Persistent *DRA* are retained after program termination and are then available to other programs; temporary arrays are not. Persistent *DRA* must be organized so as to permit access by programs executing on an arbitrary number of processors; temporary *DRA* do not need to be organized in this fashion, which can sometimes improve performance.

E

EDG Replica Manager

type:

Replication

contact:

Peter Kunszt, `peter.kunszt@cern.ch`

url:

`http://edg-wp2.web.cern.ch/edg-wp2/replication/index.html`

description:

EDG Replica Manager is a high-level system for replica management in Data Grids. It was developed within the context of the ▷*EDG* Project.

motivation:

Optimization of data access can be achieved via data replication, whereby identical copies of data are generated and stored at various globally distributed sites. However, dealing with replicas of files adds a number of issues not present if only a single file instance exists. Replicas must be kept consistent and up-to-date, their location must be stored in a catalog, their lifetime needs to be managed, etc. These and other issues necessitate a high-level system for replica management in Data Grids.

features:

- the *EDG Replica Manager* depends only on open source technologies
- adopts the web services paradigm [392] to be prepared to move to the ▶*OGSA* standard
- provides replica optimization

related work:

▶*SRB*, ▶*Giggle*, ▶*GDMP*, ▶*Globus Replication Management Architecture*

citation:

[169], [201], [403], [404], [405], [406], [407]

details:

Figure 4.21 presents the components of the *EDG Replication Manager* from the user's point of view. It is a logical single entry point for the user to the replica man-

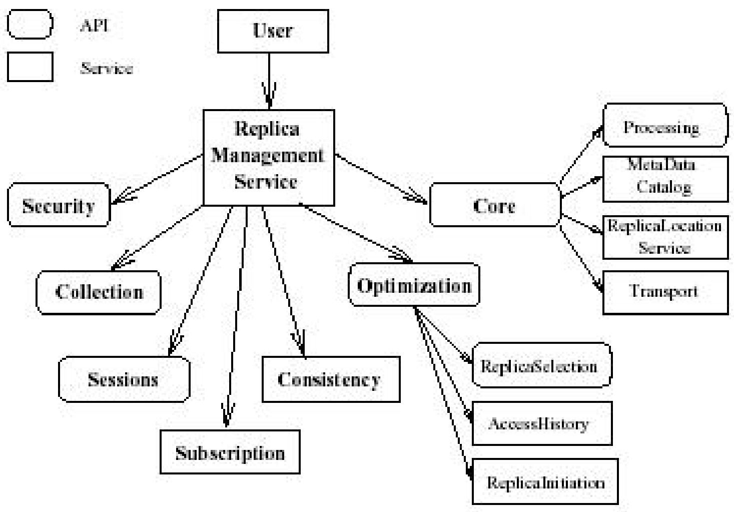


Figure 4.21: EDG Replication Manager: Main Components

agement system. It encapsulates the underlying systems and services and provides a uniform interface to the user.

It consists of the following components:

Core The Core module coordinates the main functionality of replica management, which is replica creation, deletion, and cataloging by interacting with third party modules. To fulfill this function the core package requires access to the following services:

Transport The Transport service provides the functionalities required for file transfer and may be implemented by means of different techniques, such as ▶*GridFTP*.

Processing The Processing API allows the incorporation of pre- and post-processing steps before and after a file transfer respectively.

Replica Metadata Catalog (RMC) The Replica Metadata Catalog stores mappings between user defined *Logical Filename (LFN)* aliases and the globally unique *Grid Unique Identifiers (GUID)*.

Replica Location Service (RLS) This service maintains and provides access to information about replicas of data. It consists of two components:

- The *Local Replica Catalog (LRC)* is a system that maintains independent and consistent information about replicas at a single site.

- The *Replica Location Index (RLI)* provides the means to locate in which LRC the information about a given GUID is held.

This service has been designed together with the ▷*Globus* Project. See also ►*Giggle*.

Replica Optimization Service (ROS) This service provides information about access costs for replicas based on network costs. This information is used for selecting the best replicas.

Consistency Service This service takes care of keeping the set of replicas of a file consistent as well as the meta information stored in various catalogs.

Subscription Service The Subscription Service manages subscription-based replication where data appearing at a data source is automatically replicated to subscribed sites.

Session Management The Session Management component provides generic checkpointing, restart, and rollback mechanisms to add fault tolerance to the system.

Collections are defined as sets of logical filenames and other collections.

Security Module The Security Module manages the required user authentication and authorization, in particular, issues pertaining to whether a user is allowed to create, delete, read, and write a file.

The *EDG Replica Manager* offers an API and a command line interface. Among many more, the command line interface offers the following commands:

- `copyAndRegisterFile` for putting a local file into the Grid storage and register it in the catalog
- `registerFile` for registering a file in the catalog
- `unregisterFile` for unregistering a file from the catalog
- `replicateFile` for replicating an existing file to a certain Grid storage and updating catalog
- `deleteFile` for deleting a file from storage and remove entry from catalog
- `getBestFile` for replicating a file to the nearest storage element in the cheapest way

Efficient I/O Operations

type:

Definition

contact:

Erich Schikuta, `erich.schikuta@univie.ac.at`

description:

Efficient I/O Operations can be summarized by the following characteristic goals:

- maximize the use of available Parallel I/O devices to increase the bandwidth
- minimize the number of disk read and write operations per device
- minimize the number of I/O specific messages between processes to avoid unnecessary costly communication
- maximize the hit ratio to avoid unnecessary data accesses

related work:

►*EM Algorithms and Data Structures,* ►*PDM,* ►*TPIE*

citation:

[306]

EM (External Memory) Algorithms and Data Structures

type:

General Method

contact:

Jeffrey Scott Vitter, `jsv@purdue.edu`

description:

Algorithms and data structures that explicitly manage data placement and movement as *EM algorithms and data structures*. The terms *I/O algorithms* or *out-of-core algorithms* are used likewise.

motivation:

EM algorithms and data structures exploit locality in order to reduce the I/O costs.

application:

►*TPIE*

related work:

▶*PDM*, ▶*TPIE*

citation:

[27], [380]

details:

A variety of *EM* paradigms exist for solving *batched* and *online problems* in *EM* to exploit locality efficiently. No preprocessing is done and the entire file of data items must be processed in batched problems, whereas in online problems the computation is done in response to a continuous series of query operations. The data being queried can be either *static*, which can be preprocessed for efficient query processing, or *dynamic*, where the queries are intermixed with updates such as insertions and deletions. The ▶*PDM* provides an elegant and reasonably accurate model for analyzing the relative performance of *EM Algorithms and Data Structures*. The I/O performance of many algorithms and data structures can be expressed in terms of the ▶*I/O bounds* for fundamental operations.

The most fundamental *EM Data Structure* is the *B-tree* [196], which corresponds to an internal memory balanced search tree.

Enstore

type:

Mass Storage System

contact:

Jon Bakken, `bakken@fnal.gov`

url:

`http://hppc.fnal.gov/enstore/`

description:

Enstore provides a generic interface to efficiently use MSSs as easily as if they were native file systems.

motivation:

Enstore is designed to provide high fault-tolerance and availability sufficient for Fermilab's Run II [298] data acquisition needs, as well as easy administration and monitoring.

features:

- supports both automated and manual storage media libraries
- no upper hard limit on file size, no upper limit on the number of files that can be stored on a single volume
- users can search and list the contents of media volumes as easily as native file systems
- uses a client-server architecture which provides a generic interface for users
- optimized access to large (Petabyte) datasets made up of many (100s of millions of) files of 1-2 GB in size.

application:

▷*PPDG*

related work:

►*HPSS*, ►*SAM*

citation:

[40]

details:

Enstore uses a client-server architecture to provide a generic interface for users to efficiently use MSSs. The system architecture does not dictate an exact hardware architecture. Rather, it specifies a set of general and generic networked hardware and software components.

The system is written in *Python* [280], a scripting language that has advanced object-oriented features.

Enstore has four major kinds of software components:

Pnfs Name Space The pnfs package implements an NFS [302] and mount daemon, which do not serve a file system, but, instead make a collection of database entries look like a file system. This name space is used for administration interactions, configuration information and for user file information.

encp A program used to copy files to and from media libraries. It has a syntax similar to the `cp` command in Unix.

Administration Tools

Servers

Volume Clerk keeps and administers volume information

File Clerk tracks files in the system

Library Manager is a server which queues up and dispatches work for a virtual library

Mover is a task bound to a single drive, and seeks to use that drive to service read and write requests; it is responsible for efficient data movement

Configuration Server maintains and distributes all information about system configuration

Media Changer mounts an dismounts the media into and from the drive according to a request from the Mover

Inquisitor obtains information from the *Enstore* system and creates reports about the system status

Alarm Server maintains a record of alarms raised by other servers.

Expand (Expandable Parallel File System)

type:

File System

contact:

Félix García, `fgarcia@arcos.inf.uc3m.es`

url:

`http://www.arcos.inf.uc3m.es/˜xpn`

description:

Expand is a parallel file system based on NFS [302] servers. It allows the transparent use of multiple NFS servers as a single file system, providing a single name space. The different NFS servers are combined to create a distributed partition where files are striped.

motivation:

Current parallel file systems and Parallel I/O libraries lack generality and flexibility for general purpose distributed environments, because these systems do not use standard servers. The main motivation of *Expand* is to build a parallel file system for heterogeneous general purpose distributed environments with fault tolerance features.

features:

- provides POSIX and ▶*MPI-IO* interface

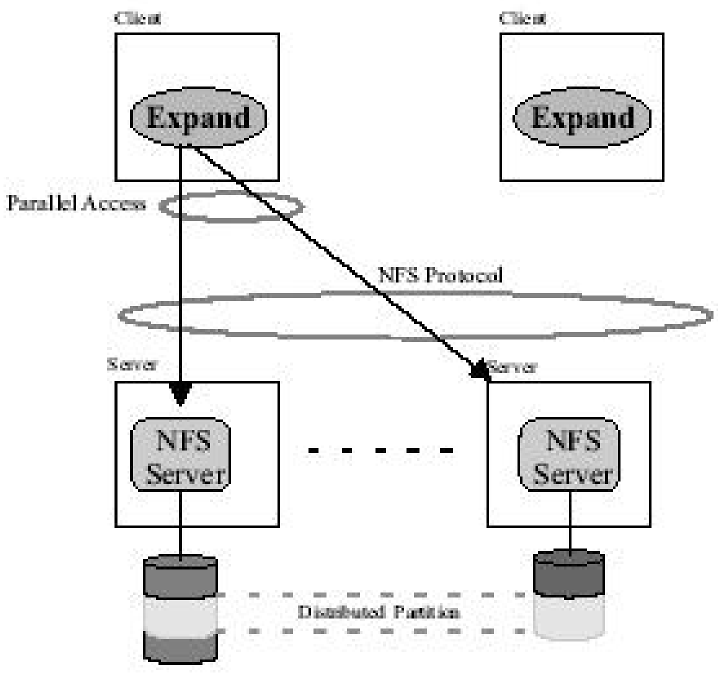


Figure 4.22: Expand: Architecture

- independent of the OS used at the server
- no changes to the NFS servers required
- independent of the OS used at the client
- parallel file system construction is greatly simplified, because all operations are implemented on the clients
- allows parallel access to both data of different files and data of the same file

related work:

Bigfoot-NFS [194], ►*Slice*

citation:

[147], [148]

details:

Figure 4.22 shows the architecture of *Expand*. *Expand* provides high-performance I/O exploiting parallel access to files striped among several NFS servers. *Expand* is designed as a client-server system with multiple NFS servers, with each file striped across all NFS servers. All operations on clients are based on the NFS protocol. Processes on the clients use the *Expand* Library to access the files.

A file in *Expand* consists of several subfiles, one for each NFS partition. All subfiles are fully transparent to the users. *Expand* hides these subfiles, offering to the clients a traditional view of the files. Each subfile of an *Expand* file has a small header at the beginning of the subfile. This header stores the subfile's meta-data.

To simplify the naming process and reduce potential bottlenecks, *Expand* does not use a meta-data manager. The meta-data of a file resides in the header of a subfile stored on a NFS Server, called the *Master Node*. To obtain the Master Node of a file, the file name is hashed into the number of the node.

Fault tolerance is provided using the RAID [83] concept applied to files. *Expand* transparently inserts parity blocks following a RAID5 (Block-Interleaved Distributed-Parity) pattern. This solution can tolerate a failure in an I/O node or an NFS server. Fault tolerant files have their meta-data replicated to several subfiles.

To enhance I/O, user requests are split by the *Expand* library into parallel sub-requests sent to the involved NFS servers. When a request involves *k* NFS servers, *Expand* issues *k* requests in parallel to the NFS servers, using threads to parallelize the operations. The same criteria is used in all *Expand* operations. A parallel operation to *k* servers is divided into *k* individual operations that use RPC and the NFS protocol to access the corresponding sub files.

F

Frangipani

type:

File System

contact:

Chandramohan A. Thekkath, `thekkath@acm.org`

url:

`http://research.compaq.com/SRC/personal/thekkath/frangipani/home.html`

description:

Frangipani is a scalable distributed file system that manages a collection of disks on multiple machines as a single shared pool of storage.

motivation:

The ideal distributed file system would provide all its users with coherent, shared access to the same set of files, yet would be arbitrarily scalable to provide more storage space and higher performance to a growing user community. It would be highly available in spite of component failures. It would require minimal human administration and would not become more complex as more components were added. *Frangipani* is a file system that approximates this ideal.

features:

- very simple internal structure
- user programs get essentially the same semantic guarantees as on a local Unix file system
- lock service to coordinate access
- per-file lock granularity
- write-ahead redo logging of meta-data, user data is not logged
- data replication

related work:

►*CXFS*, ►*GPFS*, ►*Petal*

citation:

[359]

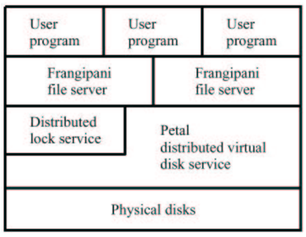


Figure 4.23: Frangipani: Layering

details:

Frangipani is layered on top of ▶*Petal*, an easy-to-administer distributed storage system, that provides virtual disks to its clients. ▶*Petal* optionally replicates data for high availability.

Figure 4.23 illustrates the layering in the *Frangipani* system. Multiple interchangeable *Frangipani Servers* provide access to the same files by running on top of a shared ▶*Petal Virtual Disk*, coordinating their actions with locks to ensure coherence. The file system layer can be scaled up by adding *Frangipani* Servers.

Frangipani uses write-ahead redo logging of meta-data to simplify failure recovery and improve performance; user data is not logged. If a *Frangipani* Server crashes, the system eventually detects the failure and a *Recovery Daemon* runs recovery on that server's log.

To coordinate access to the data and to keep the buffer caches coherent across the multiple servers *Frangipani* uses a *Lock Service*. The Lock Service is a general-purpose service that provides multiple-reader/single-writer locks to clients on the network. Its implementation is distributed for fault tolerance and scalable performance.

FTC (File Transfer Component)

type:

Data Transfer Protocol

contact:

Gregor von Laszewski, `gregor@mcs.anl.gov`

description:

FTC is a component for file transfers that separates issues related to requesting, performing and visualizing the actual file transfer.

motivation:

FTC was designed to provide a single interface to connect to various servers using different protocols.

features:

- authentication
- connects to different transfer servers
- uses ►*RFT*
- Java Client API to ►*GridFTP*
- support for FTP
- drag and drop interface

related work:

►*GridFTP*, ►*RFT*

citation:

[384]

details:

The architecture is based on a number of reusable components. It can do both client and server-side file transfers, however it is mostly used for client side.

Figure 4.24 shows the architecture that separates access, transfer, and display of data related operations cleanly. Hence, it comprises three main components: the *Access Manager*, the *Transfer Manager*, and the *GUI*.

The *File Access Providers* allow to access file systems based on API calls that are implemented by using appropriate protocols. These providers are integrated into *FTC* by using a single interface called *File Access Interface* which provides uniform access to sources supporting multiple file transfer protocols.

The *File Transfer Providers* are responsible for transferring the files. The *File Transfer Interface* provides uniform access to multiple File Transfer Providers.

FTP-Lite

type:

Data Transfer Protocol

contact:

`condor-admin@cs.wisc.edu`

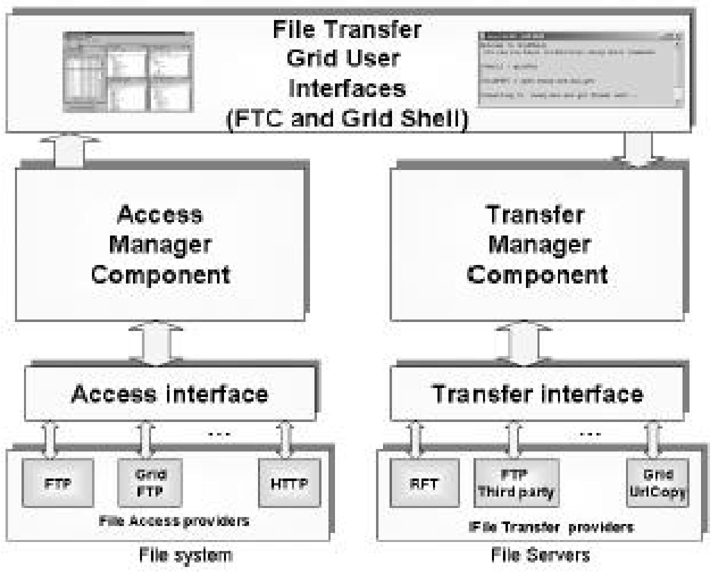


Figure 4.24: FTC: Architecture

url:

`http://www.cs.wisc.edu/condor/ftp_lite/`

description:

FTP-Lite is a simple implementation of ▶*GridFTP*. It provides a blocking, Unix-like interface to the protocol. The interface is designed to be easily used by single-threaded applications. In particular, *FTP-Lite* is used by ▶*Parrot* to attach FTP services to ordinary POSIX programs.

related work:

▷*Condor*, ▶*GridFTP*, ▶*Parrot*

G

GASS (Global Access to Secondary Storage)

type:

Data Access System

contact:

Ian Foster, `foster@mcs.anl.gov`

url:

`http://www.globus.org/gass/`

description:

GASS consists of libraries and utilities which simplify the porting and running of applications that use file I/O in the ▷*Globus* environment.

motivation:

The need of high-performance computations in *Computational Grids* to execute programs distant from their data leads to the following requirements that have to be met simultaneously:

- impose few requirements on the application programmer
- impose few requirements on the resource provider
- allow high-performance implementations and support application-oriented management of bandwidth.

application:

▷*Globus* Executable Management (GEM) [102],
GASS Command Line Tools (`globus-rcp`, `globusrun`) [58, page 7], SF-Express [232]

related work:

Automatic GASS [37], ▷*Globus*

citation:

[58]

details:

GASS is not a general-purpose distributed file system but supports four *I/O patterns* which are common in high-performance Grid applications with particularly simple access patterns and coherency requirements:

- read-only access to an entire file assumed to contain *constant* data
- shared write access to an individual file is not required, meaning a policy can be adopted that if multiple writers exist, the file written by the *last* writer produces the final value of the entire file
- append-only access to a file with output required in *real-time* at a remote location
- unrestricted read/write access to an entire file with no other concurrent accesses

These operations require no explicit coordination among accessing processes. Two multiprocess I/O structures are explicitly excluded: information exchange by concurrent reading and writing of a shared file and cooperative construction of a single output file via other than append operations.

GASS addresses bandwidth management issues by providing a *File Cache*. According to two default data movement strategies data is moved into and out of this cache when files are opened and closed:

- fetch and cache on first read open
- flush cache and transfer on last write close

To enable streaming output, a remote file, that is opened in append mode, is not placed in the cache but a communication stream is created to the remote location, and write operations to the file are translated into communication operations on that stream. *GASS* also provides mechanisms that allow programmers to refine these default data movement strategies. These mechanisms fall into two general classes: relatively high-level mechanisms concerned with presstaging data into the cache prior to program access and post-staging of data subsequent to program access and low-level mechanisms that can be used to implement alternative data movement strategies.

The **Automatic GASS** [37] module can connect an arbitrary program to the *GASS* system. It is an example application of Bypass [69], a software commonly used to hook old programs up to new storage systems.

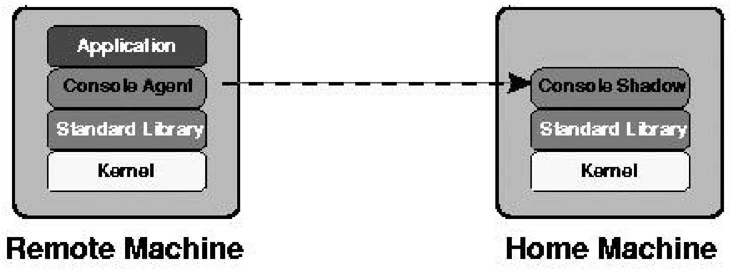


Figure 4.25: GC: System Overview

GC (Grid Console)

type:

Filter

contact:

`condor-admin@cs.wisc.edu`

url:

`http://www.cs.wisc.edu/condor/bypass/examples/grid-console/`

description:

GC is a system for getting mostly-continuous output from remote programs running on an unreliable network. *GC* is implemented using Bypass [69] and consists of two software components: an *Agent* and a *Shadow*. An Agent intercepts some of the I/O operations performed by an application running on a remote machine. When possible, it sends that output back to a Shadow process running on the home machine. Any number of remote processes may use a single Shadow to record their output. A *GC* system looks like Figure 4.25 when everything is working perfectly.

related work:

Bypass [69], ▷*Condor*, ►*dQUOB*

GDMP (Grid Data Mirroring Package)

type:

Replication

contact:

Mehnaz Hafeez, `Mehnaz.Hafeez@cern.ch`

url:

`http://cern.ch/GDMP`

description:

The *GDMP* client-server software system is a generic file replication tool that replicates files securely and efficiently from one site to another in a Data Grid environment using several ▷*Globus* Toolkit tools.

motivation:

Existing commercial database management systems provide replication features but they fail to satisfy the stringent requirements of consistency, security and high-speed transfers of huge amounts of data. An asynchronous replication mechanism that supports different levels of consistency, a uniform security policy and an efficient data transfer is necessary.

features:

- high-speed file transfers
- secure access to remote files
- selection and synchronization of replicas
- incorporation of site specific policies
- partial replication for limiting the amount of files to be replicated

application:

CMS [95]

related work:

►*Globus Replication Management Architecture,* ►*EDG Replica Manager,*

citation:

[170], [300], [334]

details:

In addition to replication *GDMP* manages replica catalog entries for file replicas and thus maintains a consistent view of names and locations of replicated files. All kinds of file formats are supported for file transfer and all files are assumed to be read-only.

GDMP is a multi-threaded client-server system that is based on the ▷*Globus* Toolkit. It consists of several modules that closely work together but are easily replaceable.

Control Communication Module This module takes care of the control of communication between clients and servers, and uses the ▷*Globus* I/O library as the middleware.

Data Mover Module This is the module which actually transfers files physically from one location to another one. It uses ►*GridFTP*.

Security Module This module provides methods to acquire credentials, initiate context establishment on the client side and accept context establishment requests on the server-side, and encrypting and decrypting messages and client authorization.

Request Manager Module This module makes it possible to generate requests on the client side and interpret these requests on the server side.

Database Manager Module This module interacts with the actual Database Management System. This is the only module which has to be swapped to use *GDMP* in a different system.

The *GDMP* replication process is based on the *producer-consumer model*. Each data production site publishes a set of newly created files (*Export Catalog*) to a set of consumer sites, which build a list of files that have not been transfered to the consumer sites (*Import Catalog*). The *GDMP Servers* deployed on each site ensure that the necessary data transfer operations complete successfully.

GDMP allows a partial-replication model where not all the available files are replicated. One or several filter criteria can be applied to the Export/Import Catalog in order to sieve out certain files. This allows for partial replication where both the producer and the consumer can limit the amount of files to be replicated.

GDS (Grid Data Services) see ***OGSA-DAI***
GDSF (Grid Data Service Factory) see ***OGSA-DAI***

Geo* (GIS on Massive Datasets)

type:
Other / Project

contact:
Jeffrey Scott Vitter, `jsv@purdue.edu`

url:
`http://www.cs.duke.edu/geo*/`

description:

The *Geo** project deals with systems and algorithms for massive data computing, focusing on geometric computing problems that are fundamental to GISs and other spatial data processing.

related work:

The following work is part of *Geo**:

- TerraFlow [343]: flow computation on massive grids
- ►*TPIE*: I/O computing toolkit
- ►*Slice*: scalable network storage
- ►*Trapeze*: high-speed network I/O

Gfarm (Grid Datafarm)

type:

Mass Storage System

contact:

`datafarm@apgrid.org`

url:

`http://datafarm.apgrid.org/`

description:

Gfarm provides a global parallel file system with online peta-scale storage, scalable I/O bandwidth, and scalable parallel processing, and it can exploit local I/O in a grid of clusters with tens of thousands of nodes.

motivation:

Gfarm specifically targets applications where data primarily consists of a set of records or objects which are analyzed independently.

features:

- Parallel I/O and parallel processing for fast data analysis
- world-wide group-oriented authentication and access control
- thousands-node, wide-area resource management and scheduling
- multi-tier data sharing and efficient access
- program sharing and management

- system monitoring and administration
- fault tolerance dynamic reconfiguration / automated data regeneration or re-computation

application:

HEP experiments at CERN [77]

related work:

▷*GriPhyN*

citation:

[339], [340]

details:

Large-scale data-intensive computing frequently involves a high degree of data access locality. To exploit this access locality, *Gfarm* schedules programs on nodes where the corresponding segments of data are stored to utilize local I/O scalability, rather than transferring the large-scale data to compute nodes.

As shown in Figure 4.26, *Gfarm* consists of the *Gfarm File System*, the *Gfarm Process Scheduler*, and *Gfarm Parallel I/O APIs*. The Gfarm File System is a parallel file system with *Gfarm File System Nodes* and *Gfarm Metadata Servers*. Each Gfarm File System Node acts as both an I/O node and a compute node with a large local disk on the Grid. A *Gfarm File* is a large-scale file that is divided into fragments and distributed across the disks of the Gfarm File System, and which will be accessed in parallel. File replicas are managed by the *Gfarm File System Metadata*. Metadata of the Gfarm File System is stored in the *Gfarm Metadata Database*. The *Gfarm File System Daemon*, called *gfsd*, runs on each Gfarm File System Node to facilitate remote file operations with access control in the Gfarm File System as well as user authentication, file replication, fast invocation, node resource status monitoring, and control.

To exploit the scalable local I/O bandwidth, the *Gfarm Process Scheduler* schedules *Gfarm File System Nodes* used by a given *Gfarm File* for affinity scheduling of process and storage.

Moreover, the *Gfarm Parallel I/O APIs* provide a local file view in which each processor operates on its own file fragment of the *Gfarm File*. The local file view is also used for newly created *Gfarm Files*.

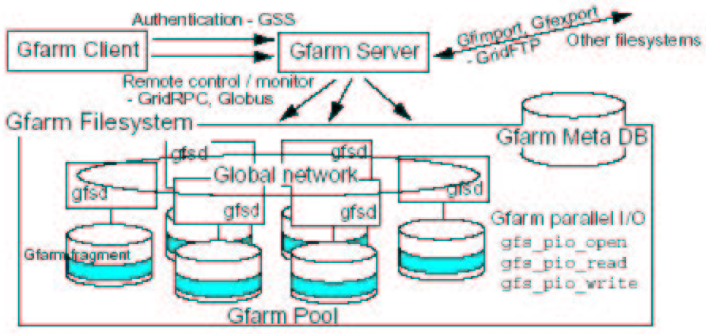


Figure 4.26: Gfarm: Software Architecture

A *Gfarm File* is partitioned into fragments and distributed across the disks on *Gfarm File System Nodes*. Fragments can be transferred and replicated in parallel by each *gfsd* using parallel streams.

The *Gfarm File System* supports file replicas which are transparently accessed by a *Gfarm URL* as long as at least one replicated fragment is available for each fragment. When there is no replica, *Gfarm* files are dynamically recovered by re-computation.

GFS (Global File System)

type:

File System

contact:

Kenneth W. Preslan, `kpreslan@sistina.com`

url:

`http://www.sistina.com/products_gfs.htm`

description:

GFS is a Linux cluster file system that allows multiple Linux machines to access and share disk and tape devices on a fibre channel or SCSI storage network. It performs well as a local file system, as a traditional network file system running over IP, and as a high-performance cluster file system running over storage networks like fibre channel.

motivation:

The goal was to develop a scalable (in number of clients and devices, capacity, connectivity, and bandwidth) server-less file system that integrates IP-based NAS and fibre channel-based SANs.

features:

- open source
- 64-bit files and file system
- bypasses buffer cache to increase performance
- symmetric scaling
- quotas control allocation of storage resources to applications or clients across entire cluster
- multi-journal for fast performance and quick recovery
- distributed meta-data: no architectural bottlenecks, no requirement for central meta-data server

application:

[327]

related work:

►*Clusterfile*, ►*CXFS*, ►*Frangipani*, ►*GPFS*

citation:

GFS-1: [330], GFS-2: [329], GFS-3: [279], GFS-4: [278]

details:

The first versions of *GFS* were developed at the University of Minnesota in the USA. Since 1997, Sistina Software Inc. [326] is marketing *GFS*.

For synchronization of client access to shared meta-data *GFS* up to version 3 used *Device Locks*. They help maintain meta-data coherence when meta-data is cached by several clients. The locks are implemented on the storage devices (disks) and accessed with the SCSI Device Lock command called *Dlock*. The Dlock command is independent of all other SCSI commands, so devices supporting the locks have no awareness of the nature of the resource that is locked. The file system provides a mapping between files and Dlocks. GFS-4 supports an abstract lock module that can exploit almost any globally accessible lock space, not just Dlocks. This is important because it allows *GFS* cluster architects to buy any disks they like, not just disks that contain Dlock firmware.

To coalesce a heterogeneous collection of shared storage into a single logical volume *GFS* uses the *Pool Logical Volume Driver*. It was developed with *GFS* to provide simple logical device capabilities. If *GFS* is used as a local file system where no locking is needed, then Pool is not required.

Meta-data is distributed throughout the network storage pool rather than concentrating it all into a single superblock. Multiple resource groups are used to partition meta-data, data and data blocks, into separate groups to increase client parallelism and file system scalability, avoid bottlenecks, and reduce the average size of typical meta-data search operations. One or more resource groups may exist on a single device or a single resource group may include multiple devices.

GFS uses a *Transaction* and *Log Manager* to implement the journaling functionality. When a journal recovery is initiated by a client, a recovery kernel thread is called with the expired client's ID. The machine then attempts to begin recovery by acquiring the journal lock of the failed client.

Giggle (GIGa-scale Global Location Engine)

type:

Replication

contact:

Ann Chervenak, `annc@isi.edu`

description:

Giggle a parameterized architectural framework within which a wide range of *Replica Location Services (RLS)* can be defined.

motivation:

In wide-area computing systems, it is often desirable to create remote read-only copies (*replicas*) of data elements (*files*). Replication can be used to reduce access latency, improve data locality, and/or increase robustness, scalability and performance for distributed applications. A system that includes replicas requires a mechanism for locating them.

features:

- design allows users to make tradeoffs among consistency, space overhead, reliability, update costs, and query costs by varying six system parameters
- uses compression to reduce network traffic and the cost of maintaining *Replica Location Indices (RLIs)*

related work:

►*EDG Replica Manager*, ►*GDMP*, ▷*Globus*

citation:

[88]

details:

A *Local Replica Catalog (LRC)* maintains information about replicas at a single replica site. Among others, a LRC must meet the following requirements:

Contents It must maintain a mapping between arbitrary *Logical File Names (LFNs)* and the *Physical File Names (PFNs)* associated with those LFNs on its storage system(s).

Queries It must respond to the following queries: Given an LFN, find the set of PFNs associated with that LFN. Given a PFN, find the set of LFNs associated with that PFN.

State Propagation The LRC must periodically send information about its state.

While the various LRCs collectively provide a complete and locally consistent record of all extant replicas, they do not directly support user queries about multiple replica sites. An additional index structure is required to support these queries. The *Giggle* framework structures this index as a set of one or more RLIs, each of which contains a set of `(LFN, pointer to an LRC)` entries. A variety of index structures can be defined with different performance characteristics, simply by varying the number of RLIs and amount of redundancy and partitioning among the RLIs.

A wide range of global index structures can be characterized in terms of six parameters (G, P_L, P_R, R, S, C). The parameter G specifies the total number of RLIs in the replica location service. P_L determines the type of logical file name space partitioning of information sent to the RLIs. The parameter P_R indicates the type of LRC name space partitioning by which a particular RLI receives state updates from only a subset portion of all LRCs. R indicates the number of redundant copies of each RLI maintained in the replica location service. The soft state algorithm S indicates the type and frequency of updates sent from LRCs to RLIs. Finally, the parameter C indicates whether a compression scheme is used to reduce network traffic and the cost of maintaining RLIs.

The following requirements must be met by an *RLI* node:

Secure Remote Access An RLI must support authentication, integrity and confidentiality and implement local access control over its contents.

State Propagation An RLI must accept periodic inputs from LRCs describing their state. If the RLI already contains an LFN entry associated with the LRC, then the existing information is updated or replaced. Otherwise, the index node creates a new entry.

Queries It must respond to queries asking for replicas associated with a specified LFN by returning information about that LFN or an indication that the LFN is not present in the index.

Soft State An RLI must implement time outs of information stored in the index. If an RLI entry associated with an LRC has not received updated state information from the LRC in the specified time out interval, the RLI must discard the entries associated with that LRC.

The developers argue that strong consistency is not required in the RLS, which allows to use a Soft State protocol to send LRC state information to relevant RLIs, which then incorporate this information into their indices. Soft State is information that times out and must be periodically refreshed. There are two advantages to Soft State mechanisms. First, stale information is removed implicitly, via time outs, rather than via explicit delete operations. Hence, removal of data associated with failed or inaccessible replica sites can occur automatically. Second, RLIs need not maintain persistent state information, since state can be reconstructed after RLI failures using the periodic Soft State updates from LRCs. Various Soft State update strategies with different performance characteristics can be defined.

Failure Recovery An RLI must contain no persistent replica state information. That is, it must be possible to recreate its contents following RLI failure using only Soft State updates from LRCs.

GIOD (Globally Interconnected Object Databases)

type:

Other / Distributed Object Database

contact:

Julian Bunn, `julian@cacr.caltech.edu`

url:

`http://pcbunn.cacr.caltech.edu`

description:

In *GIOD* several key technologies have been adopted that seem likely to play significant roles in the LHC [211] computing systems: OO software (C++ and Java), commercial *OO Database Management Systems (ODBMS)* – specifically Objectivity/DB, *Hierarchical Storage Management (HSM)* systems (specifically ►*HPSS*) and fast networks. The kernel of the *GIOD* system prototype is a large (approx. 1 Terabyte) Object database of approx. 1,000,000 fully simulated LHC events.

related work:

►*HPSS*, LHC [211]

citation:

[197]

Globus Replica Management

type:

Replication

contact:

Ann Chervenak, `annc@isi.edu`

url:

`http://www.globus.org/datagrid/replica-management.html`

description:

One fundamental data management service which is needed in a Grid environment is the ability to *register and locate multiple copies* of data files. The *Globus Replica Management* architecture is one implementation of this service.

motivation:

Goals are:

- support for data which is distributed and replicated to world-wide distributed sites
- to provide a general replication system that sits on top of database management systems or data stores since a broad user community will use several different storage technologies

features:

The following services are provided by the *Globus Replica Management* architecture:

- creating new copies of a complete or partial dataset
- registering these copies in a *Replica Catalog*
- allowing users and applications to query the catalog to find all existing copies of a particular file or collection of files
- selecting the *best* replica for access based on storage and network performance predictions provided by a *Grid Information Service*

related work:

- ►*EDG Replica Manager*, ►*GDMP*, ►*Giggle*
- In [116] a new Grid service, called *Grid Consistency Service (GCS)*, is proposed, which allows for replica update synchronization and consistency maintenance.
- [373] discusses the design an implementation of a high-level replica selection service that uses information regarding replica location and user preference to guide selection from among storage replica alternatives.
- In [175] the problem of building very large location tables containing at least some 10^{10} objects is discussed.

citation:

[5], [6]

details:

The *Globus Replica Management* architecture is a layered architecture. For transferring files between sites ►*GridFTP* is used. At the lowest level, the *Fabric Layer* (see ►*DGRA*), is a *Replica Catalog* that allows users to register files and provides mappings between logical names for files and collections and the storage system locations of one or more replicas of these objects. At the *Collective Layer* there are higher-level services for replica selection and management.

The Replica Catalog registers three types of entries:

Logical Collection This is a user-defined group of files. Aggregating files should reduce both the number of entries in the catalog and the number of catalog manipulation operations required to manage replicas.

Replica Location These entries contain all the information required for mapping a logical collection to a particular physical instance of that collection. One location entry corresponds to exactly one physical storage system location.

Logical Files Users may also want to characterize individual files. For this purpose, the Replica Catalog includes optional entries of this type.

The ▷*Globus* team implemented an API for low-level Replica Catalog manipulation as a C library called `globus_replica_catalog.c`. There is also a command-line tool that provides similar functionality.

Possible operations on Replica Catalog entries fall into three general categories:

- create and delete entire entries
- add, list or delete attributes of an entry
- list or search for specified entries

The *Globus Replica Management Service* provides higher-level functionality on top of the low-level Replica Catalog. In addition to making low-level Replica Catalog API calls, the functions in the *Replica Management* API can manipulate storage systems directly, including copying files and checking file status, such as last modification time. These are some of the functions of the *Replica Management Service*:

Catalog Creation These functions create and populate one or more entries in a Replica Catalog. There are basic functions to create empty logical collection and location objects, to *register* and *publish* filenames. A client registers a file that already exists on a storage system by adding the filename to collection and location entries in the catalog. Alternatively, the client publishes a file into logical collection and location entries by first copying the file onto the corresponding storage system and then updating the catalog entries.

File Maintenance These functions include copy, update and delete operations on physical files with corresponding Replica Catalog updates.

Access Control These functions control who is allowed to access and make replicas of individual files and logical collections.

One of the key features of this architecture is that the *Replica Management Service* is orthogonal to other services such as replica selection and meta-data management. Figure 4.27 shows a scenario where an application accesses several of these orthogonal services to identify the best location for a desired data transfer.

