Eötvös Loránd University

Thesis for Master of Science

The optimization of data access on clusters and data grids

Strategy, aspect, extension of JDL

László Csaba Lőrincz

Supervisor:
Zoltán Horváth

Budapest
2003

Table of contents

Abstract

Applications on the grid need different types of resources. Such resources are CPU, memory, network bandwidth and secondary storage. The grid middleware tries to allocate resources to applications in a way that results can be computed as fast and as efficiently as possible.

The main focus in this thesis is on the secondary storage, on data access. Many applications read and write large data files during their operations. Often these files are not available on the computing element where the application runs. The optimal strategy for accessing these files depends on the kind of the application: computation-intensive applications may spend long periods of time between successive data accesses, while data-intensive applications may need to access large files very frequently. Consequently a small puffer may be enough for storing a local copy of the data needed in the next step of computation or a complete copy of a huge file may be necessary in advance. The access method as well as the granularity of the read/write operations may vary with each input and output file used. Therefore the optimal data access may require the copying of some files as a whole, as well as accessing other files in parts.

This thesis is organized as follows:

- First it is presented the nature of data grids in general showing the arising technical challenges.
- Than a short description of the architecture and main services of the European DataGrid (EDG) is given.
- Than comes the designing of a strategy for optimizing the data access.

The techniques and results presented in this thesis were designed to be used in the EDG project too[1].

[1] This paper was created as part of the ***IKTA 89/2002*** project.

Introduction

High performance computing has become a key technology for many scientific and engineering activities. Supercomputers are increasingly used by scientists to study complex phenomena through the use of computer simulation. This use of computers adds an important new method for scientific and engineering research, one that is complementary to theory and experiment. Computer models can be used to simulate phenomena that cannot be measured with experiments or can be studied only through computer simulation, or they can simply provide a less costly means for studying phenomena.

Present supercomputers are powerful enough to predict complex nonlinear phenomena, developing engineering prototypes, exploring the physical parameter space prior to doing experiments, and even simulating events occurring in the real world. Yet, despite continued increases in supercomputer capabilities, there remain many applications whose computational requirements exceed the resources available at even the largest supercomputer centers. For these applications, computational grids offer the promise of access to increased computational capacity through the simultaneous, coordinated use of geographically separated large-scale computers linked by networks. Through this technique, the “size” of an application can be extended beyond the resources available at a particular location — indeed, potentially beyond the capacity of any one computing site anywhere [FK01].

The distributed supercomputing applications represent a class of applications whose computational requirements are so demanding that they can be met only by combining multiple high-capacity resources on a computational grid into a single, virtual distributed supercomputer.

Typical distributed supercomputer applications are the large-scale science and engineering applications, as the following:

- traditional batch physical simulations, multidisciplinary simulations, and coupled models
- distributed interactive simulations, which include dynamic terrain, weather and high-fidelity simulation of specific objects
- applications that occasionally require much more computing power for some phase of the computation (medical and numerical solvers) or realtime applications that sometime have computationally intensive components (rendering, signal processing)
- complex analyses on data from large, distributed data archives (astronomy data analysis — Digital Sky project —, radar data analysis).

Distributed supercomputing applications are distinguished by the scale of their resource requirements like peak computing speed, memory size, and communication volume, unlike high-throughput applications which are driven by aggregate performance requirements. Moreover distributed supercomputing applications frequently require access to large data archives or digital libraries (for input and output).

While the primary motivation for distributed supercomputing is access to increased capacity, additional benefits can be achieved by the heterogeneity of the underlying resources. Many applications have several phases using different algorithms. Every algorithm may run more efficiently on an architecture which satisfies its special needs (some algorithms may require more memory, some may prefer parallel systems, while others may run better on systems with special processors designed for vector related computations). By executing each phase of an application on the node with the most suitable architecture and configuration, the overall application may run in much less time than on a homogeneous system of the same aggregate power.

The real and specific problem that triggers the ***Grid concept*** is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. The sharing is not primarily file exchange but rather direct access to computers, software, data, and other resources, as it is required by a range of collaborative problem-solving and resource-brokering strategies. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs.

Technical challenges

Taking advantage of the potential benefits of distributed supercomputing requires solving a variety of **technical challenges**.

Network bandwidth and latency

The grid can be viewed as a large distributed-memory parallel computer consisting of multiple processors exchanging data across communication links. In parallel computers, the interconnection network is provided by the vendor. In clusters of workstations or PCs the links may be commodity networks such as Ethernet or higher-performance, lower-latency networks such as Myrinet. In distributed supercomputing the communication links are provided by wide area networks. Thus a distributed supercomputer can be viewed as a metacomputer, a parallel computer with few very large heterogeneous nodes, connected by a relatively slow, high-latency communication networks.

Starting from this perspective there are a set of basic templates for constructing distributed supercomputing applications (called decomposition techniques):

- *pipeline or dataflow decomposition*, used in applications, where a sequence of complex operations is applied to a series of data elements
- *functional decomposition*, used in multidisciplinary simulation and coupled models, where the functions of the application can be distributed across grid resources

- *data-parallel decomposition*, used in applications, where the same algorithm