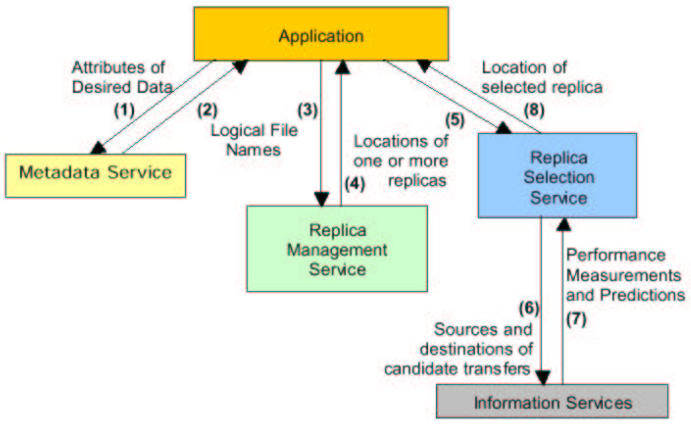

Application
Attributes of Desired Data
(1)
(2)
Logical File Names
(3)
Locations of one or more replicas
(4)
(5)
Location of selected replica
(8)
Metadata Service
Replica Management Service
Replica Selection Service
Performance Measurements and Predictions
(7)
(6)
Sources and destinations of candidate transfers
Information Services

Figure 4.27: Globus Replication Management Architecture: Data Selection Scenario

1. For retrieving the desired data the application specifies the characteristics of the desired data and passes this attribute description to a *Metadata Catalog*.
2. The Metadata Catalog queries its attribute-based indices and returns a list of logical files that contain data with the specified characteristics to the application.
3. The application passes these logical file names to the *Replica Management Service*.
4. The *Replica Management Service* returns to the application a list of physical locations for all registered copies of the desired logical files.
5. Next, the application passes this list of replica locations to a *Replica Selection Service*, which identifies the source and destination storage system locations for all candidate data transfer operations.
6. The Replica Selection Service sends the candidate source and destination locations to one ore more information services.
7. These services provide estimates of candidate transfer performance based on Grid measurements and/or predictions.
8. The Replica Selection Service chooses the best location and returns location information for the selected replica to the application.

Following this selection process, the application performs the data transfer operations.

GPFS (General Parallel File System)

type:

File System

contact:

Frank Schmuck, `schmuck@almaden.ibm.com`

url:

`http://www.almaden.ibm.com/StorageSystems/file_systems/GPFS/`

description:

GPFS is IBM's parallel, shared-disk file system for cluster computers, available on the RS/6000 SP parallel supercomputer and on Linux clusters.

motivation:

One fundamental drawback of clusters is that programs must be partitioned to run on multiple machines, and it is difficult for these partitioned programs to share resources. Perhaps the most important resource is the file system. In the absence of a cluster file system, individual components of a partitioned program must share cluster storage in an ad-hoc manner. This typically complicates programming and limits performance. *GPFS* was developed to overcome these problems.

features:

- provides, as closely as possible, the behaviour of a general-purpose POSIX file system running on a single machine
- supports file systems with up to 4096 disks of up to 1 Terabyte each for a total of 4 Petabytes per file system
- supports fully parallel access both to file data and meta-data
- performs administrative functions in parallel and while the file system is online

related work:

►*CXFS*, ►*Frangipani*, ►*GFS*

citation:

[308]

details:

Rather than relying on a separate volume manager layer, *GPFS* implements striping in the file system. Large files are divided into equal size blocks, and consecutive blocks are placed on different disks in a round-robin fashion.

GPFS prefetches data into its buffer pool, issuing requests in parallel. Similarly, dirty data buffers that are not longer being accessed are written to disk in parallel. *GPFS* recognizes sequential, reverse sequential, as well as various forms of strided access patterns. For applications that do not fit into one of these patterns, *GPFS* provides an interface that allows passing prefetch hints to the file system.

Each cluster node has a separate log for each file system it mounts, stored in that file system. Because this log can be read by all other nodes, any node can perform recovery on behalf of a failed node – it is not necessary to wait for the failed node come back to life.

GPFS uses a centralized *Global Lock Manager* in conjunction with local lock managers in each file system. The Global Lock Manager coordinates locks between local lock managers by handing out *Lock Tokens*, which convey the right to grant distributed locks without the need for a separate message exchange each time a lock is acquired or released.

GPFS uses *Byte-Range-Locking* to synchronize reads and writes to file data, thus allowing parallel applications to write concurrently to different parts of the same file, while maintaining POSIX read/write atomicity semantics. If POSIX semantics is not required, *GPFS* allows disabling Byte-Range-Locking by switching to *Data Shipping* mode. In this mode, file blocks are assigned to nodes in a round-robin fashion so that each block will be read or written only by one particular node. *GPFS* forwards read and write operations originating from other nodes to the node responsible for a particular data block. In this mode there is never any data shared between nodes so there is no need for distributed locking.

greedyWriting

type:

Device Level Method

contact:

David A. Hutchinson, `hutchins@cs.duke.edu`

description:

greedyWriting is an optimal online algorithm for output scheduling. It uses a FIFO queue for each disk. It is fed the blocks b_i of a write request sequence σ one at a time. Block b_i, labeled with an associated disk identifier, is put in to the buffer pool

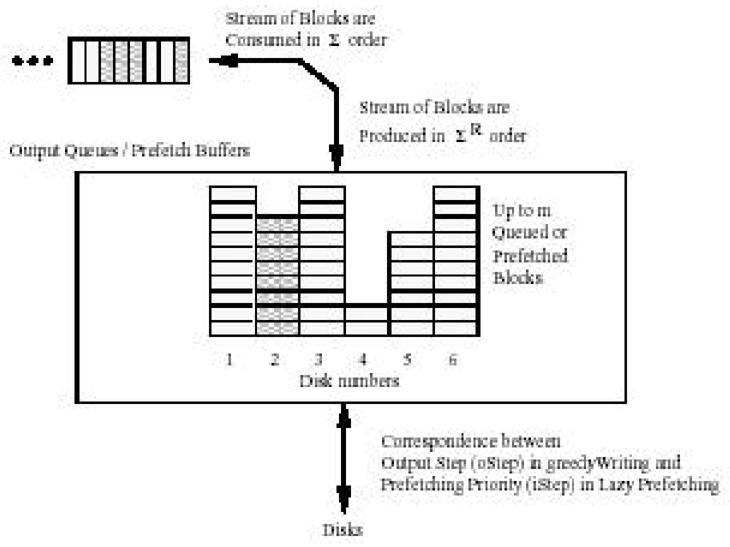

Stream of Blocks are Consumed in Σ order
Stream of Blocks are Produced in Σ^R order
Output Queues / Prefetch Buffers
Up to m Queued or Prefetched Blocks
1 2 3 4 5 6
Disk numbers
Correspondence between Output Step (oStep) in greedyWriting and Prefetching Priority (iStep) in Lazy Prefetching
Disks

Figure 4.28: Greedy Writing: Duality between Prefetching Priority and Output Step

and queued behind the other requests buffered for the same disk. One output step, consisting of leading blocks of each of the disk queues, is scheduled each time the pool gets completely full.

In [180] the authors show the duality between the *Online Output Scheduling Problem* and the *Offline* ►*Prefetching scheduling problem* as depicted in Figure 4.28. For read sequences consisting of distinct blocks *greedyWriting* mutates into *Lazy* ►*Prefetching* via this duality which also implies its optimality as a prefetching algorithm.

Algorithm *Lazy* ►*Prefetching* is equivalent to the *Reverse Aggressive* algorithm [195] for the case of read-once sequences.

An optimal writing/ ►*Prefetching* algorithm on its own may be of little practical use if all requests happen to go to the same disk. The concept of duality is applied to transfer results on writing [303] to ►*Prefetching*. For this case *prudentPrefetching* was developed.

For random allocation the algorithm gives performance guarantees.

related work:

The asynchronous variant of the algorithms can be found in [110].
Reverse Aggressive Algorithm [195], ►*Prefetching*

citation:

[180]

Grid Datafarm see ***Gfarm***

GridExpand

type:

Mass Storage System

contact:

José M. Pérez, `jmperez@arcos.inf.uc3m.es`

url:

`http://www.arcos.inf.uc3m.es`

description:

GridExpand integrates existing servers using protocols like NFS [302], CIFS [93] or WebDAV [397], facilitating the development of applications by integrating heterogeneous Grid resources under homogeneous and well known interfaces like POSIX and ►*MPI-IO*. *GridExpand* applies the Parallel I/O techniques used in most parallel file systems to Grid environments.

motivation:

To provide a Parallel I/O middleware that allows to integrate the existing heterogeneous resources.

related work:

►*Expand*

citation:

[271]

details:

The *GridExpand* architecture is presented in Figure 4.29. The architecture allows the usage of several heterogeneous clusters to define parallel distributed partitions. *GridExpand* uses the available protocols to communicate with storage servers, like NFS, CIFS or WebDav, without needing specialized servers. *GridExpand* combines several servers to provide a generic striped partition which allows to create several types of file systems.

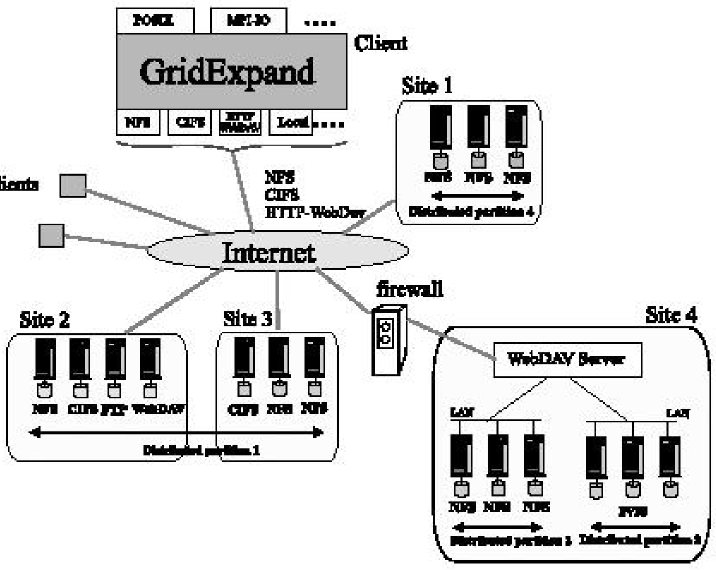


Figure 4.29: GridExpand: Architecture

GridExpand provides an *Abstract File Interface (AFI)* that allows the implementation of the typical interfaces (POSIX, Win32, ►*MPI-IO*) above it, and supports other advanced features as cache policy, ►*Prefetching*, parallelism degree configuration and fault tolerance. The access to the servers and storage resources is provided by an *Abstract I/O Adapter* similar to ►*ADIO*.

GridFTP

type:

Data Transfer Protocol

contact:

Steve Tuecke, `tuecke@mcs.anl.gov`

url:

`http://www.globus.org/datagrid/gridftp.html`

description:

GridFTP is a common data transfer and access protocol, that provides secure and efficient data movement in Grid environments. This protocol, which extends the standard FTP protocol, provides a superset of the features offered by the various Grid storage systems currently in use.

features:

- GSI and Kerberos [192] support
- third party control of data transfer
- parallel and striped data transfer
- partial file transfer
- automatic negotiation of TCP buffers / window sizes
- support for reliable and restartable data transfer

related work:

►*DPSS*, ►*FTP-Lite*, ►*RFT*

citation:

[5], [7], [9]

details:

There are a number of storage systems in use by the Grid community. Unfortunately, most of these storage systems utilize incompatible, and often unpublished protocols for accessing data, and therefore require the use of their own client libraries to access data. This effectively partitions the datasets available on the Grid.

To overcome these incompatible protocols, *GridFTP* can be used as a common protocol. It extends the standard FTP protocol and provides a superset of the features offered by the various Grid storage systems. FTP was chosen as the protocol to extend because it is the most commonly used protocol for data transfer on the Internet, is widely implemented and is a well-understood IETF standard protocol.

The implementation of the *GridFTP* protocol takes the form of two APIs and corresponding libraries: `globus_ftp_control` and `globus_ftp_client`. The libraries use high-performance I/O and security services provided by the ▷*Globus* Toolkit.

In addition to the libraries, there also exists an API/library (`globus_gass_copy`) and a command-line tool (`globus-url-copy`) that integrates *GridFTP*, HTTP, and local file I/O to enable secure transfers using any combination of these protocols.

The `globus_ftp_control_library` implements the control channel API. This API provides routines for managing a *GridFTP* connection, including authentication, creation of control and data channels, and reading and writing data over data channels. Having separate control and data channels, as defined in the FTP protocol standard, greatly facilitates the support of such features as parallel data transfer, striped transfers and third-party data transfers. For parallel and striped transfers, the control channel is used to specify a `put` or `get` operation. Multiple parallel TCP data channels provide concurrent transfers. In third-party transfers, the initiator monitors or aborts transfers via the control channel, while data is transferred over on or more data channels between source and destination sites.

The `globus_ftp_client_library` implements the *GridFTP* client API. This API provides higher-level client features on top of the `globus_ftp_control_library`, including complete file `get` and `put` operations, calls to set the level of parallelism for parallel data transfers, partial file transfer operations, third-party transfers, and eventually, functions to set TCP buffer sizes.

GSH (Grid Service Handle) see ***OGSI***

GSI-SFS (Grid Security Infrastructure – Self-certifying File System)

type:

File System

contact:

Shingo Takeda, `stakeda@ais.cmc.osaka-u.ac.jp`

url:

`http://www.biogrid.jp/e/research_work/gro1/gsi_sfs/index.html`

description:

GSI-SFS is a user-oriented and secure file system seamlessly integrated in the Grid environment.

motivation:

Few existing Grid file transfer services find a good the trade-off between convenience and security, which leads to difficulties when dealing with confidential data on the Grid. To satisfy such users demand, *GSI-SFS* was developed.

features:

- *Single Disk Image (SDI)*: Users can access remote data without being aware of the location of the data.
- Exclusiveness: Each user has his/her own SDI.
- Data confidentiality: Strong encryption is used when transferring data over insecure networks.

application:

GUIDE Grid Portal [168]

related work:

GSI [68, 166], SFS [317]

citation:

[167]

details:

GSI-SFS has been developed based on the two key technologies. The first is the *Grid Security Infrastructure (GSI)* [68, 166], the second is *Self-Certifying File System (SFS)* [317].

GSI is a Grid service and is part of the ▷*Globus* Toolkit. One of the features of GSI is single sign-on which allows users to access computational resources on the Grid in a transparent manner by typing the passphrase once.

SFS is a secure, global network file system with completely decentralized control. SFS lets users access their files from anywhere and share them with anyone. Anyone can set up an SFS server, and any user can access any server from any client. SFS lets users share files across administrative realms without involving administrators or certification authorities.

GSI-SFS consists of three layers:

NFS Layer Because the lowest layer of *GSI-SFS* is NFS [302], it works on many systems supporting NFS.

SFS Extension Layer SFS authenticates a user and a host with their public keys. After successful authentication, SFS forwards local NFS accesses to remote locations encrypting all data transferred over network.

GSI-SFS Layer SFS's public key authentication method works well in small environments, but doesn't scale well to a large number of users and hosts. To use SFS in a Grid environment a *GSI-SFS Key Server* and *Client* was implemented which automate the key registration securely with GSI.

GSR (Grid Service Reference) see ***OGSI***

GT2 (Globus Toolkit version 2) see ▷**Globus**

GT3 (Globus Toolkit version 3) see ▷**Globus**

H

HDF5 (Hierarchical Data Format 5)

type:

Data Format

contact:

HDF Helpdesk, `hdfhelp@ncsa.uiuc.edu`

url:

`http://hdf.ncsa.uiuc.edu/HDF5/`

description:

HDF5 is a format and software for scientific data capable of I/O. It is efficiently storing images, multidimensional arrays, tables etc.

motivation:

The aim of *HDF5* is to develop, promote, deploy, and support open and free technologies that facilitate scientific data storage, exchange, access, analysis and discovery.

features:

- support for high-performance applications
  - ability to create complex data structures
  - complex subsetting
  - efficient storage
  - flexible I/O (parallel, remote, etc.)
- support for key language models
  - OO compatible
  - C, Fortran, Java, C++
- mounting files
- *Virtual File Layer (VFL)*: a public API for writing I/O drivers
- remote data access

related work:

►*DPSS*, ►*HPSS*

citation:

[248], [295]

details:

HDF5 can store large numbers of large data objects, such as multidimensional arrays, tables, and computational meshes, and these can be mixed together in any way that suits a particular application. *HDF5* supports cross platform portability of the interface and corresponding file format. The main conceptual building blocks of *HDF5* are the *HDF5 Dataset* and the *HDF5 Group*. An *HDF5 Dataset* is a multidimensional array of elements of a specified datatype. When reading or writing an *HDF5 Dataset*, an application describes two datasets: a *source dataset* and a *destination dataset*. These can have different sizes and shapes and, in some instances, can involve different datatypes. *HDF5 Groups* are directory-like structures containing HDF5 Datasets, HDF5 Groups, and other objects.

The *HDF5* library contains a VFL, a public API for doing low level I/O, so that drivers can be written for different storage resources. When performing I/O in a multilayered configuration such as the one discussed here, it is useful to have a mechanism for passing information from one layer to another. To this end, the *HDF5* API provides a *Property List*, which is a list of specifications, guidelines, or hints to be used by the library itself or to be passed on by the VFL to lower layers.

HPSS (High-Performance Storage System)

type:

Mass Storage System

contact:

Dick Watson, `dwatson@llnl.gov`

url:

`http://www4.clearlake.ibm.com/hpss/index.jsp`

description:

HPSS is a *Hierarchical Storage System (HSS)* with a scalable, modular design resulting in many distributed servers, each maintaining meta-data germane to its function.

motivation:

The overall objective of *HPSS* is to provide a scalable, high-performance, high functionality HSS for environments consisting of a combination of massively parallel machines, traditional supercomputers as well as clusters of workstations.

features:

- network-centered architecture

- third party data transfer
- high data transfer rate
- parallel operation built in
- design based on standard components
- multiple hierarchies and classes of services
- data integrity through transaction management

application:

National Climatic Data Center (NCDC) [247], Visible Embryo NGI Project [379]

related work:

▶*Enstore*

citation:

[97], [157], [179], [396]

details:

The components used to define the structure of the *HPSS* name space are *Filesets* and *Junctions*. A *Fileset* is a logical collection of files that can be managed as a single administrative unit, or more simply, a disjoint directory tree. A *Junction* is an object, managed by the *HPSS Name Server*, that links a path name to a fileset.

The components containing user data include *Bitfiles*, *Physical* and *Virtual Volumes*, and *Storage Segments*. Files in *HPSS*, called *Bitfiles*, are logical strings of bytes, even though a particular Bitfile may have a structure imposed by its owner. A *Physical Volume* is a unit of storage media on which *HPSS* stores data, whereas a *Virtual Volume* is used by the *Storage Server* to provide a logical abstraction or mapping of Physical Volumes. A *Storage Segment* is an abstract storage object which is mapped onto a Virtual Volume.

Components containing meta-data, describing the attributes and characteristics of Bitfiles, Volumes, and Storage Segments, include *Storage Maps, Classes of Service (COS), Storage Classes*, and *Hierarchies*. A *Storage Map* is a data structure used by the Storage Server to manage the allocation of storage space on Virtual Volumes. Each Bitfile has an important attribute called *COS*. The COS defines a set of parameters associated with operational and performance characteristics of a Bitfile. A *Storage Class* defines a set of characteristics and usage parameters to be associated with a particular grouping of *HPSS* Virtual Volumes. A *Hierarchy* defines the storage classes on which files in that hierarchy are to be stored.

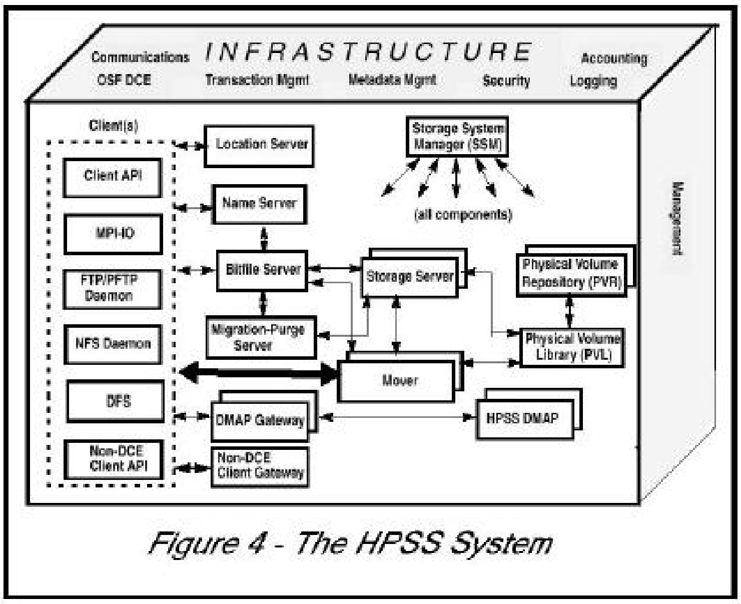


Figure 4.30: HPSS: System Overview

HPSS Servers include the *Name Server, Bitfile Server*, *Migration/Purge Server*, *Storage Server, Location Server, Data Management Application (DMAP) Gateway, Physical Volume Library*, *Physical Volume Repository*, *Mover*, *Storage System Manager*, and *Non-DCE Client Gateway*. Figure 4.30 provides a simplified view of the *HPSS* system. Each major server component is shown, along with the basic control communications paths (thin arrowed lines). The thick line reveals actual data movement. Infrastructure items (those components that *glue together* the distributed servers) are shown at the top of the cube in gray scale.

The *Name Server (NS)* translates a human-oriented name to an *HPSS* object identifier. Objects managed by the NS are files, filesets, directories, symbolic links, junctions and hard links. The NS also provides access verification to objects. The *Bitfile Server (BFS)* provides the abstraction of logical bitfiles to its clients. The *Migration/Purge Server (MPS)* helps the local site implement its storage management policies by managing the placement of data on *HPSS* storage media using site-defined policy mechanisms. The *Storage Servers (SSs)* provide a hierarchy of storage objects: Storage Segments, Virtual Volumes, and Physical Volumes. The *Location Server (LS)* acts as an information clearinghouse to its clients to enable them to locate servers and gather information from both local and remote *HPSS* systems. The *DMAP Gateway (DMG)* acts as a conduit and translator between the distributed file system and the *HPSS Servers*. The *Physical Volume Library (PVL)* manages all *HPSS Physical Volumes*. The *Physical Volume Repository (PVR)* manages all *HPSS* cartridges. The purpose of the *Mover (MVR)* is to transfer data from a source device to a sink device. *Storage System Management (SSM)* roles cover a wide range, including aspects of configuration, initialization, and termination tasks.

I

InterMezzo

type:

File System

contact:

Peter J. Braam, `braam@clusterfilesystem.com`

url:

`http://www.inter-mezzo.org/`

description:

InterMezzo is a distributed file system with a focus on high availability.

motivation:

InterMezzo will be suitable for replication of servers, mobile computing, managing system software on large clusters, and for maintenance of high-availability clusters.

related work:

►*Lustre*

citation:

[383]

details:

On both clients and servers, *InterMezzo* uses an existing file system to store its data, and tracks file system changes in *hidden* directories in those file systems.

InterMezzo basically consists of kernel components and a server process known as *InterSync* that synchronizes files on *InterMezzo* client and server systems.

An instance of *InterSync* runs on each *InterMezzo* client, keeping cached files on the client system synchronized with the contents of the *InterMezzo* server's exported file system. The *InterMezzo* server maintains a record of changes made to any files in its exported file system(s) in a file known as a *Kernel Modification Log* that is stored in the exported *InterMezzo* file system. Each client's InterSync process periodically polls the server, retrieves this file, and scans it for records related to cached files.

In order to do active synchronization an InterSync server can run on the *InterMezzo* server. Active synchronization guarantees that changes to files on the server are propagated to clients, but also guarantees that changes to cached files on client systems are propagated back to the server. Changes to files on the server are then propagated out to all other clients in the standard fashion.

Communication is done using the HTTP protocol.

I/O Bounds

type:

Definition

description:

The I/O performance of many algorithms and data structures can be expressed in terms of the bounds for fundamental operations:

- scanning (a.k.a. streaming or touching) a file of N data items, which involves the sequential reading or writing of the items in the file
- sorting a file of N data items, which puts the items into sorted order
- searching online through N sorted data items
- outputting the Z answers to a query in a blocked output-sensitive fashion

related work:

►*PDM*

citation:

[380]

I/O Communities

type:

Definition

contact:

`condor-admin@cs.wisc.edu`

description:

A system that allows jobs and data to meet by binding execution and storage sites together into *I/O Communities* which then participate in the wide-area system.

motivation:

Improving the performance of a key HEP's simulation on an international distributed system.

application:

CMS [95]

related work:

ClassAds [94], ▷*Condor*

citation:

[346]

details:

An *I/O Community* consists of several CPUs that gather around a storage device. Programs executing at such CPUs are encouraged to use the community I/O device for storing and retrieving data. By sharing the device, similar applications may reduce their consumption of wide-area resources.

The architecture is built around *Storage Appliances* and *Interposition Agents* as depicted in Figure 4.31. A *Storage Appliance* serves as the meeting place for an *I/O Community*. It is frequently conceived as a specialized hardware device. The appliance is most useful when it speaks a number of protocols. An *Interposition Agent* is a small piece of software that inserts itself between an application and the native OS, since standard applications rarely speak protocols.

In a Computational Grid, community resources come, go, and change without warning. Figure 4.31 shows how *CPU Discovery, Device Discovery*, and *Replica Discovery* fits into the system for finding community resources. Before execution, *CPU discovery* must be performed to find a CPU with the proper architecture, OS, and so on. During execution, *Device Discovery* must be performed to find ones membership in a community. Also during execution, *Replica Discovery* may be performed to locate items outside of a job immediate community.

The ClassAds [94] mechanism is used to represent all of the properties, requirements, and preferences involved in an *I/O Community*.

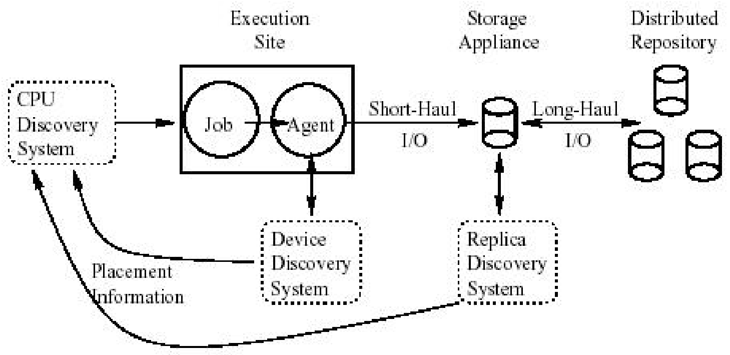


Figure 4.31: I/O Communities: Model

J

Java I/O Extensions

type:

Data Format

contact:

Phillip M. Dickens, `dickens@iit.edu`

url:

`http://www.cs.berkeley.edu/˜bonachea/java/`

description:

Java I/O Extensions address the deficiencies in the Java I/O API and improve performance dramatically.

motivation:

To be successful in the HPC domain, Java must not only be able to provide high computational performance, but also high-performance I/O.

related work:

▶*Parallel Unix Commands*

citation:

[62], [112]

details:

The following approaches for performing parallel file I/O remove much of the overhead currently associated with array I/O in Java and/or are ways of working around the problem that Java does not directly support the reading or writing of arrays of any data type other than bytes:

Using Raw Byte Arrays Multiple threads of a parallel program can write to different parts of a byte array. Each thread in the parallel program creates a `RandomAccessFile` object, calculates its offset in the shared file, and seeks to that position. It then uses the `write` method defined by the `RandomAccessFile` to write its portion of the byte array in a single operation.

Converting to/from an Array of Bytes I/O involving byte arrays is simple, but Java provides no methods for performing I/O operations on other data types, such as integers, floats and doubles. Simple casts are not allowed either. Other array types have to be converted by right-shifting one byte at a time into a byte array by first reading into a byte array and then converting the bytes into the desired array type. Sign bits have to be taken care of.

Using Data Streams It is possible to read/write a *single* integer at a time by using the methods defined in `DataInput` and `DataOutput`. The `RandomAccessFile` class makes it relatively easy to perform Parallel I/O operations using data streams.

Using Buffered Data Streams The `RandomAccessFile` class does not implement buffering, and the `FilterInput` and `FilterOutput` streams only work with objects of type `InputStream` and `OutputStream`. There is a way to use system buffering for a `RandomAccessFile` object: a `RandomAccessFile` can be chained to a `FileInputStream` or `FileOutputStream` object through its file descriptor. `FileInputStream` or `FileOutputStream` objects can be chained to a `BufferedInputStream` or `BufferedOutputStream` object, which can be chained to a `DataInputStream` or `DataOutputStream` object.

Using Buffering with Byte Array Streams Another approach to buffering a data input or output stream is to chain it to an underlying byte array stream. Then the read and write methods involved on the data stream will be directed to the underlying byte array stream rather than directly to disk.

Bulk I/O Three new subclasses (`BulkDatInputStram`, `BulkDataOutputStream` and `BulkRandomAccessFile`) added to the `java.io` hierarchy are shown in Figure 4.32. These new classes implement the methods from two new interfaces, `BulkDataInput` and `BulkDataOutput`, which are subinterfaces of the `DataInput` and `DataOutput` interfaces that currently provide single-primitive I/O. `BulkDataInput` and `BulkDataOutput` are both very simple. Each class adds two new methods for performing array-based I/O: one method for performing I/O on an entire array and a second for performing I/O on a contiguous subset of the elements in an array. The `BulkDataInputStream` class implements the methods from `BulkDataInput`, `BulkDataOutputStream` implements the methods from `BulkDataOutput` and `BulkRandomAccessFile` implements both interfaces.

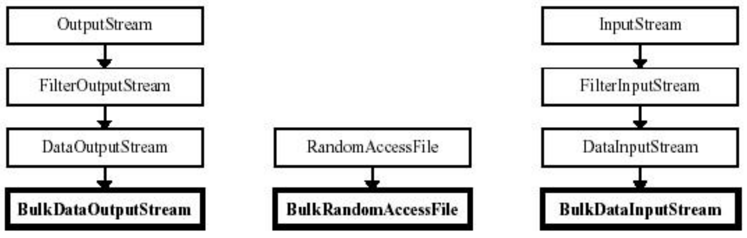


Figure 4.32: Java I/O Extensions: New Subclasses

K

Kangaroo

type:

Data Transfer Protocol

contact:

Douglas Thain, `thain@cs.wisc.edu`

url:

`http://www.cs.wisc.edu/condor/kangaroo/`

description:

Kangaroo is a wide-area data-movement system.

motivation:

Distributed systems are prone to performance variations, failed connections, and exhausted resources. Traditional OSs deal with the vagaries of disks by making a background process responsible for scheduling, coalescing, and retrying operations. *Kangaroo* applies the same principle to Grid Computing.

related work:

▷*Condor*, ►*GASS*, ▷*Legion*

citation:

[345]

details:

The *Kangaroo* architecture is centered around a chainable series of servers that implement a simple interface:

```
int kangaroo_get (host,path,offset,length,data)
void kangaroo_put (host,path,offset,length,data)
int kangaroo_commit ()
int kangaroo_push (host,path)
```

To perform a *hop* (Figure 4.33) *Kangaroo* places a server at the execution site. It satisfies `put` requests by immediately spooling them to disk. A background process, the *Mover*, is responsible for reading these requests and forwarding them to the destination as the network permits. Read operations may be satisfied from cached data without contacting the destination server. Multiple hops allow transfers over many

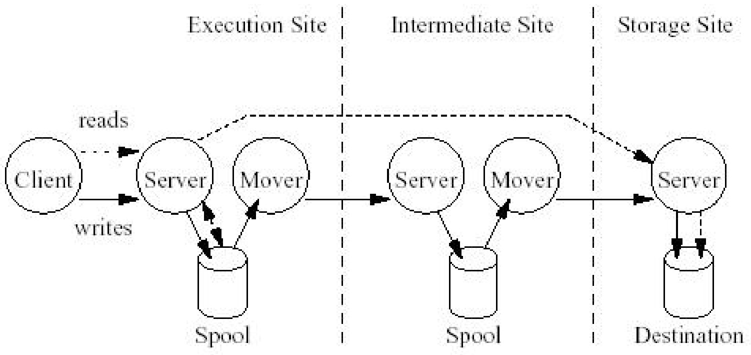


Figure 4.33: Kangaroo: Two Hop Kangaroo

network segments to be performed incrementally, avoiding the need to co-allocate network resources along all hops.

Applications that need consistency guarantees have to synchronize with the primitives `commit` and `push`. `commit` is used to make data safe from crashes while `push` is used to make changes visible to others. `commit` causes the caller to block until all outstanding changes have been written to *some* stable storage. The changes are not necessarily visible to all other callers. `push` causes the caller to block until all outstanding changes have been delivered to their respective destinations. `push` is recursively called until the target host is reached. A success message is sent after committing all outstanding data into the local file system. In case of failures the top-level caller needs to retry.

The adaptation layer converts POSIX operations into the appropriate *Kangaroo* consistency operations. Upon program termination the adaptation layer forces a `commit` to the local *Kangaroo* server. If a job needs to wait until all data arrives, a manual `push` should be issued. During execution, a POSIX `fsync` is also converted into a `push`.

KelpIO

type:

I/O Library

contact:

Bradley Broom, `broom@rice.edu`

url:

`http://www.cs.rice.edu/~dsystem/kelpio/`

description:

KelpIO is a domain-specific I/O library for irregular block-structured applications. It is designed to be used for application I/O, checkpointing, snapshooting, and *out-of-core* (►*EM (External Memory) Algorithms and Data Structures*) execution for programs written in the KeLP [130] programming system.

KelpIO provides high-level KeLP-like primitives for communicating array data between KeLP data elements and external arrays. *KelpIO* is layered above one or more underlying (Parallel) I/O systems. It transforms I/O from the geometric application domain of KeLP data elements into the implementation domain of the underlying I/O system. *KelpIO* is designed to be mostly independent of the underlying (Parallel) I/O library, without unnecessary duplication of functionality and only provides facilities for I/O. Three general classes of optimizations appropriate to *KelpIO* programs are considered: *File Layout Optimization*, *File Access Optimization*, and *Out-Of-Core* (►*EM (External Memory) Algorithms and Data Structures*) *Optimization*.

related work:

KeLP [130], ►*LEDA-SM*, ►*TPIE*

citation:

[66]

L

lazyPrefetching see ***greedyWriting***

LEDA-SM

type:

Intelligent I/O System

contact:

Andreas Crauser, `crauser@mpi-sb.mpg.de`

url:

`http://www.mpi-sb.mpg.de/˜crauser/leda-sm.html?74,45`

description:

LEDA-SM is an extension package of the LEDA-library [226] towards secondary memory computation. It consists of a collection of secondary memory data structures and algorithms and a development platform for new applications. *LEDA-SM* is no longer maintained.

motivation:

Enabling I/O efficiency for ►*EM Algorithms and Data Structures*.

features:

The following data structures and algorithms available in *LEDA-SM*:

- stacks and queues
- priority queues
- arrays
- B-Trees
- suffix arrays and strings
- matrix operations
- graphs and graph algorithms

related work:

►*MPI-IO*, ►*TPIE*, ►$\langle STXXL \rangle$

citation:

[98], [99]

Layer	Major Classes
algorithms	sorting, graph algorithms, ...
data structures	`ext_stack`, `ext_array`, ...
abstract kernel	`block⟨E⟩`, `B_ID`, `U_ID`
concrete kernel	`ext_memory_manager`, `ext_disk`, `ext_free_list`, `name_server`

Table 4.1: LEDA-SM: Layers

details:

LEDA-SM is designed in a modular way and consists of 4 layers (see Table 4.1).

The *Kernel Layer* of *LEDA-SM* manages secondary memory and the movement of data. It is divided into the *Abstract* and the *Concrete Kernel*. The *Abstract Kernel* consists of 3 C++ classes that give an abstract view of secondary memory. The *Concrete Level* is responsible for performing I/Os and managing disk space in secondary memory and consists of 4 C++ classes. The user can choose between several realizations of the concrete level at run-time. The Concrete Kernel defines functions for performing I/Os and managing disk blocks. These functions are used by the *Abstract Kernel* or directly by *Data Structures* and *Algorithms*. The *Application Level* consists of *Data Structures* and *Algorithms*. LEDA is used to implement the in-core part of the applications and the kernel of *LEDA-SM* is used to perform the I/Os. The physical I/O calls are hidden in the Concrete Kernel, the applications use the container class of the Abstract Kernel to transport data to and from secondary memory.

LegionFS

type:

File System

contact:

Andrew Grimshaw, `grimshaw@cs.virginia.edu`

url:

`http://legion.virginia.edu`

description:

LegionFS is a fully integrated serverless file system infrastructure which is used in ▷*Legion*

motivation:

In order to fulfill the evolving requirements in wide-area collaborations a more fully-integrated architecture is needed which is built upon the fundamental tenets of naming, security, scalability, extensibility and adaptability.

features:

Location-Independent Naming *LegionFS* utilizes a three-level naming scheme that shields users from low-level resource discovery.

Security Each component of the file system is represented as an object. Each object is its own security domain, controlled by ACLs.

Scalability Files can be distributed to any host participating in the system. This yields superior performance to monolithic solutions.

Extensibility Every object publishes an interface, which may be extended and specialized.

Adaptability *LegionFS* maintains a rich set of system-wide meta-data that is useful in tailoring an object's behaviour to environmental changes.

related work:

▷*Legion*, ►*SRB*

citation:

[400]

details:

Object Model ▷*Legion* is an object based system comprised of independent, logically address space-disjoint, active objects, which store their internal state on disk. Objects may be migrated simply by transferring this internal state to another host.

The ▷*Legion* file abstraction is a `BasicFileObject`, whose methods closely resemble Unix system calls. `ContextObjects` manage the ▷*Legion* name space.

Naming *Context Names*, user-defined text strings identify ▷*Legion* objects. A directory service called *Context Space* maps Context Names to unique, location-independent binary names, called *Legion Object Identifiers (LOIDs)*. For direct object-to-object communication, LOIDs must be bound (via a *Binding Process*) to low-level *Object Addresses*.

Security Each object is responsible for legislating and enforcing its own security policy. Authorization is determined by an ACL associated with each object. Public keys, which are embedded in an object's name enable secure communication with other objects.

Scalability *LegionFS* distributes files and contexts across the available resources in the system. This allows applications to access files without encountering centralized servers and ensures that they can enjoy a larger percentage of the available network bandwidth without contending with other application accesses. *Scheduler Objects* provide placement decisions upon object creation.

Extensibility *LegionFS* differentiates between objects according to their exported interfaces, not their implementation. If functionality warrants an additional method, it may be implemented, exported by the object, and incorporated into an newly generated library.

$LH*_{RS}$ (Linear Hashing using Reed Solomon Codes)

type:

Data Format

contact:

Witold Litwin, `Witold.Litwin@dauphine.fr`

url:

`http://ceria.dauphine.fr/witold.html`

description:

$LH*_{RS}$ is a new high-availability ►*SDDS*. The data storage scheme and the search performance of $LH*_{RS}$ are basically these of LH* [216].

motivation:

$LH*_{RS}$ is a data structure, that allows files to span over multiple servers.

features:

error recovery

related work:

LH* [216], ►*SDDS*

citation:

[217]

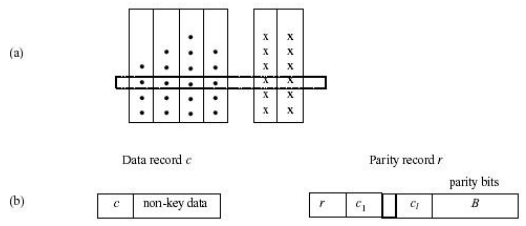


Figure 4.34: $LH*_{RS}$: (a) Bucket Group (b) Data and Parity Records

details:

A $LH*_{RS}$ file consists of an LH*-file called *data file* with the *data records* generated by the application in *data* buckets 0,1,...,M-1. The LH* data file is augmented for the high-availability with *parity* buckets to store the parity records at separate servers. A data record is identified by its key *c* and has also some non-key part. As for the generic LH*, the correct bucket *a* for data record *c* in an $LH*_{RS}$ file is given by the *linear hashing* function $_{j,n}$, $a = h_{j,n}(c)$. The parameters (j,n) called *file state* evolve dynamically. The client image consists also from *h*, but perhaps with outdated state. Data buckets are grouped in *bucket groups*. Every bucket group is provided with k1 parity buckets where the parity records for the group are stored. Figure 4.34 (a) shows a bucket group with four data buckets and their data records (●) and two parity buckets and their parity records (x). Each data record has a rank 1,2 ... that reflects the position of the record in its data bucket. A *record group* contains all the data records with the same rank *r* in the same bucket group. The record group with *r* = 3 is enclosed for example in Figure 4.34 (a). The *k* parity buckets contain parity records for each record group. A parity record consists first of the rank r of the record group, then of the primary keys *c1, c2, . . . cl* of all the (non-dummy) data records in the record group, and finally of the (generalized) parity data calculated from the data records and the number of the parity bucket. The parity data, denoted by B in Figure 4.34 (b), differ among the parity records. The ensemble of the data records in a record group and its parity records is called a *record segment* and likewise, the bucket group with its parity buckets a *bucket segment*.

Figure 4.35 illustrates the expansion of an $LH*_{RS}$ file. Parity buckets are represented shaded and above the data file, right to the last, actual or dummy, bucket of the group they serve. The file is created with data bucket 0 and one parity bucket (Figure 4.35 a). The first insert creates the first parity record, calculated from the data record and from m-1 dummy records. The data record and the parity record receive rank 1. The file is 1-available.

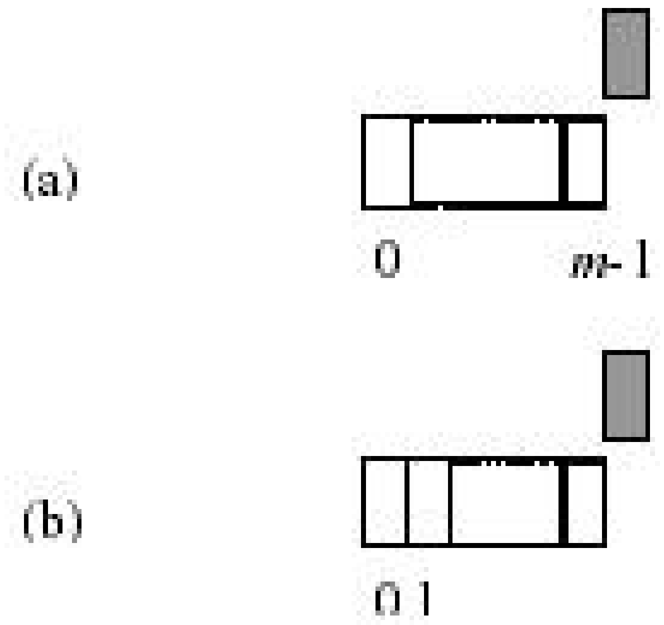


Figure 4.35: *LH*$*_{RS}$: Scalable availability of *LH*$*_{RS}$ file

When new records are inserted into data bucket 0, new parity records are generated. Each new data and parity record gets the next rank. When data bucket 0 reaches its capacity of b records, $b \gg 1$, it splits. Usually half of its records move into data bucket 1. During the split, both the remaining and the relocated records receive new consecutive ranks starting with $r = 1$. Since there are now (non-dummy) records with the same rank in both buckets, the parity records are recalculated, (Figure 4.35 b).

LH$*_{RS}$ parity calculus uses the linear *Reed Solomon (RS)* [221] codes. These are originally error correcting codes, among most efficient, since they are *Maximal Distance Separating (MDS)* codes. They are used as erasure correcting codes recovering unknown data values.

List I/O

type:

Device Level Method

contact:

Avery Ching, `aching@ece.northwestern.edu`

description:

List I/O reduces the number of I/O requests in a ►*Noncontiguous Data Access* by

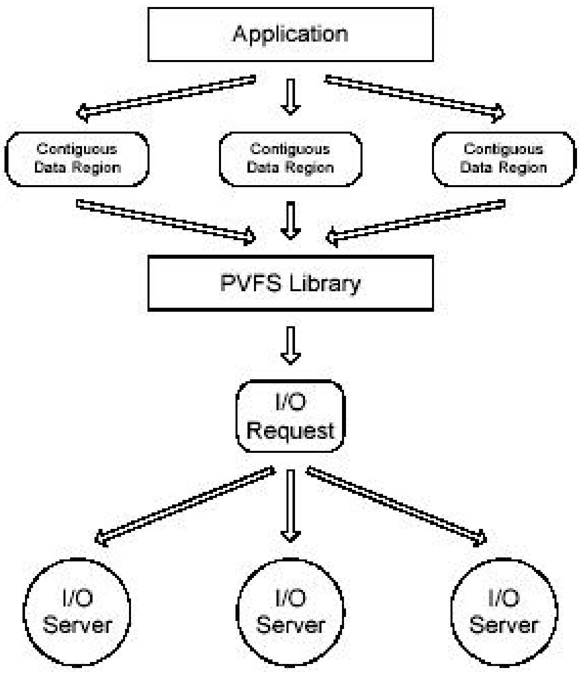


Figure 4.36: List I/O: Dataflow for Noncontiguous I/O

describing multiple file regions in a single *List I/O* request. Except for the case when noncontiguous regions are close enough to favour ►*Data Sieving*, *List I/O* performs better than ►*Data Sieving*. Support was added to the I/O server code of ►*PVFS1* to receive the trailing data and complete the I/O accesses. In this approach up to 64 contiguous file regions can be described in trailing data before another I/O request must be issued. Therefore, I/O requests that contain more file regions than the trailing data limit are broken up into several *List I/O* requests. Figure 4.36 illustrates the *List I/O* execution flow. Noncontiguous data regions are described in a single I/O request.

related work:

►*Data Sieving*, ►*Noncontiguous Data Access*, ►*PVFS1*

citation:

[91]

Load-Managed Active Storage

type:

Storage System

contact:

Rajiv Wickremesinghe, `rajiv@cs.duke.edu`

description:

Load-Managed Active Storage is an extension of ▶*TPIE* to support a flexible mapping of computations to storage-based processors exposing parallelism, ordering constraints, and primitive computation units to the system.

motivation:

The goal of *Load-Managed Active Storage* is to enable automatic dynamic placement of data and computation according to resource availability in the system.

application:

DSM-Sort [401, page 6], Spatial Indices [401, page 6], Terrain Analysis with TerraFlow [367]

related work:

▶*Active Disk*, ▶*Active Storage*, ▶*TPIE*

citation:

[401]

details:

Computation is decomposed into primitive processing steps called *Functors*, which apply specific functions to streams of records passing through them. Functors may have multiple inputs and outputs, and are composed to build complete programs that process data as it moves from stored input to output, possibly in multiple passes. A subset of the functors are capable of executing directly on *Active Storage Units (ASUs)*. These functors are *stacked* on stored data collections to process data as a side-effect of I/O operations.

To benefit from parallelism, the ▶*TPIE* stream paradigm is extended with additional data aggregation primitives and *container* types for stored data collections. *Data containers* can be *Sets*, *Streams*, and *Arrays*, illustrated in Figure 4.37. *Sets* and *Streams* are always accessed and processed in their entirety: records contained in a set or Stream are marked as *pending* or *completed* for each scan of the Container. Sets have no specific order, streams are ordered, and arrays are random-access. The model also includes a mechanism to group related records within a data collection into units called *Packets* that are always processed as a whole.

LOID (Legion Object Identifier) see ▷**Legion**

LRC (Local Replica Catalog) see ***Giggle***

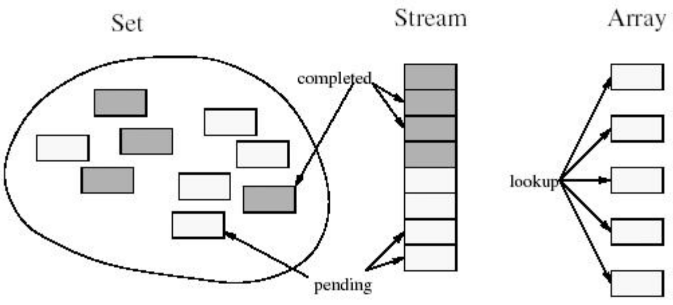


Figure 4.37: Load-Managed Active Storage: Basic Data Containers

Lustre

type:

Storage System

contact:

Peter J. Braam, `braam@clusterfilesystem.com`

url:

`http://www.lustre.org/`

description:

Lustre is a novel storage and file system architecture and implementation suitable for very large clusters.

motivation:

Lustre aims to support scalable cluster file systems for small to very large clusters.

features:

▶*Direct I/O*

related work:

▶*Intermezzo*

citation:

[64], [338]

details:

In *Lustre* clusters, a *Meta-Data Server (MDS)* manages name space operations, while *Object Storage Targets (Osts)* manage the actual storage of file data onto underlying *Object-Based Disks (OBDs)*. *Lustre* uses *Inodes* to represent files, which simply contain a reference to the OST location where the file data is actually stored.

The *Lustre* file supports ►*Direct I/O*, which is file system I/O directly to OBDs that bypass the operating system's file system buffer cache.

Lustre uses an open networking protocol known as *Portals* to provide flexible support for a wide variety of networks. At the highest level, both *Lustre's* high-performance request processing layer and the Portals stack rest on top of a *Network Abstraction Layer (NAL)* that uses a simple network description to support multiple types of networks. *Lustre* uses a distributed *Lock Manager* to manage access to files and directories and to synchronize updates. *Lustre's* Lock Manager uses intent-based locking, where file and directory lock requests also provide information about the reason the lock is being requested.

All *Lustre* configuration information is stored in XML files that are human-readable and can easily be integrated with existing administrative procedures and applications. *Lustre* is integrated with open network data management resources and mechanisms such as the LDAP and the SNMP.

M

Machine-Learning Approach to Automatic Performance Modeling

type:

General Method

contact:

Shengke Yu, `syu3@cs.uiuc.edu`

description:

The *Machine-Learning Approach to Automatic Performance Modeling* is based on the use of a platform-independent performance meta-model, which is a *Radial Basis Function (RBF)* neural network.

motivation:

A performance model for a Parallel I/O system is essential for detailed performance analysis, automatic performance optimization of I/O request handling, potential performance bottlenecks identification, and accurate application performance prediction. Also, without standard benchmarks, predictions and performance models, cross-library, cross-version, cross-platform, and even cross-installation predictions and comparisons of Parallel I/O performance are extremely difficult.

related work:

►*Panda*

citation:

[410]

details:

The meta-model is a RBF neural network [149], platform-independent and therefore portable. The input consists of twelve parameters of interest for performance modeling of the ►*Panda* system. The output of the RBF neural network is the ►*Panda* response time that the RBF network predicts for the I/O request described in the input. Each instance of training data for the meta-model consists of values for each of the twelve parameters above, plus ►*Panda's* response time for carrying out that I/O request. This approach does not use any knowledge of the underlying I/O library or platform, beyond values for ►*Panda's* internal tuning parameters.

MAPFS (Multi-Agent Parallel File System)

type:

File System

contact:

Jésus Carretero Pérez, `jcarrete@arcos.inf.uc3m.es`

description:

MAPFS is a flexible multi-agent architecture for data-intensive Grid applications.

motivation:

MAPFS's goal is to allow applications to access data repositories in an efficient and flexible fashion, providing formalisms for modifying the topology of the storage system, specifying different data access patterns and selecting additional functionalities.

features:

- system topology configuration through the usage of *Storage Groups*
- access pattern specification
- some functionalities (such as caching and ►*Prefetching*) can run in parallel on different nodes and even in the data servers

related work:

►*SRB*

citation:

[272]

details:

MAPFS is based on a client-server architecture using NFS [302] servers. *MAPFS* clients provide a Parallel I/O interface to the servers. The configuration of the system topology is achieved through the usage of *Storage Groups*. A Storage Group is defined as a set of servers clustered as groups taking the role of data repositories. Several policies can be used to try to optimize the accesses to all Storage Groups (eg.: grouping by server proximity or server similarity). *MAPFS* can be configured in order to adjust to different I/O access patterns. Hints are used in order to decrease cache faults and to prefetch the data which most probably will be used in subsequent executions. Hints can either be user-defined or built by a multi-agent subsystem.

MEMS (MicroElectroMechanical Systems)-Based Storage

type:

Storage System

contact:

Greg Ganger, ganger@ece.cmu.edu

url:

http://www.lcs.ece.cmu.edu/research/MEMS/

description:

Based on MEMS [230], this non-volatile storage technology merges magnetic recording material and thousands of recording heads to provide storage capacity of 1-10 GB of data with access times of under a millisecond and streaming bandwidths of over 50 MB per second. Another interesting aspect of *MEMS-Based Storage* is its ability to incorporate both storage and processing into the same chip.

motivation:

MEMS-Based Storage has the ability to significantly reduce application I/O stall times for file system and database workloads by incorporating both storage and processing into the same chip.

related work:

►*Active Disks*, ►*ADFS*, Millipede [234]

citation:

[162], [286], [307]

details:

Figure 4.38 illustrates the new system architecture that includes the MEMS device in the storage hierarchy. The MEMS storage module can consist of multiple MEMS devices to provide greater storage capacity and throughput. The MEMS device is accessed through the I/O bus. It can be envisioned as part of the disk drive or as an independent device. In either case, I/Os can be scheduled on the MEMS device as well as on the disk drive independently. Similar to disk caches found on current-day disk drives, it can be assumed that MEMS storage devices would also include on-device caches.

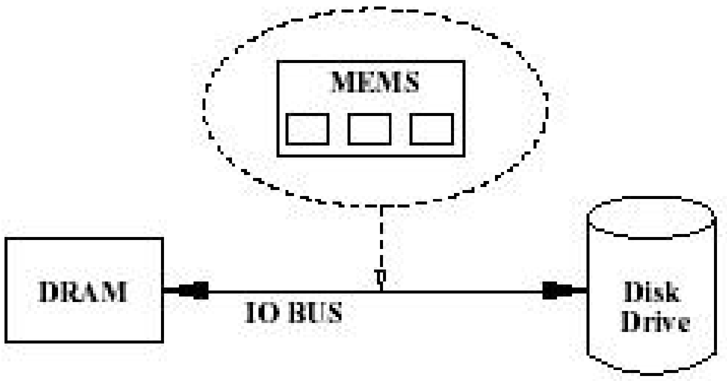


Figure 4.38: MEMS: Architecture

MercutIO

type:

I/O Library

contact:

Kumaran Rajaram, `kums@hpcl.cs.msstate.edu`

url:

`http://library.msstate.edu/etd/show.asp?etd=etd-04052002-105711`

description:

MercutIO is a portable, high-performance ▶*MPI-IO* implementation.

related work:

▶*MPI-IO*

citation:

[281]

details:

The architecture of *MercutIO* is depicted in Figure 4.39 for the case of non-collective *Bulldog Abstract File System (BAFS)*. Parallelism and portability are achieved efficiently by demarcating these two functionalities into two components: ▶MPI-IO and *BAFS*. *BAFS*, a relatively thin abstract I/O interface, is layered between the ▶*MPI-IO* layer and an underlying parallel file system. It provides a Non-▶*Collective I/O* interface that functions independently for each MPI [243] process created. ▶*MPI-IO*, on the other hand, operates on the communicator level for a

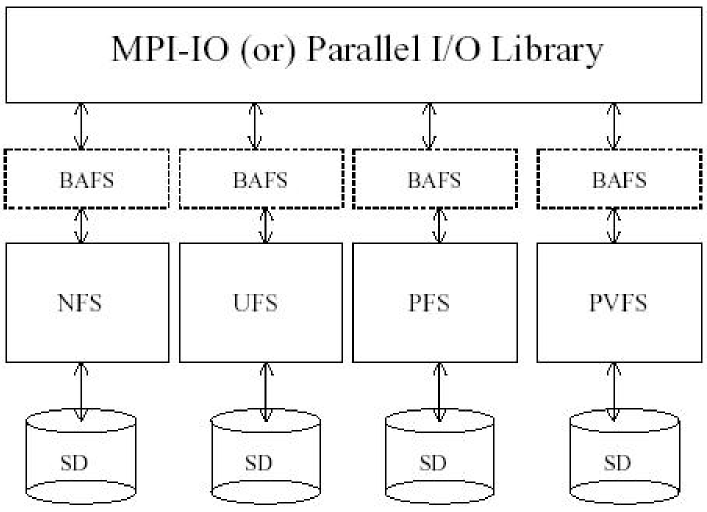


Figure 4.39: MercutIO: Non-Collective BAFS

group of processes defined by the parallel environment. The ►*MPI-IO* layer manages ►*Collective I/O* and shared file pointer operations. This layer maintains data consistency across different processes and delivers file atomicity semantics.

BAFS provides a portable high-performance point-to-point Parallel I/O interface. The *BAFS* data access routines are implemented based on non-blocking semantics in order to overlap I/O and computation. Smaller I/O requests are combined to larger chunks before the actual disk access is performed.

The current implementation of BAFS has two provisions for non-blocking I/O. The first approach makes use of the non-blocking APIs supported by the underlying file system. The second approach is based on the *producer-consumer* model. Threads are used to transfer data between the user thread and the BAFS system thread using the work queue model as depicted in Figure 4.40. In this approach, the user thread asynchronously posts I/O requests to the work queue. The system thread operates in a FIFO strategy and upon completion of an I/O request asynchronously notifies the user thread.

MFS (MOSIX File System) see ***MOSIX***

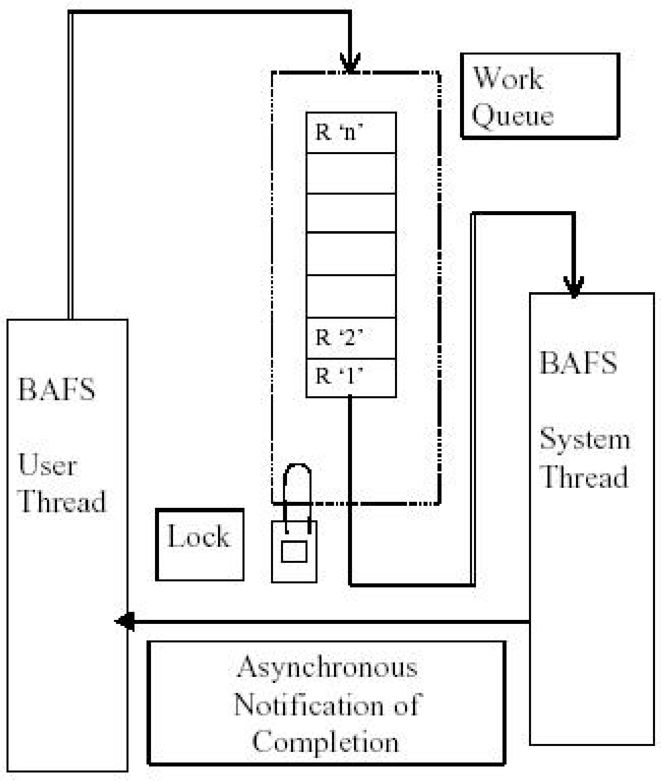


Figure 4.40: MercutIO: BAFS Model for Non-Blocking I/O

Minerva

type:

Toolkit

contact:

Alistair Veitch, `aveitch@hpl.hp.com`

description:

Minerva is an automated provisioning tool for large-scale storage systems.

motivation:

Storage systems are traditionally designed by hand, which is tedious, slow, error-prone, and frequently results in solutions that perform poorly or are over-provisioned. *Minerva* helps in the automated design of large-scale computer storage systems.

citation:

[16]

details:

The problem of system design is divided into three stages:

1. choosing the right set of storage devices for a particular application workload
2. choosing values for configuration parameters in the devices
3. mapping the user data onto the devices

The major inputs to *Minerva* are *Workload Descriptions* and *Device Descriptions*. A Workload Description contains information about the data to be stored on the system and its access patterns. Workload Descriptions contain two types if objects: *Stores* and *Streams*. Stores are logically contiguous chunks of data such as a database table or a file system, with a stated capacity. Each Store is accessed by zero or more Streams. Each *Stream* is a sequence of accesses performed on the same store. *Logical UNits (LUNs)* are sets of disks bound together using a layout such as RAID1/0 (Mirrored/Non-Redundant) or RAID5 (Block-Interleaved Distributed-Parity) [83], and addressed as a single entity by client applications. Their relationships are depicted in Figure 4.41. Hosts generate workloads, which are characterized as a set of dynamic Streams accessing static Stores. One or more Stores are mapped to each LUN.

Figure 4.42 shows the flow between each of the high-level components making up the *Minerva* system. The storage design problem is divided into three main subproblems: *Array Allocation, Array Configuration,* and *Store Assignment*. The goal of the

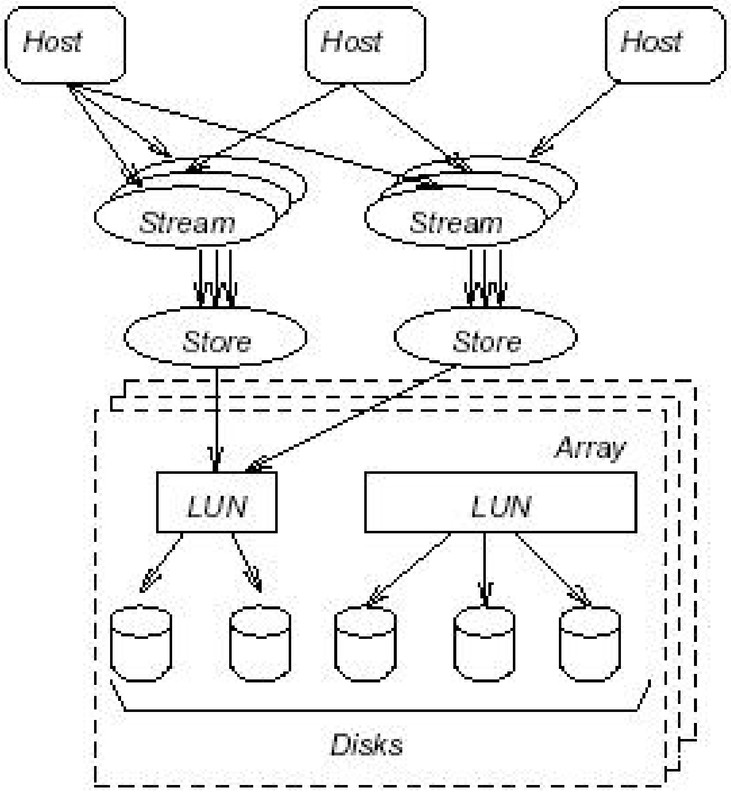


Figure 4.41: Minerva: Objects

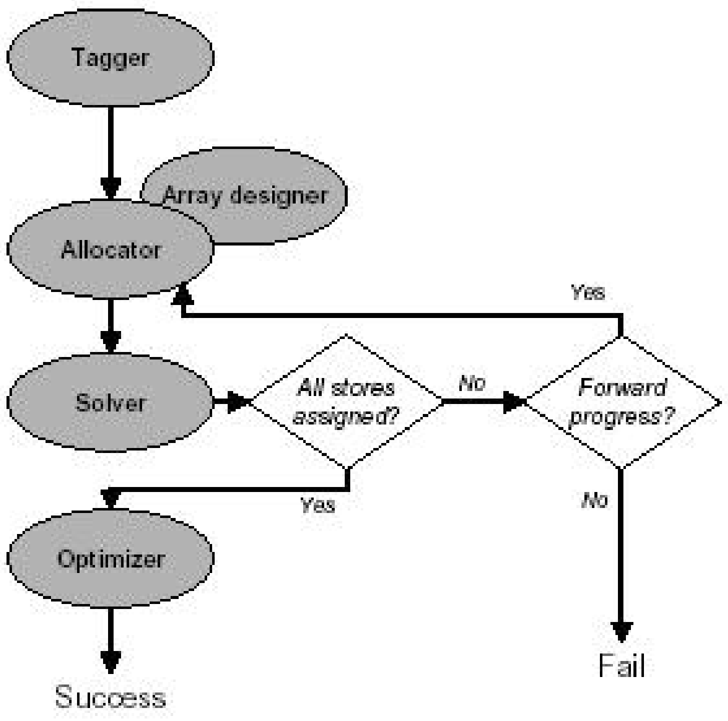


Figure 4.42: Minerva: Control Flow

Array Allocation step is to select a set of configured arrays that satisfy the resource requirements of the workload. This step is further divided into two components, the *Tagger*, which assigns a preferred RAID level to a part of the workload, and the *Allocator*, which determines how many arrays are needed. *Array Configuration* is handled by the *Array Designer*, which configures a single array at a time. The *Store Assignment* problem is handled by the *Solver*, which assigns Stores to LUNs generated by the Array Designer. The *Optimizer* prunes out unused resources and performs a reassignment that attempts to balance the load across the remaining devices. The *Evaluator* is a separate tool. It can be used to verify the correctness of the final *Minerva* solution, by applying the analytical device models.

MOCHA (Middleware Based On a Code SHipping Architecture)

type:

Filter

contact:

Manuel Rodríguez-Martínez, `manuel@cs.umd.edu`

url:

`http://www.cs.umd.edu/projects/mocha/`

description:

MOCHA is a self-extensible database middleware system that provides client applications with a uniform view and access mechanism to the data collections available in each source.

motivation:

The main purpose of *MOCHA* is to integrate collections of data sources distributed over wide-area computer networks such as the Internet.

features:

- platform independent database middleware solution built using Java and XML
- extensible execution engine allows customized data types and methods
- automatic deployment of user-defined code
- query optimization based on data movement reduction over Internet and intranets
- XML-based meta-data for system catalogs, datasets and data sources
- *light-weight* data access layer for easy installation and customization
- supports web-based thin clients
- JDBC-compliant client access interface
- on-site data filtering

related work:

►*DataCutter*, ►*dQUOB*

citation:

[293], [299]

details:

MOCHA is built around the notion that the middleware for a large-scale distributed environment should be *self-extensible*. A self-extensible middleware system is one in which new application-specific data types and query operators needed for query processing are deployed to remote sites in *automatic fashion* by the middleware system itself. In *MOCHA*, this is realized by shipping Java classes containing the new capabilities to the remote sites, where they can be used to manipulate the data of interest. All these Java classes are stored into one or more *Code Repositories* from

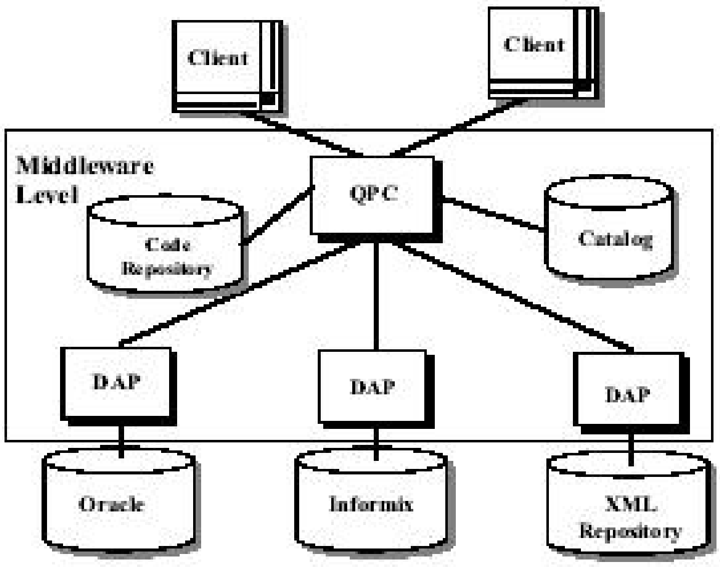


Figure 4.43: MOCHA: Architecture

which *MOCHA* later retrieves and deploys them on a *need-to-do* basis. By shipping code for query operators, such as generalized projections or predicates, *MOCHA* can generate efficient plans that place the execution of powerful *data-reducing* operators (*filters*) on the data sources. Figure 4.43 depicts the major components in the *MOCHA* Architecture. These are:

- Client Application
- Query Processing Coordinator (QPC)
- Data Access Provider (DAP)
- Data Server

MOCHA supports three kinds of *Client Applications*: 1) applets, 2) servlets, and 3) stand-alone Java applications. The *QPC* is the middle-tier component that controls the execution of all queries received from the client applications. The QPC has a *catalog* which holds all the meta-data about the user-defined types, methods, and data sites available for use by the users. The QPC also has a *Code Repository* which stores the compiled Java classes that implement the various user-defined data types and operators that are available to the user. The role of the *DAP* is to provide the QPC with an uniform access mechanism to a remote data source. The *Data Server* is the server application that stores the datasets for a particular data site.

MOPI (MOSIX Scalable Parallel Input/Output)

type:

Intelligent I/O System

contact:

Amnon Barak, `amnon@cs.huji.ac.il`

url:

`http://www.mosix.org/`

description:

MOPI enables parallel access to segments of data that are scattered among different nodes using the process migration capability of ▶*MOSIX*.

motivation:

The goal of *MOPI* is to provide Parallel I/O capabilities for large volumes of data for ▶*MOSIX* clusters.

features:

- supports partitioning of large files to independent data segments that are placed on different nodes
- support for MPI [243]

related work:

▶*MOSIX*

citation:

[17]

details:

A *MOPI* file consists of 2 parts: a *Meta Unit (MU)* and *Data Segments (DS)*, as shown in Figure 4.44. The *MU* stores the file attributes, including the number and ID of the nodes to which the file is partitioned, the size of the partition unit and the locations of the segments. Data is divided into *DS*. A DS is the smallest unit of data for I/O optimization.

The prototype *MOPI* implementation consists of three parts: a *User-Level Library* that is linked with the user application, a set of *Daemons* for managing meta-data and an optional set of *Utilities* to manage basic operations on *MOPI* files from the shell.

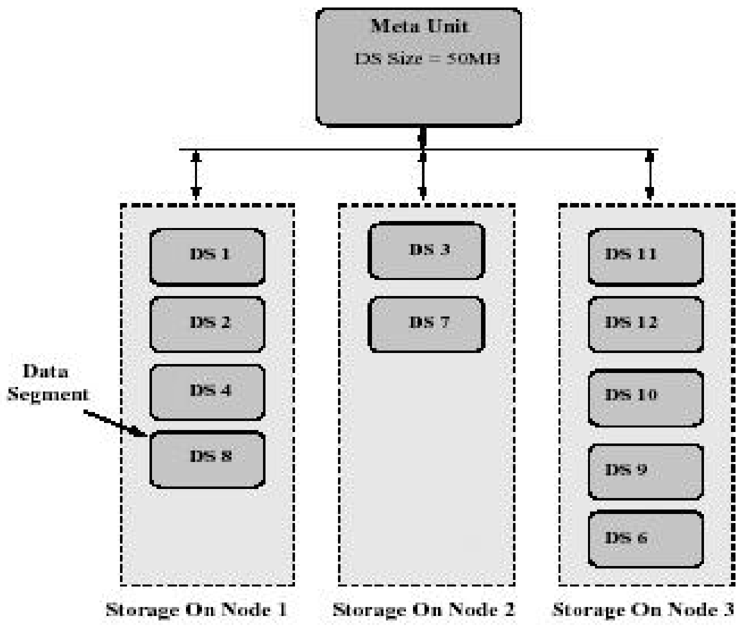


Figure 4.44: MOPI: File Structure

MOSIX

type:

Toolkit

contact:

Amnon Barak, `amnon@cs.huji.ac.il`

url:

`http://www.mosix.org/`

description:

MOSIX is a management package that can make a cluster of x86 based Linux servers and workstations (nodes) run almost like an SMP.

features:

Preemptive process migration for:

Parallel Processing automatic work distribution and redistribution

Process Migration from slower to faster nodes

Load-Balancing for even work distribution

High I/O Performance migrating intensive I/O processes to file servers

Parallel I/O migrating Parallel I/O processes from a client node to file servers

related work:

▶*MOPI*

citation:

[18], [43], [44]

details:

The *MOSIX* technology consists of two parts, both implemented at the kernel level: a *Preemptive Process Migration (PPM)* mechanism and a set of *Algorithms for Adaptive Resource Sharing*. The *PPM* can migrate any process, at any time, to any available node. Each process has a *Unique Home-Node (UHN)* where it was created. All the processes of a user's session share the execution environment of the UHN. Processes that migrate to remote nodes use local resources whenever possible, but interact with the user's environment through the UHN. When the requirements for resources exceed some threshold levels, some processes may be migrated to other nodes. Mosix has no central control or master-slave relationship between nodes.

The main *Resource Sharing Algorithms* of *MOSIX* are the *Load-Balancing* and the *Memory Ushering* [42]. The dynamic *Load-Balancing* algorithm continuously attempts to reduce the load differences between pairs of nodes, by migrating processes from higher loaded to less loaded nodes. The *Memory Ushering* (depletion prevention) algorithm is geared to place the maximal number of processes in the cluster-wide RAM, to avoid as much as possible thrashing or the swapping out of processes. The algorithm is triggered when a node starts excessive paging due to shortage of free memory. In this case the algorithm overrides the load-balancing algorithm and attempts to migrate a process to a node which has sufficient free memory.

The *Direct File System Access (DFSA)* mechanism is a re-routing switch that was designed to reduce the extra overhead of running I/O oriented system-calls of a migrated process by allowing most system-calls to run directly on the node where the process currently runs. In addition to DFSA, *MOSIX* monitors the I/O operations of each process in order to encourage a process that performs moderate to high volume of I/O to migrate to the node in which it does most of its I/O. DFSA requires a single-node file and directory consistency between processes that run on different nodes because even the same process can appear to operate from different nodes.

MOSIX File System (MFS) provides a unified view of all files on all mounted file systems in all the nodes of a *MOSIX* cluster, as if they where all within a single file system. In MFS, each node in a *MOSIX* cluster can simultaneously be a file server and run processes as clients and each process can work with any mounted file system of any type.

MPI-IO (Message Passing Interface – Input/Output)

type:

Application Level Method

url:

`http://www.mpi-forum.org`

description:

MPI-IO is the I/O chapter in MPI-2.

motivation:

The goal of *MPI-IO* is to provide a standard for describing Parallel I/O operations within an MPI-2 [243] message passing application.

features:

- allows the programmer to specify high-level information about I/O
- favors common usage patterns over obscure ones

related work:

MPI-2 [243], ►*ROMIO*

citation:

[243], [333]

details:

MPI is the de facto standard for message passing. MPI does not include one existing message passing system, but makes use of the most attractive features of them. The main advantage of the message passing standard is said to be portability and ease-of-use. MPI is intended for writing message passing programs in C and Fortran77.

MPI-2 is the product of corrections and extensions to the original MPI Standard document. Although some corrections were already made in Version 1.1 of MPI, MPI-2 includes many other additional features and substantial new types of functionality. One of the improvements is a new capability in form of Parallel I/O (*MPI-IO*).

MPI-IO should be as MPI friendly as possible. Like in MPI, a file access can be independent (no coordination between processes takes place) or collective (each process of a group associated with the communicator must participate in the collective access). What is more, MPI derived data types are used for the data layout in files and for accessing shared files. The usage of derived data types can leave holes in the file, and a process can only access data that falls under these holes. Thus, files can be distributed among parallel processes in disjoint chunks.

Since *MPI-IO* is intended as an interface that maps between data stored in memory and a file, it basically specifies how the data should be laid out in a virtual file structure rather than how the file structure is stored on disk. Another feature is that *MPI-IO* is supposed to be interrupt and thread safe.

Multi-Storage Resource Architecture

type:

Data Access System

contact:

Alok Choudhary, `choudhar@ece.nwu.edu`

description:

In the *Multi-Storage Resource Architecture*, an application can be associated with multiple storage resources that could be heterogeneously distributed over networks. These storage resources could include local disks, local databases, remote disks, remote tape systems, remote databases and so on.

motivation:

In traditional storage resource architectures an application has only one storage media available for storing the user's data. A major problem of these applications is that users have to sacrifice performance requirements in order to satisfy storage capacity requirements. Further performance improvement is impeded by the physical nature of this single storage media even if state-of-the-art I/O optimizations are employed. The *Multi-Storage Resource Architecture* was developed to solve this problem.

features:

- API for transparent management and access to various storage resources
- run-time library for each type of storage resource
- I/O performance prediction mechanism for evaluation of the application
- *PTool* to help the user automatically establish the I/O performance database that is used by the performance predictor

application:

Astro3D [356]

related work:

►*MAPFS*, ►*SRB*

citation:

[322]

details:

The architecture can be logically layered into five levels:

Physical Storage Resources At the bottom of the architecture are various storage resources including local disks, local databases, remote disks, remote databases, remote tape systems and other storage systems. They are the actual holder of data.

Native Storage Interfaces Each storage resource has its own access interface provided by the vendors of these storage systems. These interfaces to various storage systems are well established and developed by vendors. The major concerns of these native interfaces are portability, ease-of-use and reliability etc. Few of them have fully considered performance issues in a parallel and Distributed Computing environment. In addition, it is impossible for the application level users to change these interfaces directly to take care of performance issues.

Run-time Library One methodology to address the performance problem of the native storage interface is to build a run-time library that resides above it. The only concern of these libraries is performance. It captures the characteristics of user's data access pattern and performs an optimized data access method to the native storage interface.

User API This API is used in user applications to provide transparent access to various storage resources and selection of appropriate I/O optimization strategies and storage types. The API transparently consults the I/O performance database and decides the storage type and the I/O optimization approach internally according to the user's hints and meta-data information kept in database.

User Application The top layer in the logical architecture is the user application. The user writes his program by using the API and passes high-level hints. The hints are high-level since they are not concerned with low level details of storage resources and I/O optimization approaches. They only describe how the

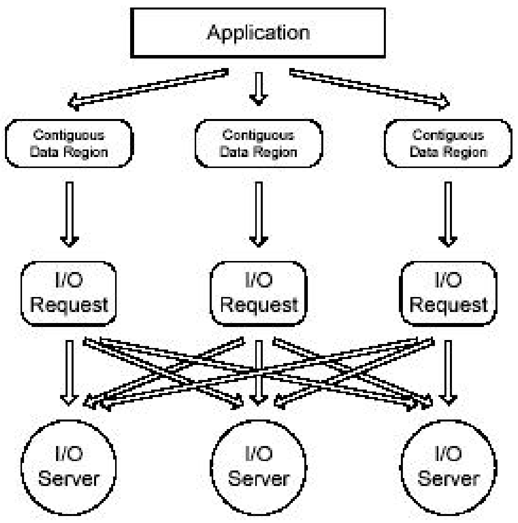


Figure 4.45: Multiple I/O: Dataflow for Noncontiguous I/O

user's dataset will be partitioned and accessed by parallel processors, how the dataset will be used in the future, what kind of storage systems the user expects to put the datasets on etc.

Multiple I/O

type:

Application Level Method

contact:

Avery Ching, `aching@ece.northwestern.edu`

description:

The interface to most parallel file systems allows for access only to a contiguous file region in a single I/O operation. Making *Multiple I/O* operations performs the required ►*Noncontiguous Data Access*, but does so with a large cost of transmitting and processing I/O requests as well as many potential disk accesses because each noncontiguous data region requires a separate I/O request that the I/O servers must process (see Figure 4.45).

related work:

►*Data Sieving*, ►*List I/O*, ►*Noncontiguous Data Access*, ►*PVFS1*

citation:

[91]

N

NeST (Network Storage)

type:

Mass Storage System

contact:

`support@nestproject.org`

url:

`http://www.cs.wisc.edu/condor/nest/`

description:

NeST is a flexible software-only storage appliance designed to meet the storage needs of the Grid.

motivation:

The primary goal is to break the false dependence between hardware and software and create software-only storage solutions that can create storage appliances out of commodity hardware. The secondary goal is to create software modules with flexible mechanisms.

features:

NeST has three key features that make it well-suited for deployment in a Grid environment:

- provides a generic data transfer architecture that supports multiple data transfer protocols (including ►*GridFTP* and ►*Chirp*), and allows for the easy addition of new protocols.
- dynamic, adapting itself on-the-fly so that it runs effectively on a wide range of hardware and software platforms.
- Grid-aware, implying that features that are necessary for integration into the Grid, such as storage space guarantees, mechanisms for resource and data discovery, user authentication, and quality of service, are a part of the *NeST* infrastructure.

related work:

Filers of Network Appliance [174], Enterprise Storage Platforms of EMC [121]

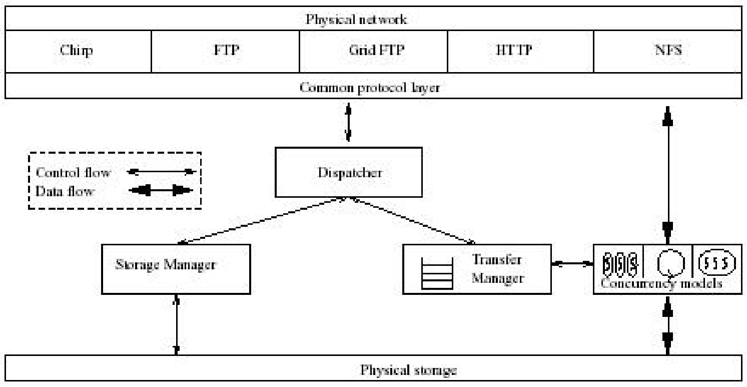


Figure 4.46: NeST: Software Design

citation:

[55], [56]

details:

As a Grid storage appliance, *NeST* provides mechanisms both for file and directory operations as well as for resource management. The implementation to provide these mechanisms is heavily dependent upon *NeST's* modular design, shown in Figure 4.46. The four major components of *NeST* are its *Protocol Layer, Dispatcher, Storage Manager* and *Transfer Manager*.

The *Protocol Layer* in *NeST* provides connectivity to the network and all client interactions are mediated through it. The *Dispatcher* is the main scheduler and macro-request router in the system and is responsible for controlling the flow of information between the other components. The *Storage Manager* has four main responsibilities: virtualizing and controlling the physical storage of the machine, directly executing non-transfer requests, implementing and enforcing access control, and managing guaranteed storage space in the form of *lots* (interfaces to guarantee storage space). The *Transfer Manager* controls data flow within *NeST*.

netCDF (Network Common Data Form)

type:

Data Format

contact:

`support@unidata.ucar.edu`

url:

`http://www.unidata.ucar.edu/packages/netcdf/`

description:

Unidata's [370] *netCDF* is a data model for array-oriented scientific data access, a package of freely available software that implements the data model, and a machine-independent data format.

features:

- sharing common data files among different applications, written in different languages, running on different computer architectures
- reduction of programming effort spent interpreting application, or machine-specific formats
- incorporation of meta-data with the data, reducing possibilities for misinterpreting the data
- accessing small subsets of large data efficiently
- making programs immune to changes caused by the addition of new variables or other additions to the data schema
- raising the level of data issues to structure and content rather than format

application:

▷*DODS*

related work:

FAN [107]

citation:

[212], [287]

details:

A *netCDF* data set consists of variables which have a name, a shape determined by its dimensions, a type, some attributes and values. Variable attributes represent ancillary information, such as units and special values used for missing data. Operations on *netCDF* components include creation, renaming, inquiring, writing, and reading. The *netCDF* format provides a platform-independent binary representation for self-describing data in a form that permits efficient access to a small subset of a large dataset, without first reading through all the preceding data. The format also allows appending data along one dimension without copying the dataset or redefining its structure.

Parallel netCDF [212] is a parallel interface for writing and reading *NetCDF* datasets. The underlying Parallel I/O is achieved through ▶*MPI-IO*, allowing for dramatic

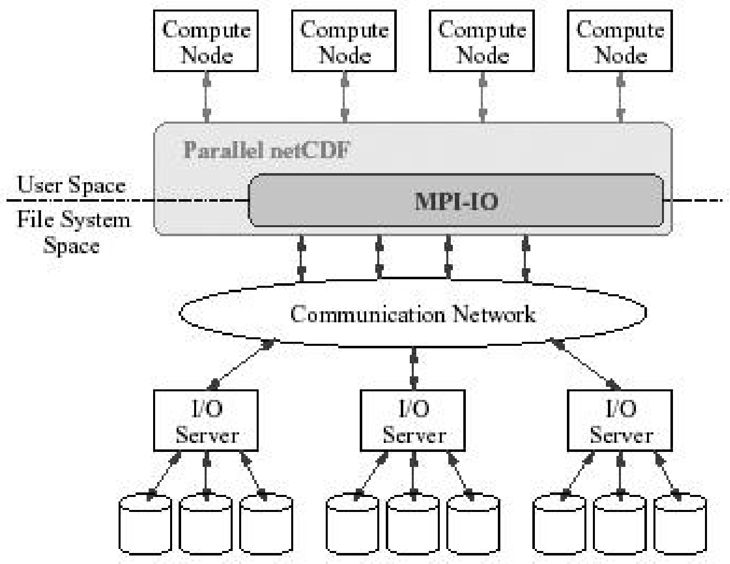


Figure 4.47: netCDF: Design of Parallel netCDF on a Parallel I/O Architecture

performance gains through the use of ►*Collective I/O* optimizations. *Parallel netCDF* runs as a library between user space and file system space. It processes *Parallel netCDF* requests from user compute nodes and, after optimization, passes the Parallel I/O requests down to ►*MPI-IO* library, and then the I/O servers receive the ►*MPI-IO* requests and perform I/O over the end storage on behalf of the user (see Figure 4.47).

NetSolve

type:

Filter

contact:

Jack Dongarra, `dongarra@cs.utk.edu`

url:

`http://icl.cs.utk.edu/netsolve/`

description:

NetSolve is a software system based on the concepts of RPC that allows for the easy access to computational resources distributed in both geography and ownership.

motivation:

Some goals of the *NetSolve* project are:

- ease-of-use for the user
- efficient use of resources
- the ability to integrate any arbitrary software component as a resource into the *NetSolve* system

features:

- uniform access to the software
- configurability
- preinstallation

application:

`http://icl.cs.utk.edu/netsolve/overview/`

related work:

▷*Globus*, NEOS [103], Ninf [253]

citation:

[30], [73], [74], [75]

details:

Figure 4.48 shows the infrastructure of the *NetSolve* system and its relation to the applications that use it. At the top tier, the *NetSolve Client Library* is linked into the user's application. The application then makes calls to *NetSolve's API* for specific services. Through the API, *NetSolve* client users gain access to aggregate resources without the necessity of user knowledge of computer networking or Distributed Computing. The *NetSolve Agent* represents the gateway to the *NetSolve* system. It serves as an information service maintaining data regarding *NetSolve Servers* and their capabilities and dynamic usage statistics. The Agent uses its resource allocation mechanism to attempt to find the server that will service the request the quickest, balance the load amongst its servers and keep track of failed servers. The Agent also implements fault-tolerant features. The *NetSolve Server* is the computational backbone of the system. It is a daemon process that awaits client requests. The Server can run on single workstations, clusters of workstations, symmetric multi-processors or machines with massively parallel processors. A key component of the *NetSolve* server is a source code generator which parses a *NetSolve Problem Description File (PDF)*. This PDF contains information that allows the *NetSolve* system to create new modules and incorporate new functionalities. In essence, the PDF defines a wrapper that *NetSolve* uses to call the function or program being incorporated.

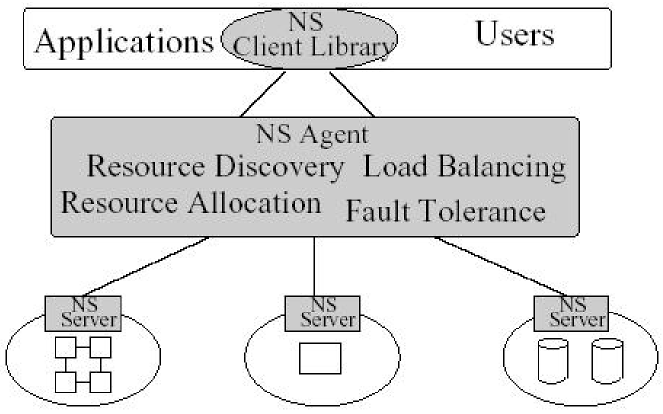


Figure 4.48: Netsolve: Architecture

Noncontiguous Data Access

type:

Application Level Method

description:

Noncontiguous Data Access is an access that works on data that is not adjacent within file, memory, or both. The various types of *Noncontiguous Data Access* are shown in Figure 4.49. One example of contiguous data in memory and noncontiguous data in file is an application that stores a two-dimensional array in a file, and then later desires to read one element from each column into a contiguous memory buffer. The more interesting of the *Noncontiguous Data Access* patterns are the ones where the file data is noncontiguous. In order to optimize access when the file data is contiguous, a memory operation can buffer the access so that data movement is executed in memory and only one file read/write request is necessary. When the file is noncontiguous, buffering alone is not adequate. Other methods must be used to perform a *Noncontiguous Data Access* when the file data is noncontiguous.

related work:

►*PVFS1* and ►*ROMIO* use ►*Data Sieving* to optimize *Noncontiguous Data Accesses*.

citation:

[91], [355]

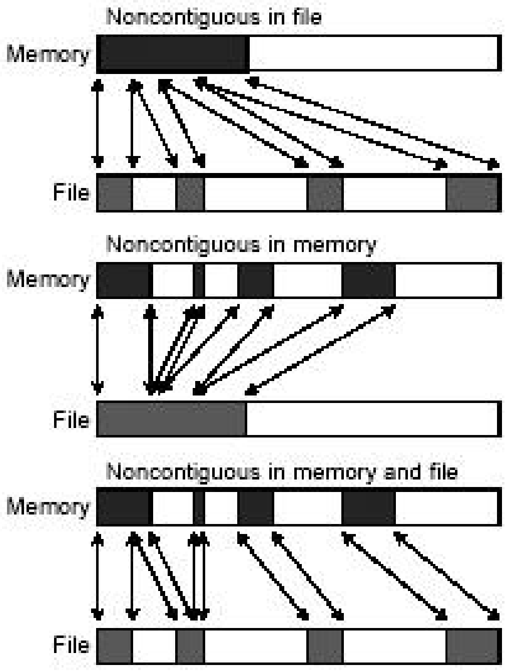


Figure 4.49: Noncontiguous Data Access: Possible Noncontiguous Data Accesses

O

Oasis+

type:

Storage System

contact:

David Watson, `dwatson@cs.ucr.edu`

url:

`www.cs.ucr.edu/~brett`

description:

The *Oasis+* distributed storage system is a reliable memory store for a small scale computing cluster. It is implemented entirely in-memory using DSM. Reliability is achieved by replication and corrective cleanup recovery actions once a failure arises.

motivation:

Oasis+ was built to operate as a backbone service for a computing cluster that supports mobile workstations or remote clients needing fast access to storage. The system acts as a storage system that can store data quickly in-memory and can do so in a dependable manner.

features:

- provides the illusion of a large virtual address space using physical memory of each of the machines on the network
- simple commodity computer network
- commodity interconnect technology
- high-availability support with configurable replication
- support for transparent failover
- high-performance distributed locking that is failure resilient
- efficient and atomic update diffs to replicas
- nodes maintain shared state so that each owner of a replica could operate in a peer-to-peer fashion

related work:

Arias [336]

citation:

[394], [395]

details:

A *Storage Region* is the basic unit of memory in *Oasis+* and is conceptually equivalent to a segment in SysV shared memory. The interface for managing Regions is modeled after standard SysV shared memory calls. The interface selected potentially simplifies the process of porting applications to *Oasis+*. An extremely general *Address Range Locking Facility* provides exclusive access to storage. Storage Regions consist of pages which are replicated for high-availability.

The protocol that *Oasis+* uses for data reliability and managing replicated copies is the *Boundary-Restricted* [358]. It guarantees robust functionality despite multiple site failures that could occur. By integrating address range locking and *Eager Release Consistency (ERC)* [209], *Oasis+* provides a flexible and efficient platform for the development of distributed services. Reliability is achieved by replication and corrective cleanup recovery actions once failures arise.

The *Address Range Lock Facility* included with *Oasis+* provides an extremely general view of the shared storage. It serves two primary purposes in the system by providing an application level primitive for mutual exclusion and, in conjunction with ERC, a high-performance method for updating shared storage.

Coupled with the need for storage system protocols is the need for base primitives that can perform updates efficiently and can manage group membership and associated topology changes in the computing cluster. For this aspect the system uses the Spread toolkit [20]. Spread provides *Atomic Multicast (ABCAST)* and group communications support to applications across local and wide-area networks. ABCAST ensures that message ordering is preserved across multi-site failures. These semantics underpin the protocols used to implement global data access and failure recovery.

OCGSA (Open Collaborative Grid Service Architecture)

type:

Standardization Effort

contact:

Kaizar Amin, `amin@mcs.anl.gov`

description:

OCGSA is a framework that builds on top of the ►*OGSA* infrastructure, providing a set of services that can be used by any collaborative application.

motivation:

OCGSA extends the notion of Grid services into the collaborative domain.

related work:

Gateway [33] ▷*Globus*, ►*OGSA*

citation:

[19]

details:

The *OCGSA* architecture composes a set of common components that can be easily customized for individual applications. The *OCGSA* framework enables users to form ad-hoc collaborative groups by interacting over a set of predefined notification topics. It provides appropriate lifetime management for individual groups, offers an advanced discovery mechanism for service instances, and establishes sophisticated security mechanisms at different levels of the application.

The *OCGSA* framework consists of the following common components for collaborative applications:

Collaborative Grid Service a Grid service with a set of predefined behaviours and meta-data elements regarding the creator of the group, name of the group, description of the group, current members of the group and level of event archiving desired

Registration Service a Grid service capable of supporting XPath/XQuery queries [408, 409]

Discovery Service Services can be discovered not only by service names but by a variety of other meta-data information.

Event Archiving Service a service that logs the messages or events communicated between online users of a group instance into a persistent database

Security Service Two levels of authorization are specified. In the *Application Level* the application service provider dictates the security policies. In the *Group Level* the group creator specifies who can join the group and the privileges of individual users within that group.

Visual Environment an interface environment that can be conveniently modified based on user preferences

OGSA (Open Grid Services Architecture)

type:

Standardization Effort

contact:

Ian Foster, `foster@mcs.anl.gov`

url:

`http://www-fp.globus.org/ogsa/`

description:

OGSA defines how a Grid functions and how Grid technologies can be implemented and applied. It enables the integration of services and resources across distributed, heterogeneous, dynamic *Virtual Organizations (VOs)* [137].

motivation:

OGSA aims to define a new common and standard architecture for Grid-based applications.

related work:

▷*GGF*, ▷*Globus*, ►*OCGSA*, ►*OGSA-DAI*, ►*OGSI*

citation:

[132], [135], [136]

details:

OGSA focuses on *Services*: computational resources, storage resources, networks, programs, databases, and the like are all represented as Services. A service-oriented view allows to address the need for standard interface definition mechanisms, local/remote transparency, adaptation to local OS services, and uniform service semantics. A service-oriented view also simplifies virtualization, that is the encapsulation behind a common interface of diverse implementations. Virtualization allows for consistent resource access across multiple heterogeneous platforms with local or remote location transparency. To express service functions in a standard form, so that any implementation of a Service is invoked in the same manner, the *Web Services Description Language (WSDL)* [92] is used.

For service interaction standard semantics are required, so that for example different Services follow the same conventions for error notification. Therefore *OGSA* defines a *Grid Service*. It is a *Web Service* [158] that provides a set of well-defined interfaces following specific conventions. The interfaces address:

Discovery Applications require mechanisms for discovering available Services and for determining the characteristics of those Services.

Dynamic service creation New service instances can be dynamically created and managed.

Lifetime management The managing of a services life-time is defined Furthermore Services can be reclaimed and their state associated with failed operations.

Notification Collections of dynamic, distributed Services are able to notify each other asynchronously of significant changes to their state.

Manageability Operations are defined to monitor and manage potentially large sets of *Grid Service Instances*.

OGSA does not address issues of implementation programming model, programming language, implementation tools, or execution environment and specifies interactions between Services in a manner independent of any hosting environment.

OGSA-DAI (OGSA - Database Access and Integration)

type:

Standardization Effort

contact:

Norman Paton, `norm@cs.man.ac.uk`

url:

`http://www.ogsadai.org.uk/`

description:

OGSA-DAI is an extension to the ▶*OGSA* architecture to support data access and integration services.

motivation:

The aim of the *OGSA-DAI* architecture is to define ▶*OGSA* generic data access and integration services, while requiring services deployed in the context of particular database or data store systems to distinguish themselves through their service data elements. *OGSA-DAI* strives to present uniform access to a wide range of data sources.

related work:

▶*OGSA*

citation:

[34], [35], [176], [268]

details:

The ▶*OGSA-DAI* project assumes an architecture that matches ▶*OGSA* and provides a simple set of composable components. The principle components and their use are illustrated in Figure 4.50. The client uses a *Data Registry* first to locate a *Grid Data Service Factory (GDSF)* service that is capable of generating the required access and integration facilities. Each Data Registry is an extension of the standard ▶*OGSA* registries that also uses meta-data about the contents, organization and operations of the data resources that may be reached. The information returned allows the client to choose an appropriate GDSF and activate it using its *Grid Service Handle (GSH)*. The client then asks that GDSF to produce a set (here one) of *Grid Data Services (GDS)* that provide the required access to data resources. They may be the data resources themselves or proxies for those data resources, as illustrated here. The client then uses the GDS to obtain a sequence of required services.

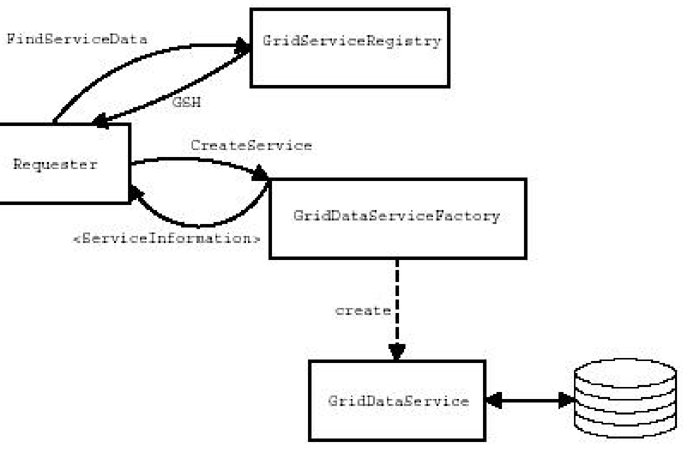


Figure 4.50: OGSA-DAI: Creating a Grid Data Service

In addition to basic database operations, such as `update, query, bulk load`, and `schema edit`, the GDS must also support a *Data Transport* specification. To enable an open-ended range of data models, operations and transport mechanisms, the required operations are requested using a *Request Document*, which specifies a sequence of activities such as database operations defined using standard query languages and data delivery. Additional categories of components required for data translation and data transport, *Grid Data Translation Services (GDTS)* and *Grid Data Transport Depots (GDTD)* are included in the *OGSA-DAI* architecture.

OGSI (Open Grid Services Infrastructure)

type:

Standardization Effort

contact:

David Snelling, `d.snelling@fle.fujitsu.com`

url:

`http://www.gridforum.org/ogsi-wg/`

description:

OGSI refers to the base infrastructure on which ►*OGSA* is built. *OGSI* is a formal and technical specification of the concepts described in ►*OGSA*.

motivation:

OGSI provides a full specification of the behaviours and *Web Services Description Language (WSDL)* [92] interfaces that define a Grid service.

related work:

▷*GGF*, ▷*Globus*, ▶*OCGSA*

citation:

[132], [135], [136], [369]

details:

The OGSA Service Model `OGSA` represents everything as a *Grid Service*: a *Web Service* [158] that conforms to a set of conventions and supports standard interfaces for such purposes as lifetime management. Grid Services are characterized by the capabilities that they offer. A Grid Service implements one or more interfaces, where each interface defines a set of operations that are invoked by exchanging a defined sequence of messages. The term *Grid Service Instance* is used to refer to a particular instantiation of a Grid Service. Every Grid Service Instance is assigned a globally unique name, the *Grid Service Handle (GSH)*, that distinguishes a specific Grid Service Instance from all other Grid Service Instances that have existed, exist now, or will exist in the future. Protocol- or instance-specific information such as network address and supported protocol bindings are encapsulated into a single abstraction called a *Grid Service Reference (GSR)*. Unlike a GSH, which is invariant, the GSR(s) for a Grid Service Instance can change over that service's lifetime.

Creating Transient Services – Factories `OGSA` defines a class of Grid Services that implements an interface that creates new Grid Service Instances. This is called the *Factory Interface* and a service that implements this interface a *Factory*. The Factory Interface's `CreateService` operation creates a requested Grid Service and returns the GSH and initial GSR for the new Service Instance.

Service Lifetime Management Grid Services Instances are created with a specified lifetime. *Soft State Lifetime Management* is implemented using the operation `SetTerminationTime` within the required *GridService Interface*, which defines operations for *negotiating an initial lifetime* for a new Service Instance, for *requesting a lifetime extension*, and for harvesting a Service Instance when its lifetime has expired.

Managing Handles and References The approach taken in ▶*OGSA* is to define a handle-to-reference mapper interface (*HandleMap*). The operations provided by this interface take a GSH and return a valid GSR. *Service Data* and *Service Discovery* associated with each Grid Service Instance is a collection of XML elements encapsulated as Service Data elements.

Notification The ►*OGSA* notification framework allows clients to register interest in being notified of particular messages (the *NotificationSource Interface*) and supports asynchronous, one-way delivery of such notifications (*Notification-Sink*).

Change Management In order to support *Discovery* and *Change Management* of Grid Services, Grid Service Interfaces must be globally and uniquely named.

Out-of-Core see ***EM Algorithms and Data Structures***

P

Pablo

type:

Other / Project

contact:

Daniel A. Reed, `reed@cs.uiuc.edu`

url:

`http://www-pablo.cs.uiuc.edu/index.htm`

description:

The *Pablo* project conducts research on performance analysis and visualization techniques for HPC systems, including experimental performance measurement and adaptive control, characterization of I/O patterns, flexible parallel file systems, collaborative virtual environments for performance analysis and software history analysis, and WWW server performance.

Key research foci concerning I/O are:

Performance Analysis techniques and scalable parallel file systems for:

- use in Grid environments to optimize application and run-time behaviour during program execution using real-time adaptive systems for resource policy control
- use in conventional HPC environments using tools that capture and communicate performance metrics

I/O system design, analysis, and optimization tools & techniques for large-scale commodity and workstation clusters, peta-scale systems, and distributed Computational Grids. This work includes extending, documenting, archiving, and disseminating tools, sample applications, and experimental data through the group's I/O tool and data repository, the national *CADRE* [70] facility.

Among the many projects of *Pablo* the ones which deal mainly with I/O are:

PPFS II ▶*PPFS II* is a next generation parallel file system with real-time control and adaptive policy control capabilities.

I/O Characterization Rapid increases in computing and communication performance are exacerbating the long-standing problem of performance-limiting I/O. Currently, work is underway to measure the impact on performance of I/O operations in areas like Unix I/O Characterization, HDF Analysis, and ►*MPI-I/O* Instrumentation. For measuring the I/O performance on different platforms the group also developed an I/O Benchmark Software Distribution.

National Facility for I/O Characterization and Optimization (CADRE) [70] It is extending, documenting, archiving, and disseminating tools, sample applications, and experimental data to stimulate education and research on I/O system design, analysis, and optimization for large-scale commodity and workstation clusters, peta-scale systems, and distributed computational Grids.

Panda

type:

I/O Library

contact:

Marianne Winslett, `winslett@uiuc.edu`

url:

`http://dais.cs.uiuc.edu/panda/`

description:

Panda is a library for input and output of multidimensional arrays on parallel and sequential platforms.

motivation:

Panda aims to provide simpler, more abstract interfaces to application programmers, to produce more advanced I/O libraries supporting efficient layout alternatives for multidimensional arrays on disk and in main memory, and to support high-performance array I/O operations.

features:

►*Collective I/O*, ►*Server-Directed I/O*

application:

Center for the Simulation of Advanced Rockets [101]

related work:

►*LEDA-SM*, ►*TPIE*

citation:

[65], [84], [85], [86], [87], [312], [313], [314]

details:

The system architecture of *Panda* is shown in Figure 4.51, which shows how *Panda* is distributed across compute nodes (*Panda Clients*) and I/O nodes (*Panda Servers*). Under ▶*Server-Directed I/O*, the application program running on the compute nodes communicates with the client via *Panda*'s high-level ▶*Collective I/O* interface for multidimensional arrays. When an application makes a ▶*Collective I/O* request to *Panda*, a selected client (the *Master Client*) sends to a selected server (the *Master Server*) a short high-level description of the two schemas for the arrays, in memory and on disk. After the initial request to the servers for a ▶*Collective I/O* operation, *Panda's Servers* then make data requests for the clients. The Master Server then informs all the other servers of the schema information, and each server plans how it will request or send its chunks of the array data to or from the relevant clients. Upon completion of a ▶*Collective I/O* request, the servers inform the master server who informs the Master Client that the ▶*Collective I/O* operation is completed. The Master Client in turn informs the other clients.

Parallel Data Compression

type:

Data Access System

contact:

Jonghyun Lee, `jlee17@uiuc.edu`

description:

Three approaches for parallel compression of scientific data for migration are proposed.

motivation:

The goal is to incorporate compression into migration in a way that reliably reduces application turnaround time on parallel platforms.

related work:

▶*Active Buffering*, ▶*GASS*, ▶*GridFTP*, ▶*RIO*

citation:

[207], [208]

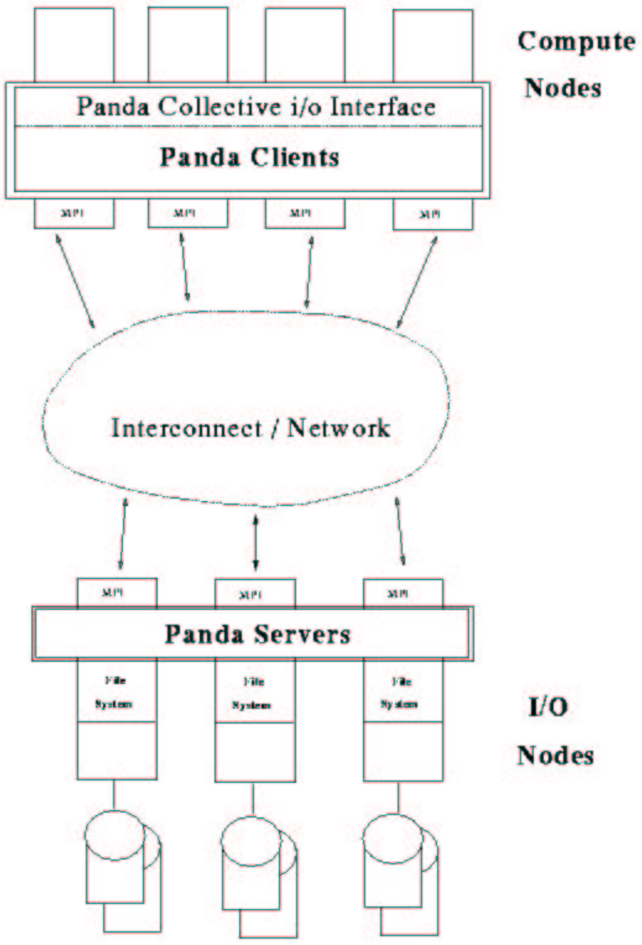


Figure 4.51: Panda: System Architecture

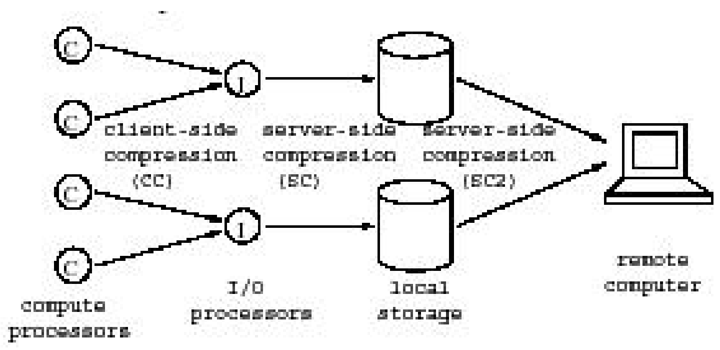

client-side compression (CC)
server-side compression (SC)
server-side compression (SC2)
compute processors
I/O processors
local storage
remote computer

Figure 4.52: Parallel Data Compression: Data Flow with Simulation/Migration

details:

I/O servers are used to stage the output to the local file system before migration. Figure 4.52 shows the data flow in a simulation run with I/O and migration, along with three possible spots for performing compression. This *local staging* prevents compute processors from stalling while data is migrated.

Client-side Compression during an I/O Phase (CC) Each client compresses each of its output chunks and sends them to a server, along with meta-data such as the compressed chunk size and the compression method. I/O servers receive compressed chunks from clients and stage them to disks.

Server-side Compression during an I/O Phase (SC) With SC, servers receive output data from clients during an I/O phase, compress them, and stage them to disk. SC allows the array to be reorganized.

Server-side Compression on Already-Staged Outputs (SC2) Before being transferred to a remote machine, a staged output needs to be read into memory from the local file system. SC2 reads and compresses the staged output, and then migrates it.

Parallel NetCDF see ***NetCDF***

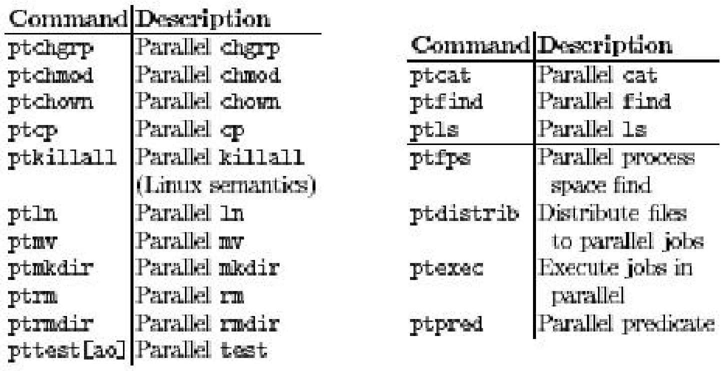
Parallel Unix Commands

type:

Toolkit

contact:

William D. Gropp, `gropp@mcs.anl.gov`



Command	Description
`ptchgrp`	Parallel `chgrp`
`ptchmod`	Parallel `chmod`
`ptchown`	Parallel `chown`
`ptcp`	Parallel `cp`
`ptkillall`	Parallel `killall` (Linux semantics)
`ptln`	Parallel `ln`
`ptmv`	Parallel `mv`
`ptmkdir`	Parallel `mkdir`
`ptrm`	Parallel `rm`
`ptrmdir`	Parallel `rmdir`
`pttest[ao]`	Parallel `test`

Command	Description
`ptcat`	Parallel `cat`
`ptfind`	Parallel `find`
`ptls`	Parallel `ls`
`ptfps`	Parallel process space find
`ptdistrib`	Distribute files to parallel jobs
`ptexec`	Execute jobs in parallel
`ptpred`	Parallel predicate

Table 4.2: Parallel Unix Commands: Commands

description:

The *Parallel Unix Commands* are a family of MPI [243] applications and are natural parallel versions of common Unix user commands.

features:

- familiar to Unix users
- interact well with other Unix tools
- run at interactive speeds like traditional Unix commands

related work:

►*Java I/O*

citation:

[265]

details:

The *Parallel Unix Commands* are shown in Table 4.2. They are of three types:

1. straightforward parallel versions of traditional commands with little or no output
2. parallel versions of traditional commands with specifically formatted output
3. new commands in the spirit of the traditional commands

All of the commands have a host argument as an (optional) first argument.

Each command parses its hostlist arguments and then starts an MPI program (with `mpirun` or `mpiexec`) on the appropriate set of hosts.

Parrot

type:

File System

contact:

Douglas Thain, `thain@cs.wisc.edu`

url:

`http://www.cs.wisc.edu/~thain/research/parrot/`

description:

Parrot is the successor of the *Plugable File System (PFS)*. It is an *Interposition Agent*, which is able to perform POSIX-like I/O on remote data services. An Interposition Agent is a middleware that adapts standard interfaces to new distributed systems. Interposition Techniques are summarized in [347, page 3].

motivation:

Applications require middleware that can integrate new distributed systems.

related work:

▶*Chirp*, ▷*Condor*, ▶*NeST*

citation:

[347]

details:

An application may be written or modified to invoke the library directly, or it may be attached via various *Interposition Techniques* [347, page 3]. Like an OS, a series of device drivers give access to remote I/O systems, but *Parrot* does not know the structure of remote devices at higher levels.

I/O operations can be classified into two categories: operations on file pointers (eg.: `read, write`) and operations on file names (eg.: `rename, stat`). The former check the validity of the arguments, and then descend the various data structures. The latter commands first pass through the *name resolver*, which may transform the program supplied names according to a variety of name(s) and systems.

In the simplest case of name resolution, *Parrot* operates on the name unchanged. A *mount list* driver makes use of a simple file that maps logical names and directories to remote files names. Alternatively, the ▶*Chirp* I/O driver can be used.

Most POSIX applications access files through explicit operations such as `read` and `write`. However, files may also be *memory-mapped* [347, page 7]. Memory-mapped files are supported in two ways, depending on the Interposition Method in use. If *Parrot* is attached via an *internal technique*, then memory-mapped files may be supported by simply allocating memory with `malloc` and loading the necessary data into memory by invoking the device driver. If *Parrot* is attached via an *external technique*, then the entire file is loaded into the I/O channel, and the application is redirected to `mmap` that portion of the channel.

The default buffering discipline performs fine-grained partial file operations on remote services to access the minimal amount of data to satisfy an application's immediate reads and writes. *Parrot* may also perform whole-file staging and caching upon first `open`.

PASSION (Parallel And Scalable Software for Input-Output)

type:

I/O Library

contact:

Rajeev Thakur, `thakur@mcs.anl.gov`

description:

PASSION is a compiler and run-time support system which provides support for compiling *out-of-core* data parallel programs (►*EM Algorithms and Data Structures*), Parallel I/O of data, communication of out-of-core data, redistribution of data stored on disks, many optimizations including ►*Prefetching*, ►*Data Sieving*, as well as support at the OS level.

motivation:

The goal of the project is to provide software support for high-performance Parallel I/O at the compiler, run-time and file system levels.

features:

- support at the compiler, run-time and OS level
- I/O optimizations transparent to users
- various optimization techniques like ►*Collective I/O*, ►*Prefetching* and ►*Data Sieving* for reducing I/O costs

related work:

►*TPIE*

citation:

[349], [351]

details:

PASSION has a layered approach. The *compiler* translates out-of-core HPF programs to message passing node programs with explicit Parallel I/O. It extracts information from user directives about the data distribution, which is required by the run-time system. It restructures loops having out-of-core arrays and also decides the transformations on out-of-core data to map the distribution on disks with the usage in the loops. It also embeds calls to appropriate *PASSION* run-time routines which carry out I/O efficiently. The *Compiler* and *Run-time Layers* pass data distribution and access pattern information to the *Two-Phase Access Manager* and the *Prefetch Manager*. They optimize I/O using buffering, redistribution and ►*Prefetching* strategies.

The *PASSION* run-time support system makes I/O optimizations transparent to users. The run-time procedures can either be used together with a compiler to translate out-of-core data parallel programs, or used directly by application programmers. Among other optimizations it hides disk data distribution from the user, reorders I/O requests to minimize seek time and prefetches disk data to hide I/O latency and performs ►*Collective I/O*.

The run-time library routines can be divided into four main categories based on their functionality:

Array Management/Access Routines handle the movement of data between in-core and out-of-core arrays

Communication Routines perform collective communication of data in the out-of-core array

Mapping Routines perform data and processor/disk mappings

Generic Routines perform computations on out-of-core arrays

For optimizing I/O *PASSION* uses ►*Two-Phase I/O*, ►*Prefetching* and ►*Data Sieving*.

PC-OPT see ***Prefetching***

PDM (Parallel Disk Model)

type:

Definition

contact:

Jeffrey Scott Vitter, `jsv@purdue.edu`

description:

PDM provides an elegant and reasonably accurate model for analyzing the performance of ▶*EM (External Memory) Algorithms and Data Structures*. It is a model for developing optimal algorithms for two-level storage systems. The main properties of magnetic disks and multiple disk systems are:

N = problem size (in units of data items),
M = internal memory size (in units of data items),
B = block transfer size (in units of data items),
D = number of independent disk drives,
P = number of CPUs,
Q = number of input queries (for a batched problem), and
Z = query output size (in units of data items),

where $M < N$, and $1 \leq DB \leq M/2$.

$n = N/B$, $m = M/B$, $q = Q/B$, $z = Z/B$ to be the problem input size, internal memory size, query specification size, and query output size, respectively, in units of disk blocks.

Figure 4.53 (a) shows the model for $P = 1$, in which the D disks are connected to a common CPU and (b) for $P = D$, in which each of the D disks is connected to a separate processor.

The primary measures of performance in *PDM* are:

- the number of I/O operations performed
- the amount of disk space used
- the internal (sequential or parallel) computation time

The *PDM* model can be generalized to the *Hierarchical Memory Model* that ranges from registers at the small end to tertiary storage at the large end.

related work:

▶*EM Algorithms and Data Structures*

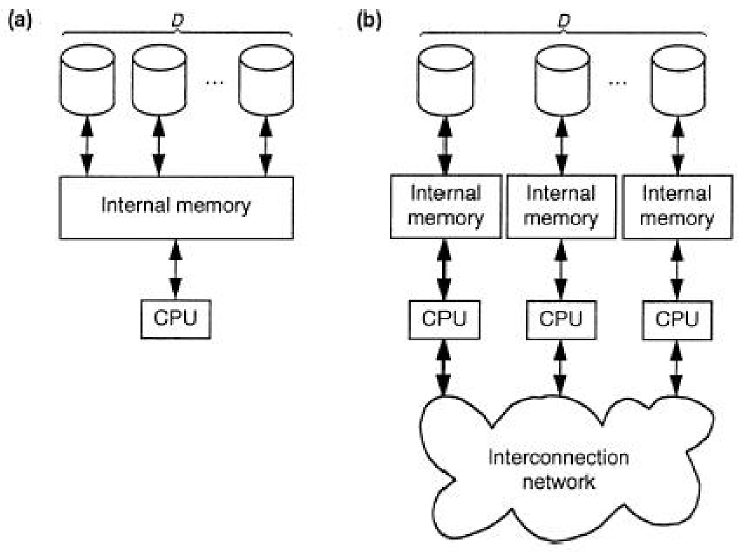


Figure 4.53: PDM: Two Configurations

citation:
: [380], [381]

Persistent Archives

type:
: Mass Storage System

contact:
: Reagan Moore, `moore@sdsc.edu`

url:
: `http://www.sdsc.edu/NARA`

description:
: The *Persistent Archive* Research Group of the ▷*GGF* promotes the development of an architecture for the construction of *Persistent Archives*. The term *persistent* is applied to the concept of an archive to represent the management of the evolution of the software and hardware infrastructure over time.

motivation:
: The project is a multi-year effort aimed at demonstrating the viability of the use of Data Grid technology to automate all of the archival processes. The ultimate goal of the prototype *Persistent Archive* is to identify the key technologies that facilitate the creation of a *Persistent Archive* of archival objects.

related work:

▷*GGF*, ▶*SRB*, ▷*VDG*

citation:

[235], [236], [237], [238], [258], [282]

details:

A *Persistent Archive* provides the mechanisms needed to manage technology evolution while preserving *Digital Entities* and their context. During the lifetime of the *Persistent Archive*, each software and hardware component may be upgraded multiple times. The challenge is creating an architecture that maintains the authenticity of the archived documents while minimizing the effort needed to incorporate new technology. *Persistent Archives* can be based on ▶*VDGs* and therefore the *Persistent Archive* research group is examining how *Persistent Archives* can be built from ▶*VDGs*.

Data Grids provide a *Logical Name Space* into which Digital Entities can be registered. The Logical Name Space is used to support global, persistent identifiers for each Digital Entity within the context of each *Archival Collection*. The Digital Entities are represented by their *Logical Name*, a *Physical File Name*, and, if desired, an *Object Identifier* that is unique across Archival Collections. Data Grids map distributed state information onto the Logical Name Space for each Grid service.

A *Persistent Archive* manages Archival Collections of Digital Entities. The Archival Collection itself can be thought of as a derived data product that results from the application of *Archival Processes* to a group of constituent documents. The Archival Processes generate the descriptive, provenance, and authenticity meta-data. The Archival Collection is used to provide a context for the Digital Entities that are stored in the archive. Discovery of an individual Digital Entity within an Archival Collection is accomplished by querying the descriptive meta-data.

Four basic categories of extended transport operations are collectively required by Data Grids, digital libraries, and *Persistent Archives*.

- Byte-Level Access
- Latency Management Mechanisms
- Object-Oriented Access
- Heterogeneous System Access

Byte-Level Access transport operations correspond to the standard operations supported by Unix file systems. Latency Management transport operations are typically operations that facilitate bulk data and bulk meta-data access and update. Object-oriented Transport operations provide the ability to process the data directly at the remote storage system. Transport operations related to access of heterogeneous systems typically involve protocol conversion and data repackaging.

To discover relevant documents, transport them from storage to the user, and interact with storage systems for document retrieval, ▶*VDGs* provide the following methods:

Derived Data Product Access ▶*VDGs* provide direct access to the derived data product when it exists. This implies the ability to store information about the derived data products within a Collection that can be queried. A similar capability, implemented as a finding aid, is used to characterize the multiple data Collections and contained data entities that are stored in a *Persistent Archive*. The finding aid can be used to decide which archived Collection to instantiate if the Collection is not already on-line.

Data Transport Data Grids provide transport mechanisms for accessing data in a distributed environment that spans multiple administration domains. Data Grids also provide multiple roles for characterizing the allowed operations on the stored data, independently of the underlying storage systems.

Storage Repository Abstraction Data Grids provide the mechanisms needed to support distributed data access across heterogeneous data resources. Data Grids implement servers that map from the protocols expected by each proprietary storage repository to the storage repository abstraction. This makes it possible to access Digital Entities through a standard interface, no matter where it is stored.

The architecture of the *Persistent Archive* at SDSC is based on commercially available software systems and application-level software developed at SDSC. The components include an archival storage system (IBM's ▶*HPSS*), a data handling system (SDSC's ▶*SRB*), an object relational database (Oracle), collection management software (SDSC's *Meta Information Catalog (MCAT)* [225], collection instantiation software (SDSC scripts), collection ingestion software (SDSC scripts), hierarchical data model XML DTD, relational data model ANSI SQL Data Definition Language, and DTD manipulation software (UCSD *XML Matching and Structuring Language (XMAS))*.

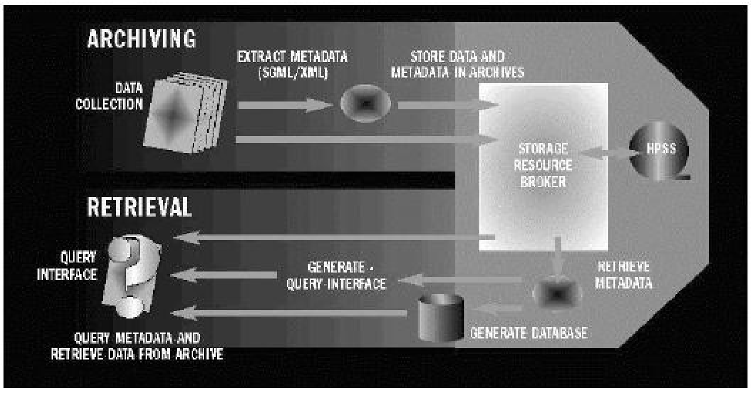


Figure 4.54: Persistent Archive: Persistent Collection Process

Each Collection is implemented as a separate Data Grid, controlling and managing the Digital Entities housed within that site. Digital Entities will be cross-registered between the Collections, and replicated onto resources at other sites to ensure preservation of both descriptive meta-data and the Digital Entities. The infrastructure will contain mechanisms for data ingestion, persistent storage, and meta-data management. The data ingestion tools will manage anomaly processing of digital objects and their associated XML meta-data.

Petal

type:

Mass Storage System

contact:

Chandramohan A. Thekkath, `thekkath@acm.org`

url:

`http://research.compaq.com/SRC/articles/199811/petal/`

description:

Petal is a distributed block-level storage system that tolerates and recovers from any single component failure, dynamically balances load between servers, and transparently expands in performance and capacity.

motivation:

The principal goal has been to design a storage system for heterogeneous environments that is easy to manage and that can scale gracefully in capacity and performance without significantly increasing the cost of managing the system.

features:

- tolerates and recovers from any single component failure, such as disk, server or network
- can be geographically distributed to tolerate site failures such as power outages and natural disasters
- transparently reconfigures to expand in performance and capacity as new servers and disks are added
- uniformly balances load and capacity throughout the servers in the system
- provides fast, efficient support for backup and recovery in environments with multiple types of clients, such as file servers and databases
- I/O rates of up to 3150 requests/sec and bandwidth up to 43,1 MB/sec

related work:

►*Frangipani*

citation:

[206]

details:

As shown in Figure 4.55, *Petal* consists of a pool of distributed storage servers that cooperatively implement a single, block-level storage system. Clients view the storage system as a collection of virtual disks and access *Petal* services via a RPC interface. A basic principle in the design of the *Petal RPC Interface* was to maintain all state needed for ensuring the integrity of the storage system in the servers, and maintain only hints in the clients. Clients maintain only a small amount of high-level mapping information that is used to route read and write requests to the most appropriate server. If a request is sent to an inappropriate server, the server returns an error code, causing the client to update its hints and retry the request.

The server software consists of the following modules:

Liveness Module This module ensures that all servers in the system will agree on the operational status, whether running or crashed, of each other. This service is used by the other modules, notably the Global State Manager, to guarantee continuous, consistent operation of the system as a whole in the face of server and communication failures.

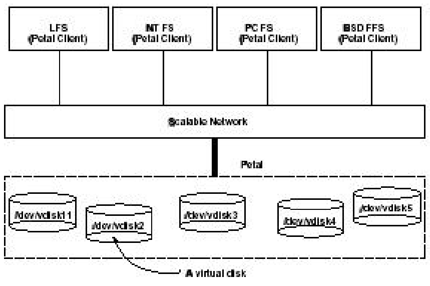


Figure 4.55: Petal: Physical View

Global State Manager *Petal* maintains information that describes the current members of the storage system and the currently supported virtual disks. This information is replicated across all *Petal* servers in the system. The Global State Manager is responsible for consistently maintaining this information.

Data Access and Recovery Modules These modules control how client data is distributed and stored in the *Petal* storage system. A different set of data access and recovery modules exists for each type of redundancy scheme supported by the system.

Virtual to Physical Translation Module This module contains common routines used by the various data access and recovery modules. These routines translate the virtual disk offsets to physical disk addresses.

PFS see ***Parrot***
PHASE-LRU see ***Prefetching***
Pipelined SRB see ***SRB***

PPFS II (Portable Parallel File System)

type:

Intelligent I/O System

contact:

Daniel A. Reed, `reed@cs.uiuc.edu`

url:

`http://www-pablo.cs.uiuc.edu/Project/PPFS/History/NEWPPFSOverview.htm`

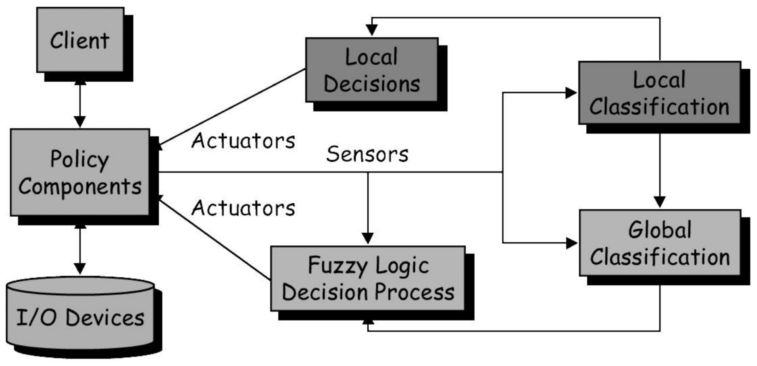


Figure 4.56: PPFS II: Real-Time Control Component Architecture

description:

PPFS II is a portable parallel file system with real-time control and adaptive policy control capabilities.

motivation:

PPFS II evolved from a desire to explore the design space for scalable I/O systems.

features:

adaptive performance optimization

related work:

▶*Pablo*

citation:

[119], [120], [178]

details:

PPFS II utilizes the *Autopilot* [38] *Real-Time Adaptive Resource Control Library* and the Nexus [250] distributed computing infrastructure. Autopilot provides a flexible set of performance sensors, decision procedures, and policy actuators to realize adaptive control of applications and resource management policies on both parallel and wide-area distributed systems. Figure 4.56 illustrates the real-time control component architecture of *PPFS II*. Qualitative application behaviour is captured by user *hints* and *access pattern classification. Adaptive policy selection* is achieved with a set of *fuzzy logic rules*.

Prefetching

type:

Access Anticipation Method

contact:

Rajeev Thakur, `thakur@mcs.anl.gov`

description:

The time taken by a program can be reduced if it is possible to overlap computation with I/O in some fashion. A simple way of achieving this is to issue an asynchronous I/O request for the next data immediately after the current data has been read. This is called Data *Prefetching*. Since the read request is asynchronous, the reading of the next data can be overlapped with the computation being performed on the current data. If the computation time is comparable to the I/O time, this can result in significant performance improvements.

In [223] the authors concentrate on the patterns of file access and how *Prefetching* policies can be tuned to optimize.

Applying the techniques of *Prefetching* and caching to Parallel I/O systems is fundamentally different from that in systems with a single disk. Intuitively appealing greedy prefetching policies that are suitable for single-disk systems can have poor worst-case and average-case performance for multiple disks. Similarly, traditional caching strategies do not account for parallelism in the I/O accesses and, consequently, may not perform well in a multiple-disk system. *PC-OPT* [191] was developed to optimize *Prefetching* and caching in the parallel disk model (▶*PDM*).

PHASE-LRU [190] is another simple *Prefetching* and caching algorithm based on ▶*PDM* using bounded look-ahead.

related work:

Lazy Prefetching see ▶*Greedy Writing*, PC-OPT [191], PHASE-LRU [190]

citation:

[349]

prudentPrefetching see ***greedyWriting***

PVFS1 (Parallel Virtual File System)

type:

File System

contact:

Rob Ross, `rross@mcs.anl.gov`

url:

`http://www.pvfs.org`

description:

PVFS1 is a scalable parallel file system for PC clusters. Its priority is placed on providing high performance for parallel scientific applications.

motivation:

The goal of the *PVFS1* project is to explore the design, implementation and uses of Parallel I/O.

features:

- compatibility with existing binaries
- ease of installation
- user-controlled striping of files across nodes
- multiple interfaces, including a ▶*MPI-IO* interface via ▶*ROMIO*
- utilizes commodity network and storage hardware

related work:

▶*PVFS2*

citation:

[71], [72]

details:

There are four major components to the *PVFS1* system:

- Metadata Server (mgr)
- I/O Server (iod)
- PVFS1 Native API (libpvfs)
- PVFS1 Linux Kernel Support

The first two components are daemons which run on nodes in the cluster. The Meta-data Server, named *mgr*, manages all file meta-data for *PVFS1* files. The second daemon is the I/O Server, or *iod*. The I/O Server handles storing and retrieving of file data stored on local disks connected to the node. the *PVFS1* Native API provides user-space access to the *PVFS1* servers. This library handles the scatter/gather operations necessary to move data between user buffers and *PVFS1* servers, keeping these operations transparent to the user. Finally the *PVFS1* Linux Kernel Support provides the functionality necessary to mount *PVFS1* file systems on Linux nodes. This allows existing programs to access *PVFS1* files without any modification.

PVFS2 (Parallel Virtual File System)

type:

File System

contact:

Rob Ross, `rross@mcs.anl.gov`

url:

`http://www.pvfs.org`

description:

PVFS2 is a parallel file system for Linux clusters.

motivation:

PVFS2 is being implemented to overcome the shortcomings of *PVFS1*, which is too socket-centric, too obviously single-threaded, won't support heterogeneous systems with different endian-ness, and relies too thoroughly on OS buffering and file system characteristics.

features:

- modular networking and storage subsystems
- powerful request format for structured ▶*Noncontiguous Data Accesses*
- flexible and extensible data distribution modules
- distributed meta-data
- stateless servers and clients (no locking subsystem)
- explicit concurrency support
- tunable semantics

- flexible mapping from file references to servers
- tight ▶*MPI-IO* integration
- support for data and meta-data redundancy

related work:

▶*PVFS1*

citation:

[294]

details:

A *PVFS2* file system may consist of the following pieces (some are optional): the *pvfs2-Server, System Interface, Management Interface, Linux Kernel Driver, pvfs2-Client*, and *ROMIO PVFS2 Device*.

Unlike ▶*PVFS1*,which has two server processes (*mgrs*, *iods*), in *PVFS 2* there is exactly one type of server process, the *pvfs2-Server*. A configuration file tells each pvfs2-Server what its role will be as part of the parallel file system. There are two possible roles, *I/O Server* and *Metadata Server*, and any given pvfs2-Server can fill one or both of these roles. I/O servers store the actual data associated with each file, typically striped across multiple servers in round-robin fashion. Metadata Servers store meta information about files, such as permissions, time stamps, and distribution parameters. Metadata Servers also store the directory hierarchy.

The *System Interface* is the lowest level user space API that provides access to the *PVFS2* file system. It is implemented as a single library, called *libpvfs2*. The System Interface API does not map directly to POSIX functions. In particular, it is a stateless API that has no concept of `open(), close()`, or file descriptors. This API does, however, abstracts the task of communicating with many servers concurrently.

The *Management Interface* is a supplemental API that adds functionality that is normally not exposed to any file system users.

The *Linux Kernel Driver* is a module that can be loaded into an unmodified Linux kernel in order to provide *Virtual File System (VFS)* support for *PVFS2*.

The *ROMIO PVFS2 Device* is a component of the ▶*ROMIO* ▶*MPI-IO* implementation that provides ▶*MPI-IO* support for *PVFS2*.

An arbitrary number of different network types can be supported through an abstraction known as the *Buffered Messaging Interface (BMI)*.

R

RAID-x

type:

Storage System

contact:

Kai Hwang, `kaihwang@usc.edu`

description:

RAID-x is a new orthogonal architecture for building distributed disk arrays in multicomputer clusters.

motivation:

The purpose of *RAID-x* is to achieve high I/O performance in scalable cluster computing.

features:

- single address space
- high-availability support
- performance and size scalability
- high-compatibility with cluster applications

related work:

▶*Petal*, RAID [83]

citation:

[181]

details:

In Figure 4.57 the orthogonal mapping of data blocks and their images in *RAID-x* is illustrated. The data blocks are denoted as B_i in the white boxes.

The mirrored images are marked with M_i in shaded boxes. The data blocks are striped across the disks horizontally on the top half of the disk array. Their images are clustered together and copied in a single disk vertically. All image blocks occupy the lower half of the disk array. This horizontal striping and vertical mirroring constitute the orthogonality property. Four data stripes and their images are illustrated by four

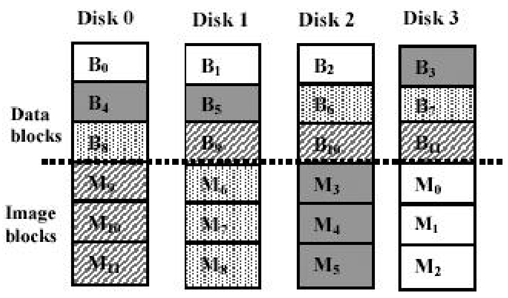


Figure 4.57: RAID-x: Orthogonal Striping and Mirroring

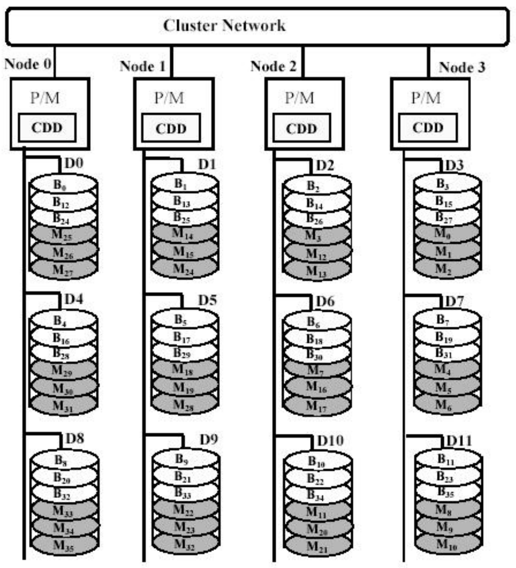


Figure 4.58: RAID-x: 4 x 3 Architecture with Orthogonal Striping and Mirroring

different shading patterns. On a *RAID-x*, the images can be copied and updated at the background, thus reducing the access latency. The top data stripe consists of the blocks B0, B1, and B2 and the images of these blocks, M0, M1 and M2, are stored in Disk 3. Similarly, the images of the second stripe B3, B4, and B5 are mapped to disk 2, etc. The rule of the thumb is that no data block and its image should be mapped to the same disk. Reading or writing can also achieve full bandwidth across all disks per each row. For example, all 4 blocks (B0, B1, B2, B3) in each row of Figure 4.57 forms a data stripe. Their images (M0, M1, M2, M3) are written into disk 2 and disk 3 in a delayed background time.

Figure 4.58 shows the orthogonal *RAID-x* architecture with 3 disks attached to each node. All disks within the same horizontal stripe, (B0, B1, B2, B3) are accessed in parallel. Consecutive stripe groups, such as (B0, B1, B2, B3) and (B4, B5, B6, B7), etc are accessed in a pipelined fashion, because multiple disks are attached on each SCSI bus.

To reduce the latency of remote disk access *cooperative disk drivers (CDDs)* are used. There is no need to use a central server. Each CDD maintains a peer-to-peer relationship with other CDDs. The idea is to redirect all I/O requests to remote disks. The results of the requests, including the requested data, are transferred back to the originating nodes. This mechanism gives the illusion to the OSs that the remote disks are attached locally.

Reactive Scheduling

type:

Application Level Method

contact:

Robert Ross, `rross@mcs.anl.gov`

description:

Reactive Scheduling is a decentralized, server-side approach to optimization in Parallel I/O. The focus is on utilizing system state and workload characteristic data available locally to servers, to make decisions on how to schedule the service of application requests.

motivation:

The emergence of new and different platforms has created a situation where the traditional Parallel I/O optimizations, while useful in some contexts, are not flexible enough to be applied in all cases. What is needed is a more general approach to optimization in Parallel I/O. Such an approach should recognize the components of the system and the parameters of workloads that determine performance: disks, network, and cache. It should provide a mechanism for determining what resources are currently limiting the performance of the system as a whole based on both system parameters and workload characteristics. Finally, using this information, it should schedule I/O operations and apply an I/O technique to best utilize the available resources.

features:

- implementation on top of ▶*PVFS1*
- 4 selection algorithms

- extendable model
- tested with contiguous, strided and multiple random block access

related work:

CHARISMA [80] was a project designed to characterize the behaviour of production parallel workloads at the level of individual reads and writes.

The implementation of ►*PPFS II* includes many advances in adaptive cache policy selection and adaptive disk striping.

citation:

[296]

details:

The *Reactive Scheduling* approach consists of three components:

- a set of Scheduling Algorithms
- a System Model
- a Selection Mechanism

The *Scheduling Algorithms* give options in terms of how the collection of requests that the system is presented at any given time is serviced, allowing to tailor service to optimize more for disk access, network access, or cache utilization. The *System Model* uses system-specific known values such as network and disk bandwidth, system state such as cache availability, and information describing the workload in progress to predict how the available optimizations would perform if utilized. The *Selection Mechanism* maps available system and workload information into the parameters used by the model. It then uses the model to predict performance using each of the available optimizations and chooses the one that it predicts will perform best. Together these three components provide a means for increasing performance in Parallel I/O systems by adapting the scheduling of service to best fit the workload presented to the system.

Reverse Aggressive Algorithm see ***greedyWriting***

RFT (Reliable File Transfer)

type:

Data Transfer Protocol

contact:

Ravi K. Madduri, `madduri@mcs.anl.gov`

url:

`http://www-unix.globus.org/ogsa/docs/alpha3/services/reliable_transfer.html`

description:

The *RFT* service is an ▶*OGSA* based service that provides interfaces for controlling and monitoring third-party file transfers using ▶*GridFTP* servers.

At the application level the file transfer client program performs the transfer. At the network level information is received from TCP. *RFT* monitors the state of the transfer and recovers from a variety of failures.

related work:

▷*Globus*, ▶*GridFTP*

citation:

[222]

RIO (Remote I/O)

type:

I/O Library

contact:

Ian Foster, `foster@mcs.anl.gov`

url:

`http://www-fp.globus.org/details/rio.html`

description:

RIO implements the ▶*ADIO* abstract I/O device interface specification. With the *RIO* library remote data, potentially in parallel file systems, can be located.

The *RIO* implementation comprises two components, as shown in Figure 4.59. The *RIO Client* defines a remote I/O device for ▶*ADIO*. This component translates ▶*ADIO* requests that name a remote file into appropriate communications to a *RIO*

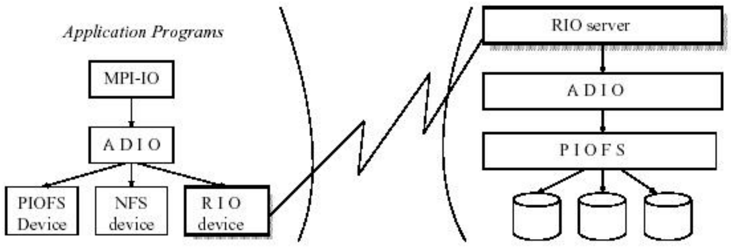


Figure 4.59: RIO: Mechanism

Server. The Server provides an interface to a particular file system, and serves requests from remote *RIO* Clients that need to access that file system. The *RIO* Server component itself calls ▶*ADIO* routines to access different file systems.

related work:

▶*GASS*, ▷*Globus*

citation:

[138], [292]

RLI (Replica Location Index) see ***Giggle***

RLS (Replica Location Service) see ***Giggle***

ROMIO

type:

I/O Library

contact:

`romio-maint@mcs.anl.gov`

url:

`http://www.mcs.anl.gov/romio/`

description:

ROMIO is a high-performance, portable implementation of ▶*MPI-IO*, the I/O chapter in MPI-2 [243]. *ROMIO* is optimized for ▶*Noncontiguous Data Access* patterns, which are common in Parallel applications. It has an optimized implementation of ▶*Collective I/O*, an important optimization in Parallel I/O. A key component of *ROMIO* that enables a portable ▶*MPI-IO* implementation is an internal abstract I/O device layer called ▶*ADIO*. *ROMIO* 1.2.5.1 includes everything defined in ▶*MPI-I/O* except support for file interoperability and user-defined error handlers for files.

related work:

▶*ADIO*, ▶*MPI-IO*

citation:

[352], [353], [354], [355], [357]

S

SAM (Sequential Data Access via Meta-Data)

type:

Data Access System

contact:

Lee Lueking, `lueking@fnal.gov`

url:

`http://d0db.fnal.gov/sam/`

description:

SAM is a file based data management and access layer between the storage management system and the data processing layers.

motivation:

The *SAM* system was developed at Fermilab to accommodate the high volume data management requirements for Run II [298] physics, and to enable streamlined access and mining of these large datasets. The *SAM* data management system is used for all Dzero [117] data cataloging, storage, and access. It is largely devoted to transparently delivering and managing caches of data.

features:

- clustering the data onto tertiary storage in a manner corresponding to access patterns
- caching frequently accessed data on disk or tape
- organizing data requests to minimize tape mounts
- estimating the resources required for file requests before they are submitted

application:

Run II [298]

related work:

▶*Enstore*

citation:

[45], [46], [47]

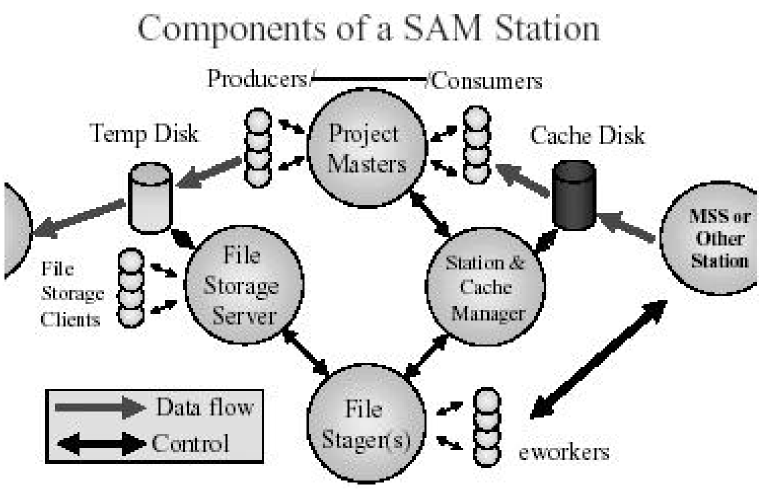


Figure 4.60: SAM: Station Components

details:

The architecture of the system is illustrated in Figures 4.60 and 4.61. The server elements which comprise each station are illustrated in Figure 4.60. Cache disks over which *SAM* is given exclusive control are managed by the *Station and Cache Manager*. This element is also responsible for starting and communicating with the *Project Masters*, which are in turn responsible for presenting data files to the user jobs that consume them. If the files already exist in the local cache, their locations are passed to the consumer and the files are locked in place while being used. If the files are not present in the local cache, they are brought in by *Stagers* from other stations or MSSs, replacing files which are no longer needed in the local cache. The *File Storage Server (FSS)* manages the storage of files. When a request to store a file to the MSS is made, the description of the file is added to the *SAM* meta-data, and the job is placed in a queue for copy to the designated MSS.

SAM Stations are network-distributed (Figure 4.61), and can receive data from, or route data through, other stations. Stations can also store data to local (relative to site) *Hierarchical Storage Systems (HSSs)* or forward data through other stations for storage in remote HSSs. In the current architecture there are several services shared among all stations, these include the:

- CORBA Name Service
- Central Oracle Database
- Global Resource Manager

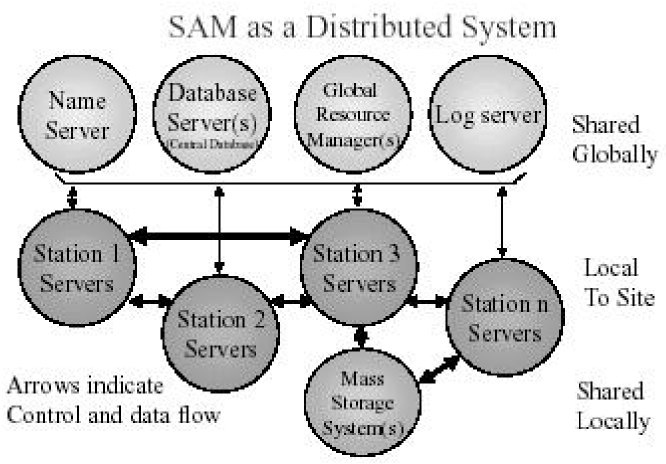


Figure 4.61: SAM: Overview of Distributed Components

- Log Server

The *Name Service* is the switchboard to register and receive addresses for the entire distributed system. The *Central Database* contains all meta-data for each file registered with the system, as well as station configuration, cache, and operational information. The *Global Resource Manager* reviews all requests for all stations and optimizes file deliveries. The *Log Server* receives logging information from all stations and records them in a central log file.

Sandboxing see ▷**Entropia**

SDDS (Scalable Distributed Data Structures)

type:

Data Format

contact:

Witold Litwin, `Witold.Litwin@dauphine.fr`

url:

`http://ceria.dauphine.fr/SDDS-bibliograhie.html`

description:

SDDSs allow for files whose records reside in *Buckets* at different server sites. The files support key-based searches and parallel/distributed scans with function (query) shipping.

An *SDDS* file is manipulated by the *SDDS* client sites. Each client has its own addressing schema called *Image* that it uses to access the correct server where the record should be. As the existing Buckets fill up, the *SDDS* splits them into new Buckets. The clients are not made aware synchronously of the splits. A client may have an outdated image and address an incorrect server. An *SDDS* server has the built-in capability to forward incorrect queries. The correct server sends finally the *Image Adjustment Message (IAM)* to the client. The information in an IAM avoids at least repeating the same error twice. It does not necessarily make the image totally accurate.

related work:

▶*DDS*, ▶*LH*$*_{RS}$

citation:

[217]

SDM (Scientific Data Manager)

type:

Mass Storage System

contact:

Alok Choudhary, `choudhar@ece.nwu.edu`

description:

SDM is a software system for storing and retrieving data that aims to combine the good features of both file I/O and databases.

motivation:

There were three major goals in developing *SDM*:

High-Performance I/O To achieve high-performance I/O a parallel file I/O system is used to store the data and ▶*MPI-IO* is used to access this data.

High-Level API *SDM* provides an API that does not require the user to know either ▶*MPI-IO* or databases. The user can specify the data with a high-level description language. *SDM* translates the user's requests into appropriate ▶*MPI-IO* calls.

Convenient Data-Retrieval Capability *SDM* allows the user to specify names and other attributes to be associated with a dataset. The user can retrieve a dataset by specifying a unique set of attributes for the desired data.

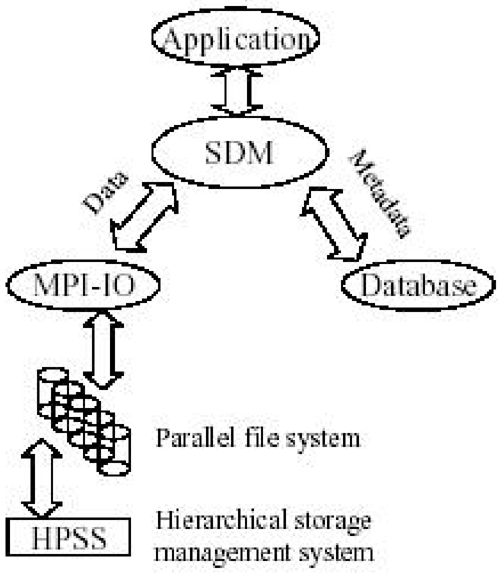


Figure 4.62: SDM: Architecture

features:

- combines the good features of high-performance file I/O and databases
- provides a high-level API to the user
- takes advantage of various I/O optimizations in a manner transparent to the user
- optimization for irregular applications

application:

Scientific Data Management System for Irregular Applications [255]

related work:

►*ADR*, ►*SRB*

citation:

[254], [256]

details:

SDM provides a high-level, user-friendly interface. Internally, *SDM* interacts with a database to store application-related meta-data and uses ►*MPI-IO* to store real data on a high-performance parallel file system. As a result, users can access data with the performance of parallel file I/O, without having to bother with the details of file I/O. Figure 4.62 illustrates the basic idea.

For regular applications *SDM* uses three database tables for storing meta-data: a *Run Table*, an *Access Pattern Table* and an *Execution Table*. These tables are made for each application. Each time an application writes datasets, *SDM* enters the problem size, dimension, current date, and a *unique identification number* to the *Run Table*. The *Access Pattern Table* includes the properties of each dataset, such as data type, storage order, data access pattern, and global size. *SDM* uses this information to make appropriate ▶*MPI-IO* calls to access the real data. The *Execution Table* stores a globally determined file offset denoting the starting offset in the file of each dataset.

For irregular applications additional tables for partitioning the index and the data arrays are used.

SDM supports three different ways of organizing data in files:

Level 1 Each dataset generated at each time step is written to a separate file.

Level 2 Each dataset (within a group) is written to a separate file, but different iterations of the same dataset are appended to the same file.

Level 3 All iterations of all datasets belonging to a group are stored in a single file.

Server-Directed I/O

type:

Device Level Method

contact:

Marianne Winslett, `winslett@uiuc.edu`

description:

Server-Directed I/O is a ▶*Collective I/O* technique proposed by the ▶*Panda* research group at the University of Illinois, which is a derivative of ▶*Disk-Directed I/O*. Like ▶*Disk-Directed I/O*, in *Server-Directed I/O*, I/O servers actively optimize the disk accesses by utilizing the data distribution in memory and on disk. However, instead of using physical disk block locations to reorder disk accesses as ▶*Disk-Directed I/O* does, *Server-Directed I/O* uses logical file offsets and performs sequential file I/O, relying on the underlying local file system which is often optimized for such access patterns. This technique was implemented in the ▶*Panda* Parallel I/O library, and tested with different platforms, array sizes, data distributions, and processor configurations. Experimental results show that without actual disk block layout information, almost the full capacity of the disk subsystem can be utilized with *Server-Directed I/O*.

related work:

▶*Disk-Directed I/O*, ▶*Panda*, ▶*Two-Phase I/O*

citation:

[312], [313]

SFIO (Striped File I/O)

type:

I/O Library

contact:

Emin Gabrielyan, `Emin.Gabrielyan@epfl.ch`

description:

SFIO is a Striped File I/O library for Parallel I/O within an MPI [241] environment.

motivation:

SFIO aims at offering scalable I/O throughput by using small striping units.

related work:

▶*MPI-IO*

citation:

[145], [146]

details:

The *SFIO* library is implemented using MPI-1.2 [241] message passing calls. The local disk access calls are non-portable, however they are separately integrated into the source for the Unix and the Windows NT versions. The *SFIO* parallel file striping library offers a simple Unix like interface. To route calls to the *SFIO* interface the ▶*ADIO* layer of MPICH [242] was modified.

SFIO is not a block-oriented library. The amount of data accessed on the disk and transferred over the network is the size specified at the application level. The functional architecture of the *SFIO* library is shown in Figure 4.63. On top of the graph is the application's interface to data access operations and at the bottom are the I/O node operations. `mread` and `mwrite` are non-optimized single block access functions and the `mreadc` and `mwritec` operations are their optimized counterparts. The `mreadb` and `mwriteb` operations are multi-block access functions. All the interface access functions are routed to the *mrw cyclical* distribution module. This module is responsible for data striping. The network communication and disk access optimization is demonstrated by the remaining part of the graph.

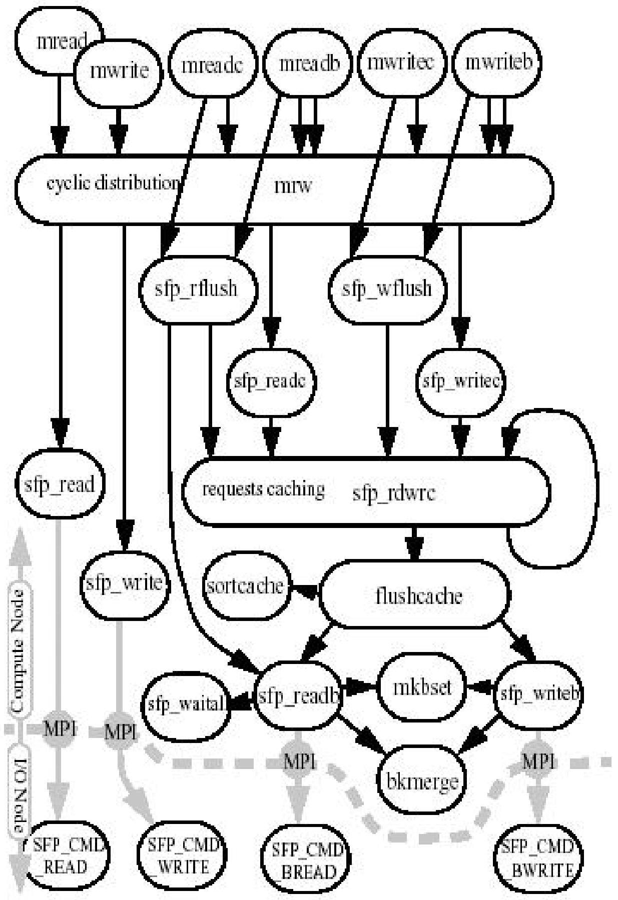


Figure 4.63: SFIO: Functional Architecture

Slice

type:

Storage System

contact:

Jeff S. Chase, `chase@cs.duke.edu`

url:

`http://www.cs.duke.edu/ari/slice/`

description:

Slice is a scalable network I/O service based on ►*Trapeze*.

motivation:

The goal of the *Slice* architecture is to provide a network file service with scalable bandwidth and capacity while preserving compatibility with off-the-shelf clients and file server appliances.

features:

- storage at network speed
- Internet/LAN as a scalable storage backplane
- decentralized file service structure
- intelligent block placement and movement

related work:

►*TPIE*, ►*Trapeze*

citation:

[22]

details:

The *Slice* file service is implemented as a set of loadable kernel modules in the FreeBSD [143] OS. A *Slice* is configured as a combination of server modules that handle specific file system functions: directory management, raw block storage, efficient storage of small files, and network caching. Servers may be added as needed to scale different components of the request stream independently, to handle a range of data-intensive and meta-data-intensive workloads. *Slice* is designed to be compatible with standard NFS [302] clients, using an interposed network-level packet translator to mediate between each client and an array of servers presenting a unified file system view.

Spitfire

type:

Mass Storage System

contact:

Diana Bosio, `Diana.Bosio@cern.ch`

url:

`http://spitfire.web.cern.ch/Spitfire/`

description:

Spitfire provides a uniform way to access many RDBMSs through standard Grid protocols and well-published Grid interfaces. *Spitfire* is a project of Work Package WP2 within the ▷*EDG* project.

motivation:

Short lived, small amounts of data and meta-data that needs to be highly accessible to many users and applications. Throughout the Grid there is a need for an abstract high-level Grid database interface.

application:

LHC [211]

related work:

►*EDG*

citation:

[52]

details:

The *Spitfire* middleware is placed between client and RDBMS. The JDBC API [41] defines a uniform vendor independent way to communicate with a wide range of RDBMSs. The *Spitfire* server is implemented as a Java servlet running within a container of a virtual hosting environment such as the Apache Tomcat servlet container [26] or a commercial servlet container. The architecture can be summarized as follows:

$$\text{Client} \xleftrightarrow{\text{SOAP/HTTP(S)}} \text{Spitfire} \xleftrightarrow{\text{JDBC}} \text{RDBMS}$$

Database services are implemented as a Java servlet through AXIS [25]. SOAP [328] is used for remote messaging to ensure interoperability. Three SOAP services are defined: a *Base* service for standard operations, an *Admin* service for administrative access and an *Info* service for information on the database and its tables.

SRB (Storage Resource Broker)

type:

Data Access System

contact:

Arcot Rajasekar, `sekar@sdsc.edu`

url:

`http://www.npaci.edu/dice/srb`

description:

The SDSC *SRB* is a client-server middleware that provides a uniform interface for connecting to heterogeneous data resources over a network and accessing replicated datasets. *SRB*, in conjunction with the *Meta Information Catalog (MCAT)* [225], provides a way to access datasets and resources based on their attributes rather than their names or physical locations.

motivation:

SRB provides seamless access to data stored on a variety of storage resources including file systems, database systems and archival storage systems.

features:

- uniform storage interface
- Meta Information Catalog (MCAT)
- the *Collection* Hierarchy
- hierarchical access control
- Tickets: a flexible mechanism for controlling read access to data
- Physical Storage Resources (PSRs):
  - with file system interfaces
  - database system interfaces
- Logical Storage Resources (LSRs): combination of one or more PSRs into a single LSR
- proxy operations: data handling operations without involving the client

- federated operations: Access to distributed storage resources
- authentication and encryption
- activity logs
- types of applications
- peer-to-peer federation of logical name spaces (meta data catalogs)

application:

Projects using *SRB* can be found at the following website:
`http://www.npaci.edu/dice/srb/Projects/main.html`

related work:

▶*DPSS*, ▶*GASS*, ▶*GridFTP*, ▶*HPSS*

citation:

[48], [51], [204], [245], [283], [284]

details:

As shown in Figure 4.64, applications use the *SRB* middleware to access heterogeneous storage resources using a client-server network model, which consists of three parts: *SRB Clients*, *SRB Servers*, and MCAT.

The *SRB Server* consists of one or more *SRB Master Daemon* processes with *SRB Agent* processes that are associated with each Master, which controls a distinct set of storage resources. Each time a client opens a connection to a master a *SRB* Agent is spawned.

Two types of API are provided on the client side:

- high-level API: functions like query, update meta-data, connecting to a server and creation of data items
- low-level API: direct storage operations like open, read & delete

MCAT is used to record location information for PSRs as well as for data items. The catalog also contains meta-data that is used for implementing hierarchical access control, the collection / subcollection hierarchy, and the ticket mechanism.

Pipelined SRB: Two new funcions `srbFileChunkWrite` and `srbFileChunkRead` have been added to the original ▶*SRB* to enable pipelining which improves the performance for remote read/writes larger than 1MB up to 50% [245]. The function `srbFileChunkRead` sends a request to the server with information on the file size,

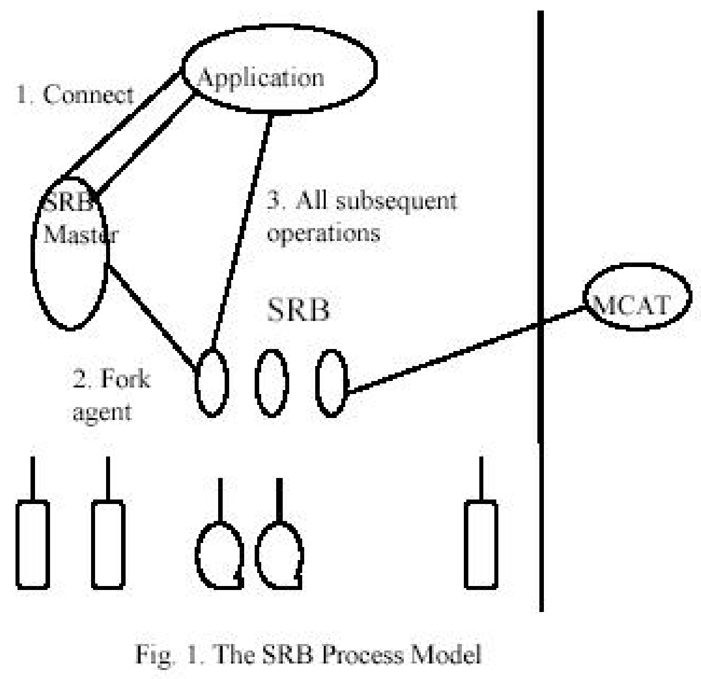


Figure 4.64: SRB: Process Model

and the number of portions, or chunks, that it should be split into for the pipelining. The server queues the appropriate number of asynchronous reads on its local disk. After sending the initial read request, the client application waits for the chunks of data. Figure 4.65 (a) shows the sequence of requests between the client and the server. In calling the `srbFileChunkWrite` function the user specifies the name of the file, the size of the file, and can choose the number of chunks or pieces that should be used to split the total data. Figure 4.65 (b) shows the sequence of requests between the client and the server.

SRM (Storage Resource Manager)

type:

Data Access System

contact:

Arie Shoshani, `shoshani@lbl.gov`

url:

`http://sdm.lbl.gov/projectindividual.php?ProjectID=SRM`

description:

SRMs dynamically optimize the use of storage resources of distributed large datasets.

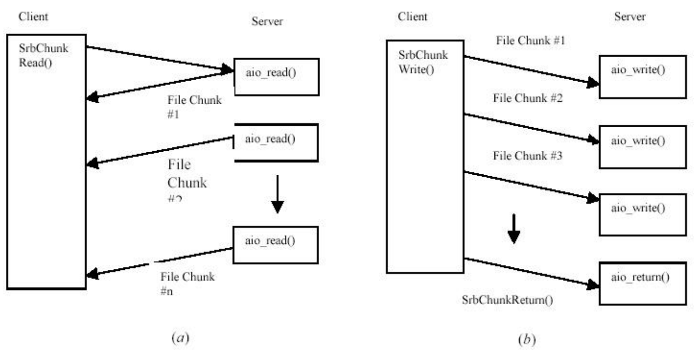


Figure 4.65: SRB: Pipelined SRB Sequence

motivation:

The access to data is becoming the main bottleneck in data-intensive applications because the data cannot be replicated in all sites. *SRMs* are designed to dynamically optimize the use of storage resources to help unclog this bottleneck.

related work:

►*STACS*

citation:

[324]

details:

SRMs are middleware software modules whose purpose is to manage in a dynamic fashion what resides on the storage resource at any one time. SRMs do not perform file movement operations, but rather interact with OSs and MSSs to perform file archiving and file staging, and invoke middleware components (such as ►*GridFTP*) to perform file transfer operations. There are several types of *SRMs*:

Disk Resource Manager (DRM) dynamically manages a single shared disk cache (a single disk, a collection of disks, or a RAID [83] system)

Tape Resource Manager (TRM) middleware layer that interfaces to MSSs such as ►*HPSS*, ►*Enstore*

Hierarchical Resource Manager (HRM) TRM that has a staging disk cache for its use. It can be viewed as a combination of a DRM and a TRM. It can use the disk cache for prestaging files for clients, and for sharing files between clients.

STACS (Storage Access Coordination System)

type:

Data Access System

contact:

Arie Shoshani, `shoshani@lbl.gov`

url:

`http://sdm.lbl.gov/projectindividual.php?ProjectID=STACS`

description:

STACS is part of the HEP STAR [332] experiment. It has to determine which files contain the experiment's reconstructed data (or the raw data if they are requested), and to schedule their caching from tape for processing.

motivation:

The need for smart cache management systems, that coordinate both the retrieval of data from tapes and the use of the restricted disk cache.

application:

PHOENIX [274], STAR [332]

related work:

►*SRM*

citation:

[57]

details:

The *STACS* architecture consists of four modules that can run in a distributed environment:

- Query Estimator (QE)
- Query Monitor (QM)
- File Catalog (FC)
- Cache Manager (CM)

The physicists interact with *STACS* by issuing a query that is passed to the QE. If the user finds the time estimate reasonable than a request to execute the query is issued and the relevant information about files and events is passed to the QM. The QM handles such requests for file caching for all the users that are using the system

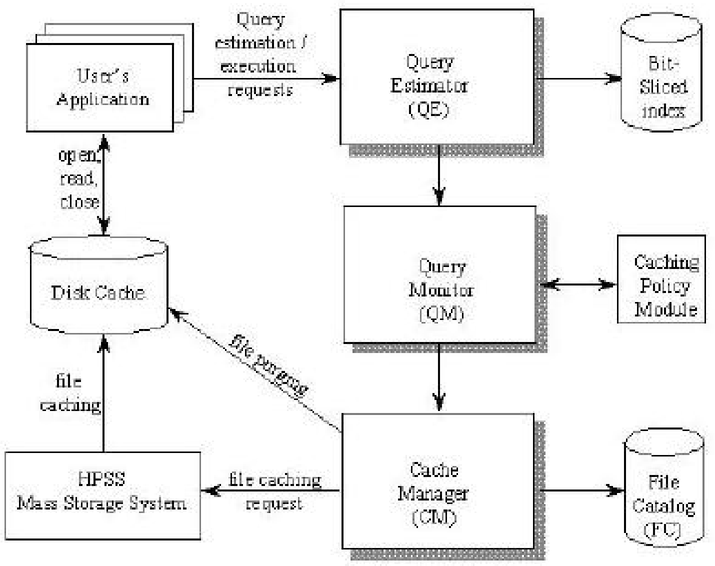


Figure 4.66: STACS: Architecture

concurrently. After the QM determines which files to cache, it passes the file requests to the CM one at a time. The CM is the module that interfaces with the MSS, which in the case of STAR [332] is ▶*HPSS*. To be able to transfer the file from ▶*HPSS* to local disk the CM needs to convert the logical name into a real physical name. This mapping can be obtained by consulting the FC, which provides a mapping of an file identifier into both a ▶*HPSS* file name and a local disk file name (see Figure 4.66).

Stream-Based I/O

type:

General Method

contact:

Robert B. Ross, `rross@mcs.anl.gov`

description:

Stream-Based I/O attempts to address network bottlenecks in Parallel I/O systems. It is the concept of combining small accesses into more efficient, large ones applied to data transfer over the network. Data being moved between clients and servers is considered to be a stream of bytes regardless of the location of data bytes within a file. This is similar to a technique known as *message coalescing* in interprocessor communication. These streams are packetized by underlying network protocols (e.g.

TCP) for movement across the network. Control messages are placed only at the beginning and end of the data stream in order to minimize their effects on packetization. This is accomplished by calculating the stream data ordering on both client and server. This is strictly a technique for optimizing network traffic. When coupled with a server that focuses on the network (almost network directed I/O), peak performance can be maintained for a variety of workloads, particularly when network performance lags behind disk performance or when most data on I/O servers is cached.

related work:

►*Data Sieving*, ►*Disk-Directed I/O*, ►*PVFS1*, ►*Server-Directed I/O*, ►*Two-Phase I/O*

citation:

[297]

⟨*STXXL*⟩

type:

I/O Library

contact:

Roman Dementiev, `dementiev@mpi-sb.mpg.de`

url:

`http://www.mpi-sb.mpg.de/~rdementi/stxxl.html`

description:

⟨*STXXL*⟩ implements algorithms and data structures from the standard template library STL for massive datasets. It consists of an *Asynchronous I/O Primitives Layer* (files, I/O requests, disk queues, completion handlers), a *Block Management Layer* (typed block, block manager, buffered streams, block prefetcher, buffered block writer), a *STL-User Layer* (containers, algorithms) and a *Streaming Layer* (pipelined sorting, zero-I/O scanning).

The *Asynchronous I/O Primitives Layer*, the lowest layer of ⟨*STXXL*⟩, supports efficient asynchronous I/O that is currently implemented using multi-threading and unbuffered blocking file system I/O. From the higher layers, (pipelined) sorting, (pipelined) scanning and some containers (vectors, stacks, priority queues) are supported.

⟨*STXXL*⟩ has the following performance features:

- transparent support of multiple disks
- variable block lengths
- overlapping of I/O and computation
- prevention of OS file buffering overhead
- algorithm pipelining

related work:

▶*LEDA-SM*

citation:

[110]

Sub-Filing

type:

Application Level Method

contact:

Gokhan Memik, `memik@ece.nwu.edu`

description:

Sub-Filing is an optimization technique invisible to the user used in ▶*April*, helping to efficiently manage the storage hierarchy which can consist of a tape sub-system, a disk sub-system and a main-memory. Each global tape-resident array is divided into *Chunks*, each of which is stored in a separate sub-file on tape. The Chunks are of equal sizes in most cases. Figure 4.67 shows a two-dimensional global array divided into 64 Chunks. Each Chunk is assigned a unique chunk coordinate (x1, x2), the first Chunk having (0,0) as its coordinate. A typical access pattern is shown in Figure 4.68. In this access a small two-dimensional portion of the global array is requested. In receiving such a request, the library performs three important tasks:

1. determining the sub-files that collectively contain the requested portion (called *cover*)
2. transferring the sub-files that are not already on disk from tape to disk
3. extracting the required data items (array elements) from the relevant sub-files from disk and copying the requested portion to a buffer in memory provided by the user call

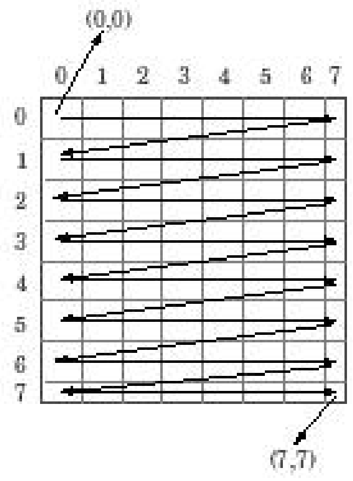


Figure 4.67: Sub-Filing: A Global Tape-Resident Array Divided into 64 Chunks

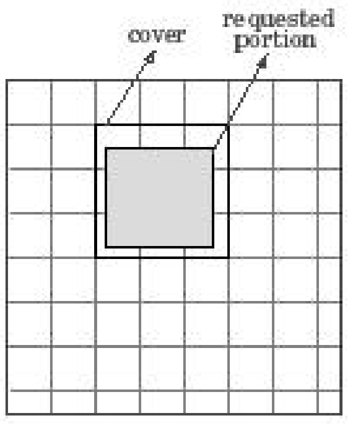


Figure 4.68: Sub-Filing: An Access Pattern and its Cover

related work:

▶*April*

citation:

[229]

Sun HPC PFS(Parallel File System)

type:

File System

description:

Sun HPC *PFSs* are based on three abstract entities: *PFS I/O Servers, Storage Objects*, and the *PFS* themselves. A *PFS I/O Server* is simply a Sun Ultra HPC node that has been configured in the RTE with additional attributes that enable it to participate with other *PFS I/O Servers* in the distributed storage and retrieval of PFS. A *PFS I/O Server* also differs from other nodes in that it runs a *PFS I/O Daemon*, which manages the storage and retrieval of data to and from its portion of the parallel file systems. Certain other attributes are also defined for the file system, such as the amount of memory on each *PFS I/O Server* that is reserved for I/O buffering and the network interface assigned to each *PFS I/O Server*.

A *Storage Object* is an RTE abstraction that represents one portion of a parallel file system–that is, the portion of a file system that resides on a particular storage device and is managed by the *PFS* I/O Daemon associated with that device. To create a specific parallel file system, the RTE is given a name and a set of storage objects to be associated with that file system. File system naming, Storage Object associations, and the other configuration tasks involved in creating and defining parallel file systems are all done using `tmadmin` commands within the *PFS* context.

related work:

▶*PVFS2*

citation:

[335]

T

TIP (Transparent Informed Prefetching and Caching)

type:

Access Anticipation Method

contact:

R. Hugo Patterson, `rhp@acm.org`

url:

`http://www.pdl.cmu.edu/TIP/index.html`

description:

TIP evolved out of a desire to reduce read latency. It uses a system performance model to estimate the *benefit* of using a buffer for ▶*Prefetching* and the *cost* of taking a buffer from the cache. The estimates are computed dynamically and buffer is reallocated from the cache for ▶*Prefetching* when the benefit is greater than the cost.

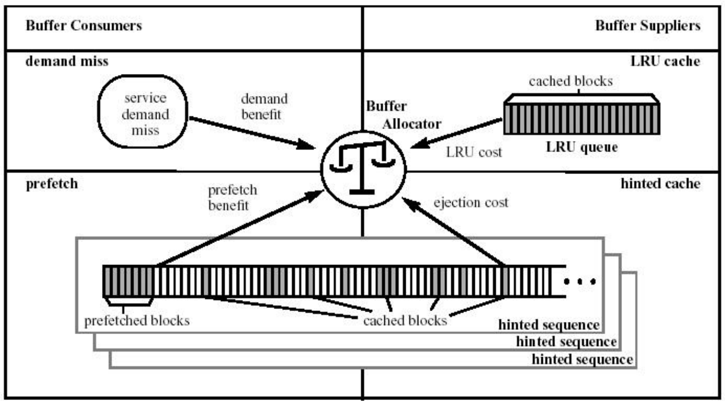


Figure 4.69: TIP: Estimation

As shown in Figure 4.69, the *TIP* system architecture includes three key components:

- *Independent Estimators* dynamically estimate either (1) the benefit (reduction in I/O service time) of allocating a buffer for ▶*Prefetching* or a demand access, or (2) the cost (increase in I/O service time) of taking a buffer from the *Least-Recently-Used (LRU)* queue or the cache of hinted blocks. Because the estimators are independent, they are relatively simple. Estimator independence also makes the system extensible because the addition of a new estimator does not require changes in the existing ones.

- A *Common Currency* for the expression of the cost and benefit estimates allows the comparison of the independent estimates at a global level.
- An efficient *Allocation Algorithm* finds the globally least valuable buffer and reallocates it for the greatest benefit when the estimated benefit exceeds the anticipated cost.

related work:

►*Prefetching*

citation:

[270]

TPIE (Transparent Parallel I/O Environment)

type:

Intelligent I/O System

contact:

Lars Age, `large@cs.duke.edu`

url:

`http://www.cs.duke.edu/TPIE/`

description:

TPIE is designed to allow programmers to write high-performance I/O-efficient programs for a variety of platforms [333, page 149]. The first development phase focused on supporting algorithms with a *sequential* I/O pattern and the second on supporting online I/O-efficient data structures.

motivation:

The goal *TPIE* is to provide a *portable, extensible, flexible,* and *easy to use* C++ programming environment for *efficiently* implementing I/O algorithms and data structures for the ►*PDM*.

related work:

►*geo**, ►*LEDA-SM*, ►*PDM*

citation:

[28], [376]

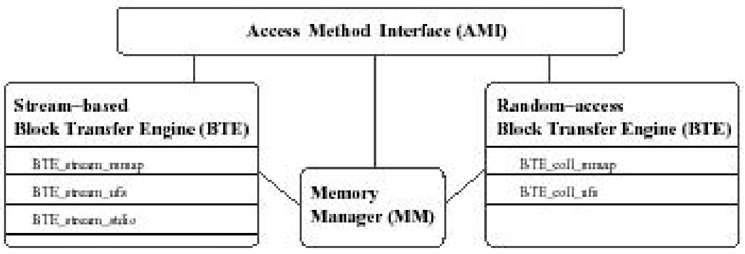

Access Method Interface (AMI)
Stream–based Block Transfer Engine (BTE)
BTE_stream_mmap
BTE_stream_ufs
BTE_stream_stdio
Memory Manager (MM)
Random–access Block Transfer Engine (BTE)
BTE_coll_mmap
BTE_coll_ufs

Figure 4.70: TPIE: Structure of the Kernel

details:

TPIE is a templated C++ library consisting of a kernel and a set of I/O-efficient algorithms and data structures implemented on top of it. The kernel is responsible for abstracting away the details of the transfers between disk and memory, managing data on disk and in main memory, and providing a unified programming interface simulating the ▶*PDM*. Each of these tasks are performed by a separate module inside the kernel, resulting in a highly extensible and portable system.

The TPIE library has been built in two phases. The first phase was initially developed for algorithms based on sequential scanning, like sorting, permuting, merging and distributing. In this*Stream-Based View* of I/O, the computation can be viewed as a continuous process in which data is fed in streams from an outside source and streams of results are written behind. The *TPIE* kernel designed in this phase consists of three modules: the *Stream-Based Block Transfer Engine (BTE)*, responsible for packaging data into blocks and performing I/O transfers, the *Memory Manager (MM)*, responsible for managing main memory resources, and the *Application Method Interface (AMI)*, which provides the high-level interface to the stream functionality provided by the BTE and various productivity tools for scanning, permutation routing, merging, sorting, distribution, and batch filtering.

I/O-efficient data structures typically exhibit random I/O patterns. The second phase of *TPIE* provides support for implementing these structures by using the full power of the disk model. Maintaining the design framework presented above, the new functionality is implemented using a new module, the *Random-Access BTE*, as well as a new set of AMI tools.

Figure 4.70 depicts the interactions between the various components of the *TPIE* kernel.

The AMI The AMI provides the high-level interface to the programmer. This is the only component with which most programmers will need to directly interact. The AMI tools needed to provide the random access I/O functionality consist of a front-end to the BTE block collection and a typed view of a disk block.

The Memory Manager (MM) The MM is responsible for managing main memory resources. All memory allocated by application programs or other components of *TPIE* is handled by the MM.

The Stream-Based BTE The BTE is responsible for moving blocks of data to and from the disk. It is also responsible for scheduling asynchronous read-ahead and write-behind when necessary to allow computation and I/O to overlap.

The Random-Access BTE The Random-Access BTE implements the functionality of a *Block Collection*. A Block Collection is a set of fixed size blocks. A Block is the unit amount of data transfered between disk and main memory, as defined by the ►*PDM*. A Block can be viewed as being in one of two states: on disk or in memory. The collection must support four main operations: `read`, `write`, `create` and `delete`.

Trapeze

type:

Data Access System

contact:

Jeff S. Chase, `chase@cs.duke.edu`

url:

`http://www.cs.duke.edu/ari/trapeze/index.html`

description:

The *Trapeze* project is an effort to harness the power of Gigabit networks to cheat the disk *I/O bottleneck* for I/O-intensive applications. The network is used as the sole access path to external storage, pushing all disk storage out into the network.

motivation:

The goal of the *Trapeze* project is to develop new techniques for high-speed communication and fast access to stored data in workstation clusters, and to demonstrate their use in prototypes of enhanced Unix systems.

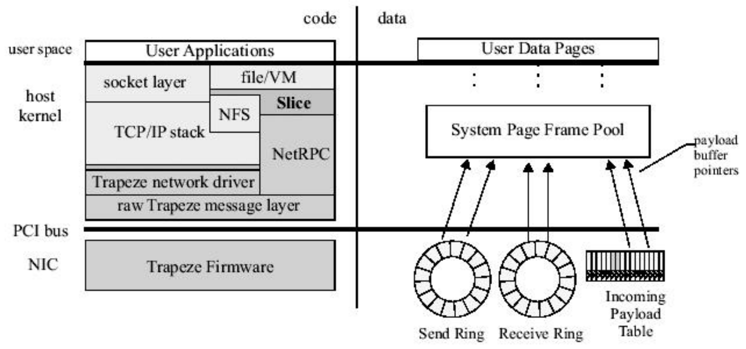


Figure 4.71: Trapeze: Prototype

features:

The *Trapeze* messaging system has several features useful for high-speed network storage access:

- separation of header and payload
- large *Maximum Transmission Units (MTUs)* with scatter/gather *Direct Memory Access (DMA)*
- adaptive message pipelining

related work:

►*Slice*, ►*TPIE*

citation:

[81]

details:

The *Trapeze* messaging system consists of two components: a *Messaging Library* that is linked into the kernel or user programs, and a *Firmware Program* that runs on the Myrinet [244] NIC.

Trapeze was designed primarily to support fast kernel-to-kernel messaging alongside conventional TCP/IP networking. Figure 4.71 depicts the structure of the current prototype client based on FreeBSD 4.0 [143]. The *Trapeze Library* is linked into the kernel along with a network device driver that interfaces to the TCP/IP protocol stack. Network storage access bypasses the TCP/IP stack, instead using NetRPC [249], a lightweight communication layer that supports an extended RPC model optimized for block I/O traffic.

TSS (Temperature Sensitive Storage)

type:

Storage System

contact:

K. Gopinath, `gopi@csa.iisc.ernet.in`

description:

The prototype system *TSS* is a host-based driver (a volume manager) for a RAID [83] storage system with 3 tiers: a small RAID1 (Mirrored) tier and larger RAID5 (Block-Interleaved Distributed-Parity) and compressed RAID5 (cRAID5) tiers. Based on access patterns (temperature), the driver automatically migrates frequently accessed (hot) data to RAID1 while demoting not so frequently accessed (cold) data to RAID5/ cRAID5.

motivation:

Disk access patterns display good locality of reference, especially in non-scientific environments. For achieving cost-effective storage systems with Terabytes of data, such locality can be exploited by using a multi-tiered storage system with different price-performance tiers that adapts to the access patterns by automatically migrating the data between the tiers.

features:

- uses only commodity hardware
- no storage media dependencies such as use of only SCSI or only IDE disks
- supports reliable persistance semantics

related work:

HP AutoRAID is a firmware implementation of the same idea [402].

citation:

[156]

details:

The *TSS* storage system is implemented as a layered device driver, which uses the host processor for performing operations like I/O processing and parity computation. The device driver is layered on top of block storage media (generally disks). A given I/O request is divided into separate I/O requests each of which is issued to the device drivers of the underlying storage media it is configured to use. To guarantee reliable

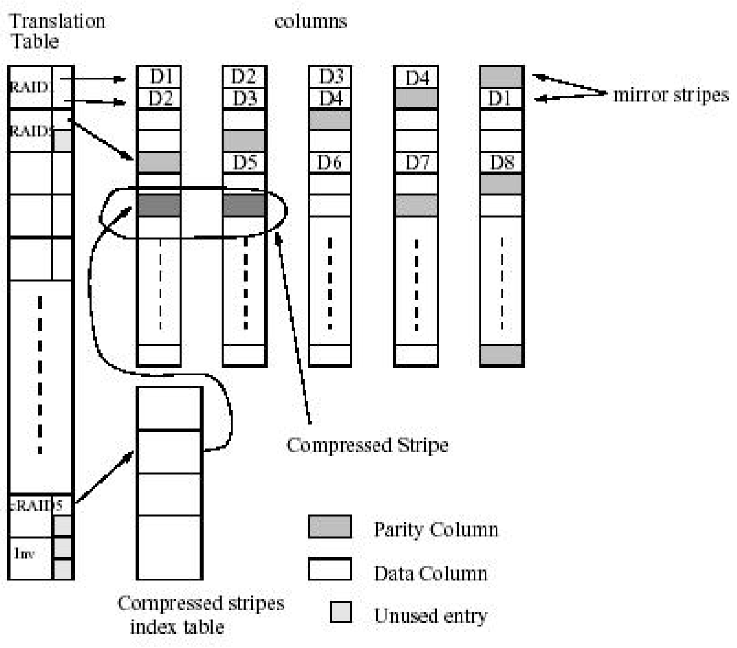


Figure 4.72: TSS: Storage Organization

persistence semantics, changes to the state of a *Stripe* are made using both ordered updates and a separate logging device.

The *Physical Storage* is organized in RAID5 fashion as shown in Figure 4.72. The storage consists of a set of *Columns*. The term column is used to distinguish them from disks. A Column is divided into contiguous regions called *Stripe Units*. A *Stripe* is a formed by grouping one Stripe Unit from each Column.

The *Logical Storage* can be viewed as a collection of *Logical Stripes*. The logical to physical translation is done by the driver, so that the user s view of the data does not change even as the Stripes undergo change of state. A Logical Stripe can be in any of the following states:

Invalid No backing store is allocated for this type of Logical Stripe. Initially all the Logical Stripes belong to this type.

RAID1 Two Physical Stripes provide backing store for a declustered RAID1 Logical Stripe.

RAID5 Because the Physical Storage is organized in RAID5 fashion, a single Physical Stripe provides backing store for a RAID5 Logical Stripe.

cRAID5 The data of this Stripe is compressed and stored in a Physical Stripe.

Migrations result when a Logical Stripe changes type. A Logical Stripe changes from invalid to RAID5 on a write to it. After the migration is completed, the I/O is retried (this time in RAID5 fashion). If the request is a partial stripe write, another migration is triggered to make it RAID1. Thus a partial write to an invalid type Stripe ultimately results in its migration to RAID1. A full stripe write to an invalid Stripe only makes it a RAID5 Stripe.

RAID1 to RAID5 migration usually happens when a RAID1 Stripe is victimized to give one of its Physical Stripes to a currently invalid Logical Stripe to make it RAID5 or to a RAID5 Stripe to make it RAID1. The strategy for migrations to/from cRAID5 is similar. First migrate the needed Stripe to RAID5 (leaving the other Stripes in cRAID5 with certain parts invalid) and migrate to RAID1 as needed. RAID5 to cRAID5 migrations typically take place through policy mechanisms when the data becomes cold. To maintain access frequencies of Stripes a LRU policy is used.

Two-Phase I/O

type:

Application Level Method

contact:

Alok Choudhary, `choudhar@ece.nwu.edu`

description:

Two-Phase I/O is one approach to ►*Collective I/O*. It is a method for reading/writing in-core arrays from/to disks. The basic principle behind the method is based on the fact that I/O performance is better when processors make a small number of large (respective data size) and contiguous requests instead of a large number of small ones. *Two-Phase I/O* splits the reading of an in-core array into main memory in two phases:

1. Processors always read data assuming a *Conforming Distribution*, which is defined as as a distribution of an array among processors such that each processor's local array is stored contiguously in the file, resulting in each processor reading a single large chunk of data. For an array stored in column-major order, a column-block distribution is the Conforming Distribution.
2. Data is redistributed among processors to the desired distribution. Since I/O cost is orders of magnitude more than communication cost, the cost incurred by the this phase is negligible.

This results in high granularity data transfers and the use of a higher bandwidth of the interconnection network (for interprocess communication).

Two-Phase I/O which is suitable for in-core arrays was extended to access arbitrary sections of *out-of-core* arrays (▶*EM (External Memory) Algorithms and Data Structures*). The *Extended Two-Phase I/O* performs I/O for out-of-core arrays by:

- dynamically partitioning the I/O workload among processors, depending on the access requests
- combining several I/O requests into fewer larger granularity requests
- reordering requests so that the file is accessed in proper sequence
- eliminating simultaneous I/O requests for the same data

related work:

▶*Collective I/O*

citation:

[109], [350]

V

VDC (Virtual Data Catalog) see ▷**VDG**

ViPFS

type:

File System

contact:

Erich Schikuta, `erich.schikuta@univie.ac.at`

description:

ViPFS is a file system on top of ViPIOS [305]. *ViPFS* aims at:

- providing tools to manage files on ViPIOS
- delivering a C-interface for application development
- viewing files as continuous data
- taking advantage of parallelism due to the underlying physical distribution

ViPFS implements a command-line interface and a C language interface providing basic functionality.

related work:

ViPIOS [305]

citation:

[128], [144]

ViPIOS Islands

type:

Intelligent I/O System

contact:

Erich Schikuta, `erich.schikuta@univie.ac.at`

description:

ViPIOS Islands aim to utilize I/O resources on distributed clusters. A *ViPIOS Island* is defined to be a closed system with its own name space consisting of a number of ViPIOS [305] 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 of the connection controller responsible for that Island. Each *ViPIOS Island* has its own name space. All parts of a single file are stored on one dedicated Island. To distinguish between files on different Islands with the same name, the buddy handle must be specified when opening a file. Each file is assigned a specific ViPIOS server process which is called the *Synch Controller* of that file. Each file has exactly one Sync Controller but a Sync Controller can serve multiple files.

related work:

ViPIOS [305]

citation:

[144], [304]

W

WiND (Wisconsin Network Disks)

type:

Mass Storage System

contact:

Andrea Arpaci-Dusseau, `dusseau@cs.wisc.edu`

description:

WiND is a MSS developed to understand the key adaptive techniques required to build a truly manageable NAS system.

motivation:

Manageability is more challenging in storage clusters due to their additional complexity. This complexity is a result of both the networking hardware and protocols between clients and disks, and the nature of modern disks drives. Thus the main focus in *WiND* is to develop the key techniques necessary to build truly manageable NAS. To achieve this goal and to fully exploit the potential of the underlying hardware, all of the software is distributed and scalable.

features:

- run-time adaptive layout and access
- off-line monitoring and adaption
- adaptive caching

related work:

►*Petal*

citation:

[31]

details:

WiND is comprised of five major software components, broken down into two groups. The first three are the run-time and off-line adaptive elements of *WiND*: *SToRM, GALE*, and *Clouds*. The other two are key pieces of supporting infrastructure: RAIN and NeST. The overall system architecture is presented in Figure 4.73.

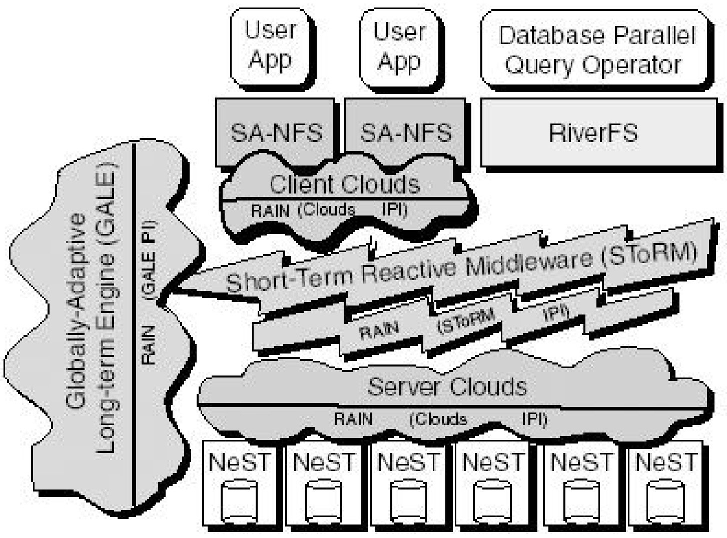


Figure 4.73: WiND: System Architecture

Short-Term Reactive Middleware (SToRM) is a distributed software layer that interposes between clients and servers and performs run-time adaptation for data access and layout. It adapts to short-term changes in workload characteristics and disk performance by quickly adjusting how much each client reads from or writes to each disk. SToRM is used by file systems to adapt to volatile disk behaviour at run-time and deliver full bandwidth to clients without intervention.

Globally Adaptive Long-Term Engine (GALE) Short-term adaptation does not solve all of the problems encountered in dynamic, heterogeneous environments; it lacks *global perspective*. To provide a long-term view of system and workload activity and to optimize system performance in ways not possible at run-time, there exists an additional software structure, the GALE. GALE provides three basic services in *WiND*.

- GALE performs *system monitoring*, using both active and passive techniques to gather workload access patterns and device performance characteristics, and detecting anomalies in component behaviour.
- GALE decides when to perform a global optimization itself via *action instantiation*, eg.: GALE may replicate an oft-read file for performance reasons.
- GALE provides information to SToRM and Clouds via *hint generation*.

Clouds Both SToRM and GALE are designed to adapt data flows to and from disk.

However, many requests to a network-attached disk may be satisfied from in-memory caches. Thus, Clouds provides flexible caching for NAS. Clouds provides mechanisms and policies for both client-side and server-side caches, taking variable costs into account. Clouds also can employ cooperative techniques to conglomerate server-side cache resources, and potentially hide disk variations from clients.

Rapid Access to Information (RAIN) The first piece of software infrastructure encapsulates the acquisition and dispersal of information within SToRM, GALE, and Clouds. RAIN provides *Information Programming Interfaces (IPIs)* to each software subsystem, which hide details of information flows and greatly simplify system structure and maintainability.

Network Storage (NeST) The second piece of software infrastructure, ►*NeST*, provides flexible and efficient single-site storage management.

X

XDGDL (eXtended Data Grid Definition Language)

type:

Data Format

contact:

Erich Schikuta, `erich.schikuta@univie.ac.at`

description:

XDGDL is an XML based language, which provides a homogeneous framework for describing data on all interpretative levels, which are *Data Representation, Structured File Information, Physical Data Layout, Problem Specific Data Partitioning, and General Information Semantics*.

motivation:

XDGDL makes it possible to add certain semantic information to a stored dataset.

related work:

[127]

citation:

[53], [54]

details:

XDGDL makes use of a novel file hierarchy with three independent layers in the Parallel I/O architecture:

Problem Layer defines the problem specific data distribution among the cooperating parallel processes

File Layer provides a composed view of the persistently stored data in the system

Data Layer defines the physical data distribution among the available disks

These layers are separated conceptually from each other with mapping functions. *Logical Data Independence* exists between the Problem and the File Layer, and *Physical Data Independence* exists between the File and Data Layer analogous to the notation in database systems.

XDGDL distinguishes between pure semantic information, which comprises the Problem and File Layer and distribution information describing the Data Layer. The semantic information is made up of the following expressive modules, which are defined by the respective DTD:

- The data is distributed onto the different processor grids. These processor grids are described by an optional name, the number of dimensions of the grid, and the extent of each dimension.
- The data types are stored in the *Logical File*. This section consists of the types which are stored in the Logical File, and in the same order as they are stored in the Logical File.
- Align information as in HPF describes a recommendation how data should be distributed onto processors/nodes.

Distribution information is organized in a similar way as it is done in ▶*MPI-IO*. *Blocks* specify regions in the Logical File. A *Physical File* is the same as the byte order within a block. A Physical File is characterized by a sequence of such blocks. The byte order in the Physical File is the same as the byte order within a block. After the bytes of a block the bytes of the next block follow.

Chapter 5

Abbreviations

ACL	Access Control List
ADFS	Active Disk-Based File System
ADIO	Abstract Device Interface for I/O
ADR	Active Data Repository
AFI	Abstract File Interface
AFS	Andrew File System
AJO	Abstract Job Object
API	Application Program Interface
ASIC	Application-Specific Integrated Circuit
ASP	Application Service Provider
ASU	Active Storage Unit
ATA	Advanced Technology Attachment
BAFS	Bulldog Abstract File System
BSD	Berkeley Software Distribution
CASTOR	CERN Advanced Storage Manager
CCTK	Cactus Computational Toolkit
CIFS	Common Internet File System
CMS	Compact Muon Solenoid
CoG	Globus Commodity Grid Toolkit
CORBA	Common Object Resource Broker Architecture
CPU	Central Processing Unit
DAFS	Direct Access File System
DAP	Data Access Protocol
DARC	Distributed Active Resource Architecture

DCE	Date Circuit-Terminating Equipment
DDS	Distributed Data Structure
DFDL	Data Format Description Language
DFS	Distributed File System
DGRA	Data Grid Reference Architecture
DICE	NPACI Data-Intensive Computing Environments Infrastructure
DIOM	Distributed I/O Management
DMA	Direct Memory Access
DMAP	Data Management Application
DODS	Distributed Oceanographic Data System
DPFS	Distributed Parallel File System
DPSS	Distributed-Parallel Storage System
dQUOB	dynamic QUery OBject
DRA	Disk Resident Arrays
DRM	Disk Resource Manager
DSM	Distributed Shared Memory
DTD	Document Type Definition
EDG	European DataGrid
EEPROM	Electrically Erasable Programmable Read-Only Memory
ERC	Eager Release Consistency
ESG-II	Earth System Grid
FIFO	First IN First OUT
FTC	File Transfer Component
FTP	File Transfer Protocol
GASS	Global Access to Secondary Storage
GAT	Grid Application Toolkit
GC	Grid Console
GCE	Grid Computing Environment
GCS	Grid Consistency Service
GDMP	Grid Data Management Pilot
GDSF	Grid Data Service Factory
GEM	Globus Executable Management
Gfarm	Grid Datafarm
GFS	Global File System
GGF	Global Grid Forum
Giggle	Giga-Scale Global Location Engine

GIS	Geographic Information System
GPFS	General Parallel File System
GRAM	Grid Resource Access and Management
GRIP	Grid Interoperability Project
GriPhyN	Grid Physics Network
GSH	Grid Service Handle
GSI	Grid Security Infrastructure
GSR	Grid Service Reference
GT2	Globus Toolkit version 2
GT3	Globus Toolkit version 3
GUI	Graphical User Interface
HDF	Hierarchical Data Format
HDF5	Hierarchical Data Format 5
HENP	High Energy and Nuclear Physics
HEP	High Energy Physics
HPC	High Performance Computing
HPF	High Performance Fortran
HPSS	High Performance Storage System
HSM	Hierarchical Storage Management
HSS	Hierarchical Storage System
HTC	High Throughput Computing
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines
IDE	Integrated Development Environment
IETF	Internet Engineering Task Force
IOD	I/O Daemon
IP	Internet Protocol
IPI	Information Programming Interface
JDBC	Java Database Connectivity
LAN	Local Area Network
LDAP	Lightweight Directory Access Protocol
LFN	Logical File Name
LHC	Large Hadron Collider
LOID	Legion Object Identifier
LRC	Local Replica Catalog
LRU	Least-Recently-Used

LUN	Logical Unit
MAPFS	Multi-Agent Parallel File System
MCAT	Metadata Catalog
MDS	Maximal Distance Separating
MDS	Meta-Data Server
MDS	Metacomputing Directory Service
MIPS	Millions of Instructions per Second
MOCHA	Middleware Based On a Code SHipping Architecture
MOPI	MOSIX Scalable Parallel Input/Output
MPI	Message Passing Interface
MPI-IO	Message Passing Interface – Input/Output
MSS	Mass Storage System
MTU	Maximum Transmission Unit
NAL	Network Abstraction Layer
NARA	National Archives and Records Administration
NAS	Network Attached Storage
NASD	Network Attached Secure Disks
NCDC	National Climatic Data Center
NCSA	National Center for Supercomputing Applications
NeST	Network Storage
netCDF	Network Common Data Form
NetRPC	NetRemote Procedure Call
NFS	Network File System
NIC	Network Interface Card
NOW	Network of Workstations
OBD	Object-Based Disk
ODBMS	Object-Oriented Database Management System
OGSA	Open Grid Services Architecture
OGSA-DAI	OGSA - Database Access and Integration
OGSI	Open Grid Services Infrastructure
OID	Object Identifier
OO	Object-Oriented
OOC	Out-Of-Core
ORB	Object Request Broker
OS	Operating System
OST	Object Storage Targets

PASSION	Parallel And Scalable Software for Input-Output
PDF	Problem Description File
PDM	Parallel Disk Model
PET	Position Emission Tomography
PFN	Physical File Name
PFS	Parallel File System
PKI	Public Key Infrastructure
POSIX	Portable Operating System Interface for UNIX
PPDG	Particle Physics Data Grid
PPFS II	Portable Parallel File System
PSE	Problem Solving Environment
PVDG	Petascale Virtual Data Grid
PVFS1	Parallel Virtual File System 1
PVFS2	Parallel Virtual File System 2
RAID	Redundant Array of Independent Disks, Redundant Array of Inexpensive Disks
RBF	Radial Basis Function
RDBMS	Relational Database Management System
RFT	Reliable File Transfer
RIO	Remote I/O
RLI	Replica Location Index
RLS	Replica Location Service
RPC	Remote Procedure Call
RTE	Run-Time Environment
SAN	Storage Area Network
SCI	Scalable Coherent Interface
SCSI	Small Computer System Interface
SDDS	Scalable Distributed Data Structure
SDI	Single Disk Image
SDK	Software Developement Kit
SDM	Scientific Data Manager
SDSC	San Diego Supercomputer Center
SFIO	Striped File I/O
SGI	Silicion Graphics, Inc.
SMP	Symmetric Multiprocessor
SNMP	Simple Network Management Protocol
SOAP	Simple Object Access Protocol

SP	Service Provider
SQL	Structured Query Language
SRB	Storage Resource Broker
SRB	Storage Request Broker
SRM	Storage Resource Manager
SSL	Secure Sockets Layer
STACS	Storage Access Coordination System
SVR4	System V Release 4
Tcl	Tool Command Language
TCP/IP	Transmission Control Protocol / Internet Protocol
TIP	Transparent Informed Prefetching and Caching
TPIE	Transparent Parallel I/O Environment
TRM	Tape Resource Manager
TSS	Temperature Sensitive Storage
UCSD	University of California, San Diego
UNC	Universal Naming Convention
UNICORE	Uniform Interface to Computing Resources
UPL	UNICORE Protocol Layer
URL	Uniform Resource Locator
VDG	Virtual Data Grid
VDT	Virtual Data Toolkit
VFL	Virtual File Layer
VFS	Virtual File System
VI	Virtual Interface
VO	Virtual Organization
WALDO	Wide-Area Large Data Object Architecture
WAN	Wide-Area Network
WebDAV	Web-Based Distributed Authoring and Versioning
WiND	Wisconsin Network Disks
WSDL	Web Service Description Language
WSIF	Web Services Invocation Framework
WWW	World Wide Web
XMAS	XML Matching and Structuring Language
XML	eXtensible Markup Language

Bibliography

[1] ABACUS: Architecture of an Object-Based Distributed Filesystem Built for Abacus . `http://www.pdl.cmu.edu/Abacus/index.html`.

[2] Michael Aderholz and et al. Models of Networked Analysis at Regional Centres for LHC Experiments (MONARC), June 1999.

[3] Asmara Afework, Michael Beynon, Fabian Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman, and Hubert Tsang. Digital Dynamic Telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*, Orlando, Florida, USA, 1–7 November 1998.

[4] Abdollah A. Afjeh, Patrick T. Homert, Henry Lewandowski, John A. Reed, and Richard D. Schlichting. Development of an Intelligent Monitoring and Control System for Heterogeneous Numerical Propulsion System Simulation. In *Proceedings of the 28th Annual Simulation Symposium*, pages 278–287, Santa Barbara, California, USA, 25–28 April 1995.

[5] Bill Allcock, Joe Bester, John Bresnahan, Ann Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steve Tuecke. Secure, Efficient Data Transport and Replica Management for High-Performance Data-Intensive Computing. In *IEEE Mass Storage Conference*, 2001.

[6] Bill Allcock, Joe Bester, John Bresnahan, Ann Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steve Tuecke. Data Management and Transfer in High-Performance Computational Grid Environments. *Parallel Computing*, 28(5), 2002.

[7] Bill Allcock, John Bresnahan, Lee Liming Ian Foster, Joe Link, and Pawel Plaszczac. GridFTP Update January 2002. Technical report, http://www.globus.org, 2002.

[8] Bill Allcock, Ian Foster, Veronika Nefedova, Ann Chervenak, Ewa Deelman, Carl Kesselman, Jason Lee, Alex Sim, Arie Shoshani, Bob Drach, and Dean Williams. High-Performance Remote Access to Climate Simulation Data: A Challenge Problem for Data Grid Technologies. In *Proceeding of the IEEE Supercomputing 2001 Conference*, November 2001.

[9] Bill Allcock, Lee Liming, Steve Tuecke, and Ann Chervenak. GridFTP: A Data Transfer Protocol for the Grid. Technical report, The Globus Project.

[10] Gabrielle Allen, Werner Benger, Thomas Dramlitsch, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, André Merzky, Thomas Radke, and Edward Seidel. Cactus Grid Computing: Review of Current Development. *Parallel Processing: 7th International Euro-Par Conference, UK August 28-31*, 2150:817, January 2001.

[11] Gabrielle Allen, Werner Benger, Thomas Dramlitsch, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, André Merzky, Thomas Radke, Edward Seidel, and John Shalf. Cactus Tools for Grid Applications. *Cluster Computing*, 4(3):179–188, 2001.

[12] Gabrielle Allen, Werner Benger, Tom Goodale, Hans-Christian Hege, Gerd Lanfermann, André Merzky, Thomas Radke, Edward Seidel, and John Shalf. The Cactus Code: A Problem Solving Environment for the Grid. In *HPDC*, pages 253–260, 2000.

[13] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, André Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. Enabling Applications on the Grid: A GridLab Overview . *International Journal of High Performance Computing Applications: Special issue on Grid Computing: Infrastructure and Applications*, August 2003.

[14] Gabrielle Allen, Thomas Dramlitsch, Ian Foster, Tom Goodale, Nick Karonis, Matei Ripeanu, Ed Seidel, and Brian Toonen. Cactus-G Toolkit: Supporting Efficient Execution in Heterogeneous Distributed Computing Environments. In *Supercomputing 2000 Proceedings of 4th Globus Retreat*, Pittsburgh, USA, July 30 - August 1 2000.

[15] Gabrielle Allen, Tom Goodale, Joan Massó, and Edward Seidel. The Cactus Computational Toolkit and Using Distributed Computing to Collide Neutron Stars. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 57–61, Redondo Beach, CA, USA, August 1999. IEEE Computer Society Press.

[16] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An Automated Resource Provisioning Tool for Large-Scale Storage Systems. *ACM Transactions on Computer Systems*, 19(4):483–518, November 2001.

[17] Lior Amar, Amnon Barak, and Amnon Shiloh. The MOSIX Parallel I/O System for Scalable I/O Performance. In *Proceedings of the 14-th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 495–500, Cambridge, MA, USA, November 2002.

[18] Lior Amar, Amnon Barak, and Amnon Shiloh. The MOSIX Direct File System Access Method for Supporting Scalable Cluster File Systems , March 2003.

[19] Kaizar Amin, Sandeep Nijsure, and Gregor von Laszewski. Open Collaborative Grid Service Architecture (OCGSA). Technical Report Preprint ANL/MCS-P840-0800, Mathematics and Computer Science Division, Argonne National Laboratory, November 2002.

[20] Yair Amir and Jonathan Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS-98-4, Johns Hopkins University, Center for Networking and Distributed Systems, 1998.

[21] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic Function Placement for Data-Intensive Cluster Computing. In *Proceedings of the USENIX Annual Technical Conference*, pages 307–322, San Diego, CA, USA, June 2000.

[22] Darrell Anderson and Jeff Chase. Failure-Atomic File Access in the Slice Interposed Network Storage System . *Cluster Computing*, 5(1), 2002.

[23] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an Experiment in Public-Resource Computing . *CACM 45*, 11:56–61, 2002.

[24] James Annis, Yong Zhao, Jens Vöckler, Michael Wilde, Steve Kent, and Ian Foster. Applying Chimera Virtual Data Concepts to Cluster Finding in the Sloan Sky Survey. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, 2002.

[25] Apache. AXIS: WebServices . `http://xml.apache.org/axis/`.

[26] Apache. Tomcat . `http://jakarta.apache.org/tomcat/`.

[27] Lars Arge. External Memory Data Structures. *Lecture Notes in Computer Science*, 2161:1–46, 2001.

[28] Lars Arge, Octavian Procopiuc, and Jeffrey Scott Vitter. Implementing I/O-Efficient Data Structures Using TPIE . In *Proceedings of the 10th European Symposium on Algorithms (ESA '02)*, Rom, Italy, September 2002.

[29] ARM . `http://www.arm.com/`.

[30] Dorian C. Arnold and Jack Dongarra. The NetSolve Environment: Progressing Towards the Seamless Grid. In *2000 International Conference on Parallel Processing (ICPP-2000)*, Toronto, Canada, August 2000.

[31] Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, John Bent, Brian Forny, Sambavi Muthukrishnan, Florentina Popovici, and Omer Zaki. Manageable Storage via Adaptation in WiND. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid (CCGRID '01)*, 15–18 May 2001.

[32] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David Patterson, and Kathy Yelick. Cluster I/O with River: Making the Fast Case Common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, May 1999. ACM Press.

[33] Bill Asbury, Geoffrey Fox, Tom Haupt, and Ken Flurchick. The Gateway Project: An Interoperable Problem Solving Environments Framework For High Performance Computing . In *DoD HPC Users Group Conference*, June 1999.

[34] Malcolm P. Atkinson, Rob Baxter, and Neil Chue Hong. Grid Data Access and Integration in OGSA. Technical report, Documents Produced for GGF5.

[35] Malcolm P. Atkinson, Vijay Dialani, Leanne Guy, Inderpal Narang, Norman W. Paton, Dave Pearson, Tony Storey, and Paul Watson. Grid Database Access and Integration: Requirements and Functionalities. Technical report, Documents Produced for GGF7.

[36] ATLAS: the ATLAS Experiment for the Large Hadron Collider. `http://pdg.lbl.gov/atlas/atlas.html`.

[37] Automatic GASS. `http://www.cs.wisc.edu/condor/bypass/examples/automatic-gass/`.

[38] Autopilot: Real-Time Adaptive Resource Control . `http://www-pablo.cs.uiuc.edu/Project/Autopilot/AutopilotOverview.htm`.

[39] AVO: Astrophysical Virtual Observatory . `http://www.euro-vo.org/`.

[40] Jon Bakken, Eileen Berman, Chih-Hao Huang, Alexander Moibenko, Don Petravick, Ron Rechenmacher, and Kurt Ruthmansdorfer. Enstore Technical Design Document. Technical Report JP0026, Fermi National Accelerator Laboratory, 1999.

[41] Donald Bales. *Java Programming with Oracle JDBC*. O'Reilly, December 2001.

[42] Amnon Barak and Avner Braverman. Memory Ushering in a Scalable Computing Cluster, 1997.

[43] Amnon Barak, Shai Guday, and Richard G. Wheeler. *MOSIX Distributed Operating System, Load Balancing for UNIX* , volume 672 of *Lecture Notes in Computer Science*. Springer, 1996.

[44] Amnon Barak, Oren La'adan, and Amnon Shiloh. Scalable Cluster Computing with MOSIX for LINUX, 1999.

[45] Andrew Baranovski, Diana Bonham, Gabriele Garzoglio, Chris Jozwiak, Lauri Loebel Carpenter, Lee Lueking, Carmenita Moore, Ruth Pordes, Heidi Schellman, Igor Terekhov, Matthew Vranicar, Sinisa Veseli, Stephen White, and Victoria White. SAM Managed Cache and Processing for Clusters in a Worldwide Grid-Enabled System. Technical Report FERMILAB-TM-2175, Fermilab, July 2002.

[46] Andrew Baranovski, Gabriele Garzoglio, Hannu Koutaniemi, Lee Lueking, Siddharth Patil, Ruth Pordes, Abhishek Rana, Igor Terekhov, Sinisa Veseli, Jae Yu,

Rod Walker, and Vicky White. The SAM-GRID Project: Architecture and Plan. In *ACAT 02*, Moscow, Russia, June 2003.

[47] Andrew Baranovski, Gabriele Garzoglio, Lee Lueking, Dane Skow, Igor Terekhov, and Rodney Walker. SAM-GRID: A System Utilizing Grid Middleware and SAM to Enable Full Function Grid Computing. In *Beauty 02*, Santiago de Compostela, Spain, June 2002.

[48] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, and Michael Wan. The SDSC Storage Resource Broker . In *CASCON'98 Conference*, Toronto, Canada, 30 November–3 December 1998.

[49] Jean-Philippe Baud. Castor Architecture. Technical report, CERN, October 2002.

[50] Bay and Estuary Simulation . `http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/bay.html`.

[51] Keith Bell, Andrew Chien, and Mario Lauria. A High-Performance Cluster Storage Server . In *The 11th International Symposium on High Performance Distributed Computing (HPDC-11)*, Edinburgh, Scotland, 24–26 July 2002.

[52] William H. Bell, Diana Bosio, Wolfgang Hoschek, Peter Kunszt, Gavin McCance, and Mika Silander. Project Spitfire - Towards Grid Web Service Databases . In *Global Grid Forum 5*, Edinburgh, Scotland, July 2002.

[53] András Belokosztolszki. An XML based Language for Meta Information in Distributed File Systems. Master's thesis, University of Vienna and Eotvos Lorand University of Science, Budapest, June 2000.

[54] András Belokosztolszki and Erich Schikuta. An XML based Framework for Self-Describing Parallel I/O Data. In *Proceedings of the 11th Euromicro Workshop on Parallel and Distributed Processing PDP 2003*, Genova, Italy, February 2003. IEEE Computer Society Press.

[55] John Bent, Venkateshwaran Venkataramani, Nick LeRoy, Alain Roy, Joseph Stanley, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. NeST - A Grid Enabled Storage Appliance. In Jan Weglarz, Jarek Nabrzyski, Jennifer Schopf, and Macief Stroinkski, editors, *Grid Resource Management*. Kluwer Academic Publishers, 2003.

[56] John Bent, Venkateshwaran Venkataramani, Nick LeRoy, Alain Roy, Joseph Stanley, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Flexibility, Manageability, and Performance in a Grid Storage Appliance. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC-11)*, Edinburgh, Scotland, July 2002.

[57] Luis M. Bernardo, Arie Shoshani, Alexander Sim, and Henrik Nordberg. Access Coordination of Tertiary Storage For High Energy Physics Applications. In *IEEE Symposium on Mass Storage Systems*, College park, MD, USA, March 2000.

[58] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steve Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 78–88, Atlanta, GA, USA, May 1999. ACM Press.

[59] Michael Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, and Joel Saltz. DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems. In *Proceedings of the 8th Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, USA, 27–30 March 2000.

[60] Michael Beynon, Tahsin Kurc, Alan Sussman, and Joel Saltz. Design of a Framework for Data-Intensive Wide-Area Applications. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000)*, pages 116–130. IEEE Computer Society Press, May 2000.

[61] Bio-GRID . `http://www.eurogrid.org/wp1.html`.

[62] Dan Bonachea, Phillip Dickens, and Rajeev Thakur. High-Performance File I/O in Java: Existing Approaches and Bulk I/O Extensions. Technical Report ANL/MCS-P840-0800, Mathematics and Computer Science Division, Argonne National Laboratory, August 2000.

[63] Rajesh Bordawekar. Implementation of Collective I/O in the Intel Paragon Parallel File System: Initial Experiences. In *Proceedings of the 11th ACM International Conference on Supercomputing*, pages 20–27. ACM Press, July 1997.

[64] Peter J. Braam. The Lustre Storage Architecture. Technical report, http://www.lustre.org, 2002.

[65] Peter Brezany and Marianne Winslett. Parallel Access to Persistent Multidimensional Arrays from HPF Applications Using Panda. In *Proceedings of the Eighth International Conference on High Performance Computing and Networking*, Amsterdam, Netherlands, April 2000.

[66] Bradley Broom, Rob Fowler, and Ken Kennedy. KelpIO: A Telescope-Ready Domain-Specific I/O Library for Irregular Block-Structured Applications. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 148–155, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[67] Julian J. Bunn and Harvey B. Newman. Data-Intensive Grids for High-Energy Physics. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, chapter 39. John Wiley & Sons Inc., December 2002.

[68] Randy Butler, Von Welch, Douglas Engert, Ian Foster, Steve Tuecke, John Volmer, and Carl Kesselman. A National-Scale Authentication Infrastructure. *Computer*, 33(12):60–66, December 2000.

[69] Bypass. `http://www.cs.wisc.edu/condor/bypass/`.

[70] CADRE: A National Facility for I/O Characterization and Optimization . `http://www-pablo.cs.uiuc.edu/Project/CADRE/index.htm`.

[71] Philip H. Carns and Walter B. Ligon III. Parallel Virtual File System Version 2. Technical report, poster presentation from 2002 NPACI meeting, 2002.

[72] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[73] Henri Casanova and Jack Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[74] Henri Casanova and Jack Dongarra. NetSolve: A Network Enabled Server, Examples and Users . In *Proceedings of the Heterogeneous Computing Workshop*, Orlando, Florida, 1998.

[75] Henri Casanova, Jack Dongarra, and Keith Moore. Network-Enabled Solvers and the NetSolve Project. *SIAM News*, 31(1), January 1998.

[76] Centurion: Legion Project Testbed. `http://legion.virginia.edu/centurion/Applications.html`.

[77] CERN: Conseil Européen pour la Recherche Nucléaire. `http://www.cern.ch`.

[78] Chialin Chang, Renato Ferreira, Alan Sussman, and Joel Saltz. Infrastructure for Building Parallel Database Systems for Multi-Dimensional Data. In *Proceedings of the Second Merged IPPS/SPDP Symposium*. IEEE Computer Society Press, April 1999.

[79] Chaos Project. `http://www.cs.umd.edu/projects/hpsl/chaos/`.

[80] CHARISMA: CHARacterize I/O in Scientific Multiprocessor Applications . `http://www.cs.dartmouth.edu/~dfk/charisma/`.

[81] Jeffrey S. Chase, Darrell C. Anderson, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Network I/O with Trapeze. In *Hot Interconnects Symposium*, August 1999.

[82] Jeffrey S. Chase, Richard Kisley, Andrew J. Gallatin, and Darrell C. Anderson. DAFS Demo White Paper. In *DAFS Collaborative Developer's Conference*, June 2001.

[83] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[84] Ying Chen and Marianne Winslett. Automated Tuning of Parallel I/O Systems: An Approach to Portable I/O Performance for Scientific Applications. *IEEE Transactions on Software Engineering*, 26(4):362–383, April 2000.

[85] Ying Chen, Marianne Winslett, Yong Cho, and Szu-Wen Kuo. Automatic Parallel I/O Performance Optimization in Panda. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 108–118, June 1998.

[86] Ying Chen, Marianne Winslett, Szu-Wen Kuo, Yong Cho, Mahesh Subramaniam, and Kent E. Seamons. Performance Modeling for the Panda Array I/O Library. In

Proceedings of Supercomputing '96. ACM Press and IEEE Computer Society Press, November 1996.

[87] Ying Chen, Marianne Winslett, Kent E. Seamons, Szu-Wen Kuo, Yong Cho, and Mahesh Subramaniam. Scalable Message Passing in Panda. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems*, pages 109–121, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.

[88] Ann Chervenak, Ewa Deelmann, Ian Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselmann, Peter Kunszt, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggle: A Framework for Constructing Scalable Replica Location Services. In *In Porceedings of the IEEE Supercomputing Conference 2002*, November 2002.

[89] Andrew Chien, Brad Calder, Stephen Elbert, and Karan Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel Distributed Computing*, 2003.

[90] Chimera Virtual Data System. `http://www.griphyn.org/chimera/`.

[91] Avery Ching, Alok Choudhary, Wei-Keng Liao, Rob Ross, and William Gropp. Noncontiguous I/O through PVFS. In William Gropp, Rob Pennington, Dan Reed, Mark Baker, Maxine Brown, and Rajkumar Buyya, editors, *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02)*, pages 405–414. IEEE Computer Society, 2002.

[92] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawanaa. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 2001.

[93] CIFS: Common Internet File System . `http://www.microsoft.com/mind/1196/cifs.asp`.

[94] ClassAds: Classified Advertisements . `http://www.cs.wisc.edu/condor/classad/`.

[95] CMS: Compact Muon Solenoid. `http://cmsinfo.cern.ch/Welcome.html/`.

[96] Toni Cortes and Jesus Labarta. Taking Advantage of Heterogeneity in Disk Arrays. *Journal of Parallel and Distributed Computing*, 63(4):448–464, April 2003.

[97] Robert A. Coyne, Harry Hulen, and Richard Watson. The High Performance Storage System. In *Proceedings of Supercomputing '93*, pages 83–92, Portland, OR, 1993. IEEE Computer Society Press.

[98] Andreas Crauser. *LEDA-SM: External Memory Algorithms and Data Structures in Theory and Practice*. PhD thesis, University of Saarland, 2001.

[99] Andreas Crauser and Kurt Mehlhorn. LEDA-SM: Extending LEDA to Secondary Memory. *Lecture Notes in Computer Science*, 1668:228–242, 1999.

[100] CrossGrid. CrossGrid. Annex 1 - Description of Work. Technical Report 3.1, www.crossgrid.org, 2002.

[101] CSAR: Center for Simulation of Advanced Rockets. `http://www.csar.uiuc.edu`.

[102] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steve Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proceedings of the IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[103] Joseph Czyzyk, Michael Mesnier, and Jorge J. More. NEOS: The Network-Enabled Optimization System. Technical Report Technical Report MCS-P615-1096, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[104] Michael Dahlin, Randolph Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Operating Systems Design and Implementation*, pages 267–280, 1994.

[105] DAMIEN: Distributed Applications and Middleware for Industrial Use of European Networks . `http://www.hlrs.de/organization/pds/projects/damien/`.

[106] DataTAG: Project Description. `http://datatag.web.cern.ch/datatag/project.html`.

[107] Harvey L. Davies. FAN - An Array-Oriented Query Language. In *Second Workshop on Database Issues for Data Visualization (Visualization '95)*, Atlanta, Georgia, USA, 1995. IEEE.

[108] Matt DeBergalis, Peter Corbett, Steve Kleiman, Arthur Lent, Dave Noveck, Tom Talpey, and Mark Wittle. The Direct Access File System. In *USENIX File and Storage Technology (FAST) Conference*, San Francisco, CA, USA, April 2003.

[109] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved Parallel I/O via a Two-Phase Run-Time Access Strategy. In *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993.

[110] Roman Dementiev and Peter Sanders. Asynchronous Parallel Disk Sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, San Diego, 2003.

[111] DFDL: Data Format Description Language . `https://forge.gridforum.org/projects/dfdl-wg/`.

[112] Phillip M. Dickens and Rajeev Thakur. An Evaluation of Java's I/O Capabilities for High-Performance Computing. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 26–35. ACM Press, June 2000.

[113] Chris H.Q. Ding and Yun He. Data Organization and I/O in a Parallel Ocean Circulation Model. In *Proceedings of SC99: High Performance Networking and Computing*, Portland, OR, November 1999. ACM Press and IEEE Computer Society Press.

[114] Direct I/O. Oracle Internals Notes . `http://www.ixora.com.au/notes/direct_io.htm`.

[115] DODS: Architecture . `http://www.unidata.ucar.edu/packages/dods/home/getStarted/architecture.html`.

[116] Dirk Düllmann, Wolfgang Hoschek, Javier Jaen-Martinez, Ben Segal, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid. In *10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10)*, San Francisco, California, 7–9 August 2001.

[117] Dzero . `http://www-d0.fnal.gov/`.

[118] EDG Workpackages . `http://web.datagrid.cnr.it/servlet/page?_pageid=1429&_dad=portal30&_schema=P%ORTAL30&_mode=3`.

[119] Chris Elford, Jay Huber, Chris Kuszmaul, and Tara Madhyastha. Portable Parallel File System Detailed Design. Technical report, University of Illinois at Urbana-Champaign, USA, November 1993.

[120] Chris Elford, Jay Huber, Chris Kuszmaul, and Tara Madhyastha. PPFS High Level Design Documentation. Technical report, University of Illinois at Urbana-Champaign, USA, November 1993.

[121] EMC Corporation . `http://www.emc.com`.

[122] Dietmar Erwin. UNICORE Plus Final Report - Uniform Interface to Computing Resources. Technical report, Jülich, Forschungszentrum, Germany, 2003.

[123] ESG-II: ESG-II Architectural Specification. `http://sdm.lbl.gov/esg/esg-2-arch-spec.php`.

[124] ESG-II: Functional Specification for ESG-II. `http://sdm.lbl.gov/esg/esg-2-func-spec.php`.

[125] ESGO: European Grid of Solar Observations . `http://www.egso.org/`.

[126] EuroGrid. Eurogrid: European Testbed for GRID Applications. Technical report, http://www.eurogrid.org.

[127] Dror G. Feitelson and Tomer Klainer. XML, Hyper-Media, and Fortran I/O. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 43, pages 633–644. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[128] Rene Felder. ViPFS: An XML Based Distributed File System for Cluster Architecture . Master's thesis, University of Vienna, November 2001.

[129] Renato Ferreira, Tahsin Kurc, Michael Beynon, Chialin Chang, Alan Sussman, and Joel Saltz. Object-Relational Queries into Multi-dimensional Databases with the Active Data Repository. *Parallel Processing Letters*, 9(2):173–195, 1999.

[130] Stephen J. Fink, Scott R. Kohn, and Scott B. Baden. Efficient Run-Time Support for Irregular Block-Structured Applications . *Journal of Parallel and Distributed Computing*, 50(12):618, April/May 1998.

[131] Fleet Numerical Meteorology and Oceanography Center. `http://www.fnoc.navy.mil`.

[132] Ian Foster. Grid Technologies & Applications: Architecture & Achievements. Technical report, GriPhyN Technical Report, http://www.griphyn.org, 2001.

[133] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[134] Ian Foster and Carl Kesselman. A Data Grid Reference Architecture. Technical report, http://www.griphyn.org, February 2001.

[135] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steve Tuecke. Grid Services for Distributed System Integration. *Computer*, 35(6), 2002.

[136] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Open Grid Service Infrastructure WG, http://www.ggf.org, June 2002.

[137] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *High Performance Computing Applications*, 15(3):200–222, 2001.

[138] Ian Foster, David Kohr, Jr., Rakesh Krishnaiyer, and Jace Mogill. Remote I/O: Fast Access to Distant Storage. In *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pages 14–25, San Jose, CA, USA, November 1997. ACM Press.

[139] Ian Foster and Jarek Nieplocha. ChemIO: High-Performance I/O for Computational Chemistry Applications. http://www.mcs.anl.gov/chemio/, February 1996.

[140] Ian Foster and Jarek Nieplocha. Disk Resident Arrays: An Array-Oriented I/O Library for Out-of-Core Computations. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 33, pages 488–498. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[141] Ian Foster, Jens Vöckler, Michael Wilde, and Yong Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management (SSDBM02)*, page 37ff., July 2002.

[142] Ian Foster, Jens Vöckler, Michael Wilde, and Yong Zhao. The Virtual Data Grid: A New Model and Architecture for Data Intensive Collaboration. In *Proceedings*

of th 1st Biennial Conference on Innovative Data Systems Research (CIDR 2003), January 2003.

[143] FreeBSD. `http://www.freebsd.org/`.

[144] Thomas Fuerle, Oliver Jorns, Erich Schikuta, and Helmut Wanek. Meta-ViPIOS: Harness Distributed I/O Resources with ViPIOS. *Iberoamerican Journal of Research ”Computing and Systems”, Special Issue on Parallel Computing*, 4(2):124–142, October-December 2000.

[145] Emin Gabrielyan. SFIO, Parallel File Striping for MPI-I/O . *EPFL-SCR*, 12, November 2000.

[146] Emin Gabrielyan and Roger D. Hersch. SFIO, a Striped File I/O Library for MPI . In *http://storageconference.com*, 2001.

[147] Felix García, Alejandaro Calderón, Jesus Carretero, Jesus Pérez, and Javier Fernández. A Parallel and Fault Tolerant File System Based on NFS Servers. In *Proceedings of the Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 83–92, February 2003.

[148] Felix García, Alejandaro Calderón, Jesus Carretero, Jesus Pérez, and Javier Fernández. The Design of the Expand Parallel File System. *International Journal of High Performance Computing Applications*, 17:21–37, 2003.

[149] Joydeep Ghosh and Arindam Nag. An Overview of Radial Basis Function Networks . *Radial Basis Function Neural Network Theory and Applications*, 2000.

[150] Francesco Giacomini, Francesco Prelz, Massimo Sgaravatto, Igor Terekhov, Gabriele Garzoglio, and Todd Tannenbaum. Planning on the Grid: A Status Report [DRAFT]. Technical Report PPDG-20, Particle Physics Data Grid collaboration, http://www.ppdg.net, October 2002.

[151] Garth A. Gibson, David Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File Server Scaling with Network-Attached Secure Disks. In *Measurement and Modeling of Computer Systems*, pages 272–284, 1997.

[152] Global Grid Forum. `http://www.gridforum.org`.

[153] Globus. Globus Toolkit 2.2 MDS Technology Brief . Technical Report Draft 4, http://www.globus.org, January 2003.

[154] Globus I/O Functions: globus_io . `http://www-unix.globus.org/api/c/globus_io/html/main.html`.

[155] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Masso, Thomas Radke, Edward Seidel, and John Shalf. The Cactus Framework and Toolkit: Design and Applications. In *Vector and Parallel Processing - VECPAR'2002, 5th International Conference*, Lecture Notes in Computer Science, 2002.

[156] K. Gopinath, Nitin Muppalaneni, N. Suresh Kumar, and Pankaj Risbood. A 3-Tier RAID Storage System with RAID1, RAID5, and Compressed RAID5 for Linux. In *Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference*, 18 – 23 June 2000.

[157] Otis Graf. Basics of the High Performance Storage System . `http://www4.clearlake.ibm.com/hpss/about/HPSS-Basics.pdf`.

[158] Steve Graham, Simon Simeonov, Toufic Boubez, Glen Daniels, Doug Davis, Yuichi Nakamura, and Ryo Nyeama. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL, and UDDI* . SAMS, 2001.

[159] GRAM: Globus Resource Allocation Manager . `http://www-unix.globus.org/developer/resource-management.html`.

[160] GRIA: GRID for Business and Industry . `http://www.gria.org`.

[161] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, 2000.

[162] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David F. Nagle. Modeling and Performance of MEMS-Based Storage Devices. In *Proceedings of ACM SIGMETRICS 2000*, Santa Clara, California, USA, June 2000.

[163] Andrew Grimshaw, Adam Ferrari, Fritz Knabe, and Marty Humphrey. Legion: An Operating System for Wide-Area Computing. *IEEE Computer*, 32(5):29–37, May 1999.

[164] GRIP: GRid Interoperability Project . `http://www.grid-interoperability.org`.

[165] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[166] GSI: Grid Security Infrastructure . `http://www.globus.org/security/`.

[167] GSI-SFS. `http://www.biogrid.jp/e/research_work/gro1/gsi_sfs/index.html`.

[168] GUIDE Grid Portal. `http://www.biogrid.jp/e/research_work/gro1/guide/index.html`.

[169] Leanne Guy, Peter Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. Replica Management in Data Grids. Technical report, CERN, European Organization for Nuclear Research, July 2002.

[170] Mehnaz Hafeez, Asad Samar, and Heinz Stockinger. A Data Grid Prototype for Distributed Data Production in CMS. In *VII International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT2000)*, October 2000.

[171] Steve Hammond. Prototyping an Earth System Grid. In *Workshop on Advanced Networking Infrastructure Needs in Atmospheric and Related Sciences*, National Center for Atmospheric Research, Boulder, CO, USA, June 1999.

[172] Bruce Hendrickson and Robert Leland. The Chaco User's Guide: Version 2.0. Technical Report SAND94-2692, Sandia National Laboratories, 1994.

[173] Bill Hibbard. VisAD: Connecting People to Computations and People to People. *ACM SIGGRAPH Computer Graphics*, 32(3):10–12, August 1998.

[174] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[175] Koen Holtman and Heinz Stockinger. Building a Large Location Table to Find Replicas of Physics Objects. *Journal of Computer Physics Communications*, 140:146–152, 2001.

[176] Neil P. Chue Hong, Amy Krause, Simon Laws, Susan Malaika, Gavin McCance, James Magowan, Norman W. Paton, and Greg Riccardi. Grid Database Service Specification. Technical report, Documents Produced for GGF7.

[177] Hans-Christian Hoppe and Daniel Mallmann. EUROGRID - European Testbed for GRID Applications . In *GRIDSTART Technical Bulletin*, October 2002.

[178] James V. Huber, Jr., Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A High Performance Portable Parallel File System. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 22, pages 330–343. IEEE Computer Society Press and Wiley, New York, NY, USA, 2001.

[179] Harry Hulen, Otis Graf, Keith Fitzgerald, and Richard W. Watson. Storage Area Networks and the High Performance Storage System, April 15 - 18 2002.

[180] David A. Hutchinson, Peter Sanders, and Jeffrey Scott Vitter. The Power of Duality for Prefetching and Sorting with Parallel Disks. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 334–335, 2001.

[181] Kai Hwang, Hai Jin, and Roy S. C. Ho. Orthogonal Striping and Mirroring in Distributed RAID for I/O-Centric Cluster Computing. In *IEEE Transactions on Parallel and Distributed Systems*, volume 13, pages 26–44, January 2002.

[182] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 2000 . `http://www.infinibandta.org`.

[183] Yannis E. Ioannidis, Miron Livny, Anastassia Ailamaki, Anand Narayanan, and Andrew Therber. Zoo: A Desktop Experiment Management Environment. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 274–285, 1996.

[184] Florin Isaila. An Overview of File System Architectures. In Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 273–289. Springer, 2003.

[185] Florin Isaila and Walter F. Tichy. Clusterfile: A Flexible Physical Layout Parallel File System. In *Proceedings of the 3rd IEEE International Conference on Cluster Computing (CLUSTER'01)*, page 37ff., 8–11 October 2001.

[186] iVDGL: international Virtual Data Grid Laboratory . http://www.ivdgl.org/.

[187] Keith R. Jackson. pyGlobus: a Python Interface to the Globus Toolkit. *Concurrency and Computation: Practice and Experience*, 14(13–15):1075–1083, 2002.

[188] Java Universe . http://www.cs.wisc.edu/condor/manual/v6.4/2_4Road_map_Running.html#SECTION00%341600000000000000.

[189] Kaiser: Medical Imaging . http://www-itg.lbl.gov/DPSS/Kaiser/.

[190] Mahesh Kallahalla and Peter J. Varman. Analysis of Simple Randomized Buffer Management for Parallel I/O. IPL (accepted).

[191] Mahesh Kallahalla and Peter J. Varman. PC-OPT: Optimal Prefetching and Caching for Parallel I/O Systems. In *IEEE Transactions on Computers*, volume 15, pages 1333–1344, November 2002.

[192] Kerberos: The Network Authentication Protocol . http://web.mit.edu/kerberos/www/.

[193] Carl Kesselman, Randy Butler, Ian Foster, Joe Futrelle, Doru Marcusiu, Sridhar Gulipalli, Laura Pearlman, and Chuck Severance. NEESgrid System Architecture. Technical Report Version 1.1, http://www.neesgrid.org, May 2003.

[194] Gene Kim, Ronald Minnich, and Larry McVoy. A Parallel File-Striping NFS-Server. Technical report, Sun Microsystems Computer Corp., 1994.

[195] Tracy Kimbrel and Anna R. Karlin. Near-Optimal Parallel Prefetching and Caching. In *IEEE Symposium on Foundations of Computer Science*, pages 540–549, 1996.

[196] Donald Ervin Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Pub Co, April 1998.

[197] Julian Bunn Koen. The GIOD Project - Globally Interconnected Object Databases. In *Proceedings of CHEP 2000*, Padova, Italy, February 2000.

[198] David Kotz. Expanding the Potential for Disk-Directed I/O. In *Proceedings of the 1995 IEEE Symposium on Parallel and Distributed Processing*, pages 490–495, 1995.

[199] David Kotz. Disk-Directed I/O for MIMD Supercomputers. In *ACM Transactions on Computer Systems*, volume 15, pages 41–74, February 1997.

[200] Orran Krieger and Michael Stumm. HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions. *ACM Trans. Comp. Syst.*, 15(3):286–321, 1997.

[201] Peter Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. Advanced Replica Management with Reptor. In *In 5th International Conference on Parallel Processing and Applied Mathematics*, September 7–10 2003.

[202] Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman, and Joel Saltz. Exploration and Visualization of Very Large Datasets with the Active Data Repository. *IEEE Computer Graphics and Applications*, 21(4):24–33, July/August 2001.

[203] Tahsin Kurc, Chialin Chang, Renato Ferreira, and Alan Sussman. Querying Very Large Multi-Dimensional Datasets in ADR. In *Proceedings of SC99: High Performance Networking and Computing*, Portland, OR, USA, 1999. ACM Press and IEEE Computer Society Press.

[204] Mario Lauria, Keith Bell, and Andrew Chien. A High-Performance Cluster Storage Server. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing*, pages 311–320, Edinburgh, Scotland, 2002. IEEE Computer Society Press.

[205] LCG: LHC Computing Grid Project . `http://lcg.web.cern.ch/LCG/`.

[206] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-VII*, pages 84–92, October 1996.

[207] Jonghyun Lee, Xiaosong Ma, Marianne Winslett, and Shengke Yu. Active Buffering Plus Compressed Migration: an Integrated Solution to Parallel Simulations' Data Transport Needs. In *Proceedings of the 16th international conference on Supercomputing*, pages 156–166, June 2002.

[208] Jonghyun Lee, Marianne Winslett, Xiaosong Ma, and Shengke Yu. Enhancing Data Migration Performance via Parallel Data Compression. In *IPDPS 2002*, 2002.

[209] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multipro-

cessor. In *Proceedings of the 17th International Symp. on Computer Architecture*, pages 148–159, Seattle, WA, May 1990.

[210] Michael J. Lewis and Andrew Grimshaw. The Core Legion Object Model . In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, August 1996. IEEE Computer Society Press.

[211] LHC - The Large Hadron Collider. `http://public.web.cern.ch/public/about/future/whatisLHC/whatisLHC.html`.

[212] Jianwei Lia, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, and Rob Latham. Parallel netCDF: A Scientific High-Performance I/O Interface . In *SC2003*, November 2003.

[213] LIGO: Laser Interferometer Gravitational Wave Observatory. `http://www.ligo.caltech.edu/`.

[214] Hyeran Lim, Vikram Kapoor, Chirag Wighe, and David H.-C. Du. Active Disk File System: A Distributed, Scalable File System, April 2001.

[215] Ling Lio, Calton Pu, and Wei Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *Transactions on Knowledge and Data Engineering*, 11(4):610–628, July 1999.

[216] Witold Litwin, Marie-Anna Neimat, and Donovan A. Schneider. LH* - a Scalable, Distributed Data Structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[217] Witold Litwin and Thomas Schwarz. LH*RS : A High-Availability Scalable Distributed Data Structure Using Reed Solomon Codes. In *SIGMOD Conference*, pages 237–248, 2000.

[218] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[219] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Faster Collective Output through Active Buffering. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 34–41, 15–19 April 2002.

[220] Xiaosong Ma, Marianne Winslett, Jonghyun Lee, and Shengke Yu. Improving MPI-IO Output Performance with Active Buffering Plus Threads. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 68–77. IEEE Computer Society Press, 22–26 April 2003.

[221] F. J. Mac Williams and N. J. Sloane. The Theory of Error-Correcting Codes, 1977.

[222] Ravi K. Madduri, Cynthia S. Hood, and William E. Allcock. Reliable File Transfer in Grid Environments. Technical Report Preprint ANL/MCS-P983-0802, Mathematics and Computer Science Division, Argonne National Laboratory, August 2002.

[223] Tara M. Madhyastha, Christopher L. Elford, and Daniel A. Reed. Optimizing Input/Output Using Adaptive File System Policies. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, pages II:493–514, September 1996.

[224] Kostas Magoutis, Salimah Addetia, Alexandra Fedorova, Margo I. Seltzer, Jeffrey S. Chase, Andrew Gallatin, Richard Kisley, Rajiv Wickremesinghe, and Eran Gabber. Structure and Performance of the Direct Access File System. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA, USA*, pages 1–14, June 2002.

[225] MCAT: A Meta Information Catalog . `http://www.npaci.edu/DICE/SRB/mcat.html`.

[226] Kurt Mehlhorn and Stephan Näher. Leda, a Platform for Combinatorial and Geometric Computing . *Communications of the ACM*, 38:96–102, 1995.

[227] Gokhan Memik, Mahmut T. Kandemir, and Alok Choudhary. Design and Evaluation of a Compiler-Directed Collective I/O Technique. In *Proceedings of the European Conference on Parallel Computing (Euro-Par 2000)*, pages 1263–1272, August 2000.

[228] Gokhan Memik, Mahmut T. Kandemir, and Alok Choudhary. Exploiting Inter-File Access Patterns Using Multi-Collective I/O. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, January 2002.

[229] Gokhan Memik, Mahmut T. Kandemir, Alok Choudhary, and Valerie E. Taylor. APRIL: A Run-Time Library for Tape-Resident Data. In *IEEE Symposium on Mass Storage Systems*, pages 61–74, 2000.

[230] MEMS: Microelectromechanical Systems Laboratory. `http://www.ece.cmu.edu/~mems/`.

[231] Vincent Messerli. *Tools for Parallel I/O and Compute Intensive Applications*. PhD thesis, École Polytechnique Fédérale de Lausanne, 1999.

[232] Paul C. Messina, Sharon Brunett, Dan Davis, Thomas T. Gottschalk, David Curkendall, Laura Ekroot, and Peter H. Siegel. Distributed Interactive Simulation for Synthetic Forces. In *Proceedings of the 6th Heterogeneous Computing Workshop*, Geneva, Switzerland, April 1997.

[233] Microsoft. Distributed File System: A Logical View of Physical Storage. Technical report, Microsoft Corporation, 1999.

[234] Millipede: A Future AFM-Based Data Storage System . `http://www.zurich.ibm.com/st/storage/millipede.html`.

[235] Reagan W. Moore. Knowledge-Based Persistent Archives. Technical Report SDSC TR-2001-7, SDSC, January 2001.

[236] Reagan W. Moore. The San Diego Project: Persistent Objects. In *Proceedings of the Workshop on XML as a Preservation Language*, Urbino, Italy, October 2002.

[237] Reagen W. Moore. Recommendations for Standard Operations at Remote Sites. Technical report, SDSC, 2003.

[238] Reagen W. Moore and Andre Merzky. Persistent Archive Concept. Technical report, SDSC, 2003.

[239] Richard P. Mount. US Grid Projects: PPDG and iVDGL. In *CHEP 2001*, Beijing, China, September 2001.

[240] Steven A. Moyer and Vaidy S. Sunderam. A Parallel I/O System for High-Performance Distributed Computing . Technical report, Department of Mathematics and Computer Science, Emory University, Atlanta, GA, USA, January 1994.

[241] MPIF: Message Passing Interface Forum . `http://www.mpi-forum.org/`.

[242] MPICH: A Portable Implementation of MPI . `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[243] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, USA, 1996.

[244] Myrinet . `http://www.myri.com/`.

[245] Elsie Nallipogu, Füsun Özgüner, and Mario Lauria. Improving the Throughput of Remote Storage Acces Through Pipelining . In *Grid 2002*, pages 305–316, 2002.

[246] Anand Natrajan, Marty Humphrey, and Andrew Grimshaw. Grids: Harnessing Geographically-Separated Resources in a Multi-Organisational Context. Technical report, Presented at High Performance Computing Systems, June 2001.

[247] NCDC: National Climatic Data Center . `http://www.ncdc.noaa.gov`.

[248] NCSA/UIUC. Introduction to HDF5, May 2000.

[249] NetRPC: NetRemote Procedure Call . `http://www.cs.duke.edu/ari/publications/talks/freebsdcon/tsld025.htm`.

[250] Nexus: The Nexus Multithreaded Communication Library . `http://www.globus.org/nexus/`.

[251] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[252] Niels Nieuwejaar. *Galley: A New Parallel File System for Scientific Workloads*. PhD thesis, Department of Computer Science, Dartmouth College, Hanover, NH 03755-3510, USA, 1996.

[253] Ninf: A Global Computing Infrastructure . `http://ninf.apgrid.org/`.

[254] Jaechun No, Rajeev Thakur, and Alok Choudhary. High-Performance Scientific Data Management System. *Journal of Parallel and Distributed Computing*, 63(4):434–447, 2003.

[255] Jaechun No, Rajeev Thakur, Dinesh Kaushik, Lori Freitag, and Alok Choudhary. A Scientific Data Management System for Irregular Applications. In *Proc. of the Eighth International Workshop on Solving Irregular Problems in Parallel (Irregular 2001)*, April 2001.

[256] Jaechun No, Rajev Thakur, and Alok Choudhary. Integrating Parallel File I/O and Database Support for High-Performance Scientific Data Management. In *Supercomputing Conference*, Dallas, Texas, USA, November 2000.

[257] Klaus-Dieter Oertel and Mathilde Romberg. The UNICORE Grid System. In *Euro-Par*, 2002.

[258] Ron Oldfield. Summary of Existing and Developing Data Grids. White paper for the Remote Data Access group of the Global Grid Forum, March 2001.

[259] Ron Oldfield. *Efficient I/O for Computational Grid Applications*. PhD thesis, Dept. of Computer Science, Dartmouth College, May 2003. Available as Dartmouth Computer Science Technical Report TR2003-459.

[260] Ron Oldfield. High-Performance I/O for Computational Grid Applications. Invited talk at Sandia National Laboratories, Albuquerque, NM, USA, January 2003.

[261] Ron Oldfield and David Kotz. Armada: A Parallel File System for Computational Grids. In *Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 194–201, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[262] Ron Oldfield and David Kotz. Armada: a Parallel I/O Framework for Computational Grids. *Future Generation Computing Systems (FGCS)*, 18(4):501–523, March 2002.

[263] Ron Oldfield and David Kotz. The Armada Framework for Parallel I/O on Computational Grids. Work-in-progress report at the Conference on File and Storage Technologies, January 2002.

[264] Ron Oldfield and David Kotz. Improving Data Access for Computational Grid Applications. *Cluster Computing, The Journal of Networks, Software Tools and Applications*, 2003. Accepted for publication.

[265] Emil Ong, Ewing Lusk, and William Gropp. Scalable Unix Commands for Parallel Processors: A High-Performance Implementation . In *Proceedings of the Recent Advances in Parallel Virtual Machine and Message Passing Interface: 8th European PVM/MPI Users' Group Meeting*, volume 2131/2001 of *Lecture Notes in Computer Science*, page 410 ff, Santorini/Thera, Greece, September 2001. Springer.

[266] Parallel I/O Archive. `http://www.cs.dartmouth.edu/pario/`.

[267] Manish Parashar, Gregor von Laszewski, Snigdha Verma, Jarek Gawor, and Kate Keahey. A CORBA Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 14:1057–1074, July 2002.

[268] Norman Paton, Malcolm Atkinson, Vijay Dialani, Dave Pearson, Tony Storey, and Paul Watson. Database Access and Integration Services on the Grid. Technical Report UK e-Science Programme Technical Report Series Number UKeS-2002-03, National e-Science Centre, March 2002.

[269] Craig J. Patten and Ken A. Hawick. Flexible High-Performance Access to Distributed Storage Resources. In *Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, page 175, Pittsburgh, Pennsylvania, USA, 2000.

[270] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 16, pages 224–244. IEEE Computer Society Press and Wiley, New York, NY, USA, 2001.

[271] José M. Pérez, Félix García, Jesús Carretero, Alejandro Calderón, and Javier Fernández. A Parallel I/O Middleware to Integrate Heterogeneous Storage Resources on Grids. In *1st European Across Grids Conference*, Universidad de Santiago de Compostela, Spain, February 2003.

[272] María S. Pérez, Jesús Carretero, Félix Garcia, José M. Peña, and Victor Robles. MAPFS: A Flexible Infrastructure for Data-Intensive Grid Applications. In *1st European Across Grids Conference*, Universidad de Santiago de Compostela, Spain, February 2003.

[273] David Petrou, Khalil Amiri, Gregory R. Ganger, and Garth A. Gibson. Easing the Management of Data-Parallel Systems via Adaptation . In *Proceedings of the 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[274] PHOENIX: A Physics Experiment at RHIC. `http://www.phenix.bnl.gov/`.

[275] Beth Plale, Volker Elling, Greg Eisenhauer, Karsten Schwan, Davis King, and Vernard Martin. Realizing Distributed Computational Laboratories. *International Journal of Parallel and Distributed Systems and Networks*, 2(3), 1999.

[276] Beth Plale and Karsten Schwan. dQUOB: Managing Large Data Flows Using Dynamic Embedded Queries. In *IEEE High Performance Distributed Computing (HPDC)*, August 2000.

[277] Beth Plale and Karsten Schwan. Dynamic Querying of Streaming Data with the dQUOB System. In *IEEE Transactions on Parallel and Distributed Systems*, volume 14, April 2003.

[278] Kenneth W. Preslan, Andrew P. Barry, Jonathan Brassow, Russell Cattelan, Adam Manthei, Erling Nygaard, Seth Van Oort, David Teigland, Mike Tilstra, Matthew O'Keefe, Grant Erickson, and Manish Agarwal. Implementing Journaling in a Linux Shared Disk File System. In *IEEE Symposium on Mass Storage Systems*, pages 351–378, 2000.

[279] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe. A 64-bit, Shared Disk File System for Linux. In *Proceedings of the Seventh NASA Goddard Conference on Mass Storage Systems*, pages 22–41, San Diego, CA, USA, March 1999. IEEE Computer Society Press.

[280] Python . `http://www.python.org`.

[281] Kumaran Rajaram. Principal Design Criteria Influencing the Performance of a Portable, High Performance Parallel I/O Implementation. Master's thesis, Department of Computer Science, Mississippi State University, May 2002.

[282] Arcot Rajasekar, Richard Marciano, and Reagan Moore. Collection-Based Persistent Archives. In *IEEE Symposium on Mass Storage Systems 1999*, pages 176–184, 1999.

[283] Arcot Rajasekar, Michael Wan, and Reagan Moore. MySRB & SRB – Components of a Data Grid . In *The 11th International Symposium on High Performance Distributed Computing (HPDC-11)*, Edinburgh, Scotland, 24–26 July 2002.

[284] Arcot Rajasekar, Michael Wan, Reagan W. Moore, Arun Jagatheesan, and George Kremenek. Real Experiences with Data Grids - Case-Studies in Using the SRB. In *Proceedings of the 6th International Conference/Exhibition on High Performance Computing Conference in Asia Pacific Region (HPC-Asia)*, Bangalore, India, December 2002.

[285] Rajesh Raman and Miron Livny. High Throughput Resource Management. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, chapter 13. Morgan Kaufmann, San Francisco, CA, USA, 1999.

[286] Raju Rangaswami, Zoran Dimitrijevic, Edward Chang, and Klaus E. Schauser. MEMS-Based Disk Buffer for Streaming Media Servers. *To appear in IEEE International Conference on Data Engineering*, 2003.

[287] Russell K. Rew and Glenn P. Davis. Unidata's NetCDF Interface for Data Access. In *Proceedings of the Thirteenth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, Anaheim, California, USA, February 1997. American Meteorology Society.

[288] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, and Daniel A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 172ff., Chicago, Illinois, USA, 28–31 July 1998.

[289] John R. Rice and Ronald F. Boisvert. From Scientific Software Libraries to Problem-Solving Environments. *IEEE Computational Science and Engineering*, pages 44–53, 1996.

[290] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active Disks for Large-Scale Data Processing . *IEEE Computer*, 34(6):68–74, June 2001.

[291] Erik Riedel and Garth A. Gibson. *Active Disks - Remote Execution for Network-Attached Storage*. PhD thesis, Electrical & Computer Engineering, Carnegie Mellon University Pittsburgh, Pittsburgh, PA, USA, November 1999.

[292] RIO: Remote I/O for Metasystems . `http://www-fp.globus.org/details/rio.html`.

[293] Manuel Rodríguez-Martínez and Nick Roussopoulos. MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources. In *ACM SIGMOD Conference 2000*, pages 213–224, Dallas, TX, USA, May 2000.

[294] Rob Ross, Walt Ligon, Phil Carns, Neill Miller, and Rob Latham. Parallel Virtual File System, Version 2 . Technical report, Parallel Architecture Research Laboratory (PARL), Clemson University, Clemson, South Carolina, USA, September 2003.

[295] Robert Ross, Daniel Nurmi, and Albert Cheng Michael Zingale. A Case Study in Application I/O on Linux Clusters. In *Proceedings of SC2001*, Denver, CO, USA, November 2001.

[296] Robert B. Ross. *Reactive Scheduling for Parallel I/O Systems*. PhD thesis, Electrical and Computer Engineering Department, Clemson University, Clemson, South Carolina, USA, 2000.

[297] Robert B. Ross and Walter B. Ligon III. Server-Side Scheduling in Cluster Parallel I/O Systems . *Calculateurs Parallèles Journal Special Issue on Parallel I/O for Cluster Computing*, October 2001.

[298] Run II Computing Project . `http://runiicomputing.fnal.gov/`.

[299] Joel Saltz. Data Realted Systems Software. Technical report, University of Maryland, Johns Hopkins Medical School.

[300] Asad Samar and Heinz Stockinger. Grid Data Management Pilot (GDMP): A Tool for Wide Area Replication. In *IASTED International Conference on Applied Informatics (AI2001)*, Innsbruck, Austria, February 2001.

[301] Samba . `http://www.samba.org`.

[302] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Technical Conference*, pages 119–130, June 1985.

[303] Peter Sanders, Sebastian Egner, and Jan H. M. Korst. Fast Concurrent Access to Parallel Disks. In *Symposium on Discrete Algorithms*, pages 849–858, 2000.

[304] Erich Schikuta and Thomas Fuerle. ViPIOS Islands: Utilizing I/O Resources on Distributed Clusters. In *15th International Conference on Parallel and Distributed Computing Systems (PDCS 2002), Special Session on Data Management and I/O Techniques for Data Intensive Applications*, Louisville, KY USA, September 2002. IASTED.

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

[306] Erich Schikuta and Heinz Stockinger. Parallel I/O for Clusters: Methodologies and Systems. In Rajkumar Buyya, editor, *High Peformance Cluster Computing*, pages 439–462. Prentice Hall PTR, 1999.

[307] Steven W. Schlosser, John Linwood Griffin, David Nagle, and Gregory R. Ganger. Designing Computer Systems with MEMS-Based Storage. In *Architectural Support for Programming Languages and Operating Systems*, pages 1–12, 2000.

[308] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST02)*, 28–30 January 2002.

[309] Martin Schulz. DIOM: Parallel I/O for Data Intensive Applications on Commodity Clusters. In *Parallel and Distributed Computing and Systems (PDCS)*, Los Angeles, CA, USA, August 2001.

[310] Martin Schulz and Daniel A. Reed. Using Semantic Information to Guide Efficient I/O on Clusters. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing*, pages 135–142, Edinburgh, Scotland, 2002. IEEE Computer Society Press.

[311] SDSS: Sloan Digital Sky Survey . `http://www.sdss.org/`.

[312] Kent E. Seamons. *Panda: Fast Access to Persistent Arrays Using High Level Interfaces and Server Directed Input/Output*. PhD thesis, University of Illinois at Urbana-Champaign, May 1996.

[313] Kent E. Seamons, Ying Chen, Mark P. Jones, J. Jozwiak, and Marianne Winslett. Server-Directed Collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, USA, December 1995. IEEE Computer Society Press.

[314] Kent E. Seamons and Marianne Winslett. Multidimensional Array I/O in Panda 1.0. *Journal of Supercomputing*, 10(2):191–211, 1996.

[315] Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, October 2000.

[316] Edward Seidel and Wai-Mo Suen. Numerical Relativity as a Tool for Computational Astrophysics. *Journal of Computational and Applied Mathematics*, 1999.

[317] Self-Certifying File System. `http://www.fs.net`.

[318] SETI@home . `http://setiathome.ssl.berkeley.edu/`.

[319] SGI CXFS: A High-Performance, Multi-OS SAN Filesystem from SGI.

[320] SGI CXFS Shared Filesystem. `http://www.sgi.com/pdfs/2816.pdf`.

[321] SGI XFS: Extended File System. `http://www.sgi.com/software/xfs`.

[322] Xiaohui Shen and Alok Choudhary. A Distributed Multi-Storage Resource Architecture and I/O Performance Prediction for Scientific Computing. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, page 21ff., 1–4 August 2000.

[323] Xiaohui Shen and Alok N. Choudhary. DPFS: A Distributed Parallel File System. In *Proceedings of the International Conference on Parallel Processing (ICPP '01)*, pages 533–544, Valencia, Spain, 03–07 September 2001.

[324] Arie Shoshani, Alex Sim, and Junmin Gu. Storage Resource Managers: Middleware Components for Grid Storage . In *MSS*, 2002.

[325] Seth Shostak. *Sharing the Universe: Perspectives on Extraterrestrial Life*. Berkeley Hills Books, January 1998.

[326] Sistina . `http://www.sistina.com`.

[327] Global Telecommunications Company Calls on Sistina GFS. `http://www.sistina.com/downloads/casestudies/telco_CS202A.pdf`.

[328] SOAP: Simple Object Access Protocol. `http://www.w3.org/TR/SOAP/`.

[329] Steve R. Soltis, Grant Erickson, Ken Preslan, Matthew T. O'Keefe, and Tom Ruwart. The Design and Performance of a Shared Disk File System for IRIX. In *The Sixth Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Fifteenth IEEE Symposium on Mass Storage Systems*, pages 41–56, College Park, MD, USA, March 1998.

[330] Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O'Keefe. The Global File System. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems,* pages 319–342, College Park, MD, September 1996. IEEE Computer Society Press.

[331] Hyo J. Song, Xin Liu, Dennis Jakobsen, Ranjita Bhagwan, Xianan Zhang, Kenjiro Taura, and Andrew A. Chien. The MicroGrid: a Scientific Tool for Modeling Computational Grids. In *in Proceedings of SC2000*, 2000.

[332] STAR. `http://www.star.bnl.gov/`.

[333] Heinz Stockinger. Dictionary on Parallel Input/Output. Master's thesis, Department of Data Engineering, University of Vienna, Austria, February 1998.

[334] Heinz Stockinger, Asad Samar, Bill Allcock, Ian Foster, Koen Holtman, and Brian Tierney. File and Object Replication in Data Grids. In *10th IEEE Symposium on High Performance Distributed Computing (HPDC-10)*. IEEE Press, 7–9 August 2001.

[335] Sun HPC Parallel File System Documentation . `http://docs.sun.com/db/doc/805-1555/6j1h5o7ng?a=view`.

[336] Mark Swanson, Leigh Stoller, and John Carter. Making Distributed Shared Memory Simple, yet Efficient. In *HIPS 98*, 1998.

[337] Adam Sweeney, Doug Doucette, Wwi Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, 22–26 1996.

[338] Cluster File Systems. Lustre: a Scalable, High-Performance File System. Technical Report Whitepaper Version 1.0, Cluster File Systems, Inc., November 2002.

[339] Osamu Tatebe, Youhei Morita, Satoshi Matsuoka, Noriyuki Soda, Hiroyuki Sato, Yoshio Tanaka, Satoshi Sekiguchi, Yoshiyuki Watase, Masatoshi Imori, and Tomio Kobayashi. Grid Data Farm for Petascale Data Intensive Computing. Technical Report ETL-TR2001-4, Electrotechnical Laboratory, 2001.

[340] Osamu Tatebe, Youhei Morita, Satoshi Matsuoka, Noriyuki Soda, and Satoshi Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, pages 102–110, 2002.

[341] Tcl: Tool Command Language . `http://tcl.sourceforge.net`.

[342] TeraGrid . `http://www.teragrid.org/`.

[343] TerraFlow: Computations on Massive Grids. `http://www.cs.duke.edu/geo*/terraflow`.

[344] Terravision: Interactive Terrain Visualization System . `http://www.ai.sri.com/TerraVision/`.

[345] Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny. The Kangaroo Approach to Data Movement on the Grid. In *Proceedings of the Tenth (IEEE) Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, CA, August 2001.

[346] Douglas Thain, John Bent, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. Gathering at the Well: Creating Communities for Grid I/O. In *Proceedings of Supercomputing 2001*, Denver, Colorado, USA, November 2001.

[347] Douglas Thain and Miron Livny. Parrot: Transparent User-Level Middleware for Data-Intensive Computing , September 2003.

[348] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the Grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[349] Rajeev Thakur, Rajesh Bordawekar, Alok Choudhary, Ravi Ponnusamy, and Tarvinder Singh. PASSION Runtime Library for Parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[350] Rajeev Thakur and Alok Choudhary. An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays. *Scientific Programming*, 5(4):301–317, 1996.

[351] Rajeev Thakur, Alok Choudhary, Rajesh Bordawekar, Sachin More, and Sivaramakrishna Kuditipudi. Passion: Optimized I/O for Parallel Applications . *IEEE Computer*, 29(6):70–78, June 1996.

[352] Rajeev Thakur, William Gropp, and Ewing Lusk. An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In *Proceedings of the Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 180–187, October 1996.

[353] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[354] Rajeev Thakur, William Gropp, and Ewing Lusk. On Implementing MPI-IO Portably and with High Performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[355] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing Noncontiguous Accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, January 2002.

[356] Rajeev Thakur, Ewing Lusk, and William Gropp. I/O Characterization of a Portable Astrophysics Application on the IBM SP and Intel Paragon. Technical Report MCS-P534-0895, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[357] Rajeev Thakur, Ewing Lusk, and William Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, October 1997.

[358] Oliver E. Theel and Brett D. Fleisch. The Boundary-Restricted Coherence Protocol for Scalable and Highly Available DSM Systems. *The Computer Journal*, 39(6):496–510, 1996.

[359] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[360] Mary Thomas, Steve Mock, and Gregor von Laszewski. A Perl Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, accepted.

[361] Brian L. Tierney. High-Performance Data Intensive Distributed Computing. Technical report, Lawrence Berkeley National Laboratory.

[362] Brian L. Tierney, William E. Johnston, Hanan Herzog, Gary Hoo, Guojun Jin, and Jason Lee. System Issues in Implementing High Speed Distributed Parallel Storage Systems. In *Proceedings of the USENIX Symposium on High Speed Networking*, pages 61–72, August 1994.

[363] Brian L. Tierney, William E. Johnston, Hanan Herzog, Gary Hoo, Guojun Jin, Jason Lee, Ling Tony Chen, and Doron Rotem. Using High Speed Networks to Enable Distributed Parallel Image Server Systems. In *Supercomputing '94 Conference*, pages 610–619, 1994.

[364] Brian L. Tierney, William E. Johnston, Jason Lee, Gary Hoo, and Mary Thompson. An Overview of the Distributed Parallel Storage Server (DPSS) . Technical report, Lawrence Berkeley National Laboratory.

[365] Brian L. Tierney, Jason Lee, Ling Tony Chen, Hanan Herzog, Gary Hoo, Guojun Jin, and William E. Johnston. Distributed Parallel Data Storage Systems: A Scalable Approach to High Speed Image Servers. In *Proceedings of the ACM Multimedia*, pages 399–405, October 1994.

[366] Titan Project. `http://www.cs.umd.edu/projects/hpsl/chaos/ResearchAreas/titan.html`.

[367] Laura Toma, Rajiv Wickremesinghe, Lars Arge, Jeffrey S. Chase, Jeffrey Scott Vitter, Patrick N. Halpin, and Dean Urban. Flow Computation on Massive Grids. In *Proc. ACM Symposium on Advances in Geographic Information Systems*, 2001. Journal version in preparation.

[368] Triana . `http://www.triana.co.uk/`.

[369] Steve Tuecke, Karl Czajkowski, Ian Foster, Jeff Frey, Steve Graham, Carl Kesselman, Tom Maquire, Thomas Sandholm, David Snelling, and Pete Vanderbilt. Open Grid Services Infrastructure (OGSI) Version 1.0 . Technical report, Global Grid Forum, http://www.ggf.org, June 2003.

[370] Unidata. `http://www.unidata.ucar.edu`.

[371] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Evaluation of Active Disks for Large Decision Support Databases. Technical Report TRCS99-25, Computer Science Dept., University of California, Santa Barbara, CA, USA, 25, 1999.

[372] Vanilla Universe . `http://www.cs.wisc.edu/condor/manual/v6.4/2_4Road_map_Running.html#SECTION00%3412000000000000000`.

[373] Sudharshan Vazhkudai, Steve Tuecke, and Ian Foster. Replica Selection in the Globus Data Grid. In *Proceedings of the First IEEE/ACM International Conference on Cluster Computing and the Grid (CCGRID 2001)*, pages 106–113. IEEE Computer Society Press, May 2001.

[374] VDT: Virtual Data Toolkit. Components . `http://www.lsc-group.phys.uwm.edu/vdt/contents1.1.11.html`.

[375] VDT: Virtual Data Toolkit . `http://www.lsc-group.phys.uwm.edu/vdt/home.html`.

[376] Darren Erik Vengroff and Jeffrey Scott Vitter. Supporting I/O-Efficient Scientific Computation in TPIE. In *In Proceedings of the IEEE Symposium on Parallel and Distributed Computing*, 1995.

[377] Snighda Verma, Manish Parashar, Jarek Gawor, and Gregor von Laszewski. Design and Implementation of a CORBA Commodity Grid Kit. In Craig A. Lee, editor, *Second International Workshop on Grid Computing - GRID 2001*, volume 2241 of *Lecture Notes in Computer Science*, pages 2–12, Denver, CO, USA, November 2001. Springer.

[378] Murali Vilayannur, Anand Sivasubramaniam, Mahmut Kandemir, Rajeev Thakur, and Robert Ross. Discretionary Caching for I/O on Clusters. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid (CCGRID)*, page 96ff., 12–15 May 2003.

[379] Visible Embryo NGI Project . `http://www.visembryo.org`.

[380] Jeffrey Scott Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[381] Jeffrey Scott Vitter and Elizabeth A. M. Shriver. Algorithms for Parallel Memory I: Two-Level Memories. *Algorithmica*, 12(2-3):110–147, 1994.

[382] Visualization of Astronomy Data with ADR and MPIRE . `http://mpire.sdsc.edu/adr99.html`.

[383] Bill von Hagen. Using the Intermezzo Distributed Filesystem Getting Connected in a Disconnected World. *http://www.linuxplanet.com*, August 2002.

[384] Gregor von Laszewski, Beulah Alunkal, Jarek Gawor, Ravi Madduri, Pawel Plaszczak, and Xian-He Sun. A File Transfer Component for Grids . In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2003.

[385] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.

[386] Gregor von Laszewski, Ian Foster, Jarek Gawor, Peter Lane, Nell Rehn, and Mike Russell. Designing Grid-Based Problem Solving Environments and Portals. In *34th Hawaiian International Conference on System Science*, Maui, Hawaii, USA, January 2001.

[387] Gregor von Laszewski, Ian Foster, Jarek Gawor, Warren Smith, and Steve Tuecke. CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids. In *ACM Java Grande 2000 Conference*, pages 97–106, San Francisco, CA, USA, June 2000.

[388] Gregor von Laszewski, Jarek Gawor, Sriram Krishnan, and Keith Jackson. Commodity Grid Kits - Middleware for Building Grid Computing Environments. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, chapter 26. John Wiley & Sons Inc., December 2002.

[389] Gregor von Laszewski, Jarek Gawor, Peter Lane, Nell Rehn, Mike Russell, and Keith Jackson. Features of the Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 14(13–15):1045–1055, 2002.

[390] Gregor von Laszewski, Gail Pieper, and Patrick Wagstrom. Gestalt of the Grid. In Salim Hariri and Manish Parashar, editors, *Performance Evaluation and Characterization of Parallel and Distributed Computing Tools*, Wiley Book Series on Parallel and Distributed Computing. John Wiley & Sons Inc., 2002.

[391] Gregor von Laszewski, Branko Ruscic, Patrick Wagstrom, Sriram Krishnan, Kaizar Amin, Sandeep Nijsure, Sandra Bittner, Reinhardt Pinzon, John C. Hewson, Melita L. Norton, Mike Minkoff, and Al Wagner. A Grid Service-Based Active Thermochemical Table Framework. In *Proceedings of Grid Computing - GRID 2002 : Third International Workshop*, volume 2536 of *Lecture Notes in Computer Science*, pages 25–38, Baltimore, MD, USA, November 2002. Springer.

[392] W3C. Web Services Activity. `http://www.w3c.org/2002/ws`.

[393] WALDO: Wide Area Large Data Object . `http://www-itg.lbl.gov/WALDO/`.

[394] David Watson, Yan Luo, and Brett D. Fleisch. The Oasis+ Dependable Distributed Storage System. In *Proceedings of the 2000 Pacific Rim International Symposium on Dependable Computing*, Los Angeles, CA, USA, 18–19 December 2000.

[395] David Watson, Yan Luo, and Brett D. Fleisch. Experiences with Oasis+: A Fault Tolerant Storage System. In *Proceedings of the IEEE International Conference on Cluster Computing*, Newport Beach, CA, USA, 8–11 October 2001.

[396] Richard W. Watson and Robert A. Coyne. The Parallel I/O Architecture of the High-Performance Storage System (HPSS). In *Proceedings of the Fourteenth IEEE Symposium on Mass Storage Systems*, pages 27–44. IEEE Computer Society Press, September 1995.

[397] WebDAV: Web-based Distributed Authoring and Versioning . `http://www.webdav.org`.

[398] Von Welch, Frank Siebenlist, Ian Foster, John Bresnahan, Karl Czajkowski, Jarek Gawor, Carl Kesselman, Sam Meder, Laura Pearlman, and Steve Tuecke. Security for Grid Services. In *Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*. IEEE Press, 2003.

[399] Brian S. White, Andrew S. Grimshaw, and Anh Nguyen-Tuong. Grid-Based File Access: The Legion I/O Model , August 2000.

[400] Brian S. White, Michael Walker, Marty Humphrey, and Andrew S. Grimshaw. LegionFS: A Secure and Scalable File System Supporting Cross-Domain High-Performance Applications. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 59–59, Denver, Colorado, 2001.

[401] Rajiv Wickremesinghe, Jeffrey S. Chase, and Jeffrey S. Vitter. Distributed Computing with Load-Managed Active Storage. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing*, pages 24–34, Edinburgh, Scotland, 2002. IEEE Computer Society Press.

[402] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors,

High Performance Mass Storage and Parallel I/O: Technologies and Applications, pages 90–106. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[403] WP2. User Guide for EDG Replica Manager 1.5.4. Technical report, CERN, European Organization for Nuclear Research, October 2003.

[404] WP2. User Guide for EDG Replica Optimization Service 2.1.4. Technical report, CERN, European Organization for Nuclear Research, October 2003.

[405] WP2. User Guide for EDG RLS Replica Location Index 2.1.4. Technical report, CERN, European Organization for Nuclear Research, October 2003.

[406] WP2. User Guide for Local Replica Catalog 2.1.3. Technical report, CERN, European Organization for Nuclear Research, October 2003.

[407] WP2. User Guide for Replica Metadata Service 2.1.3. Technical report, CERN, European Organization for Nuclear Research, October 2003.

[408] XPath: XML Path Language . `http://www.w3.org/TR/xpath`.

[409] XQuery: XML Query Language . `http://www.w3.org/TR/xquery/`.

[410] Shengke Yu, Marianne Winslett, Jonghyun Lee, and Xiaosong Ma. Automatic and Portable Performance Modeling for Parallel I/O: A Machine-Learning Approach. *ACM SIGMETRICS Performance Evaluation Review*, 30(3):3–5, December 2002.

[411] Erez Zadok, Ion Badulescu, and Alex Shender. Extending File System Using Stackable Templates. In *Proceedings of the 1999 Annual USENIX Technical Conference*, pages 57–70. USENIX Association, 1999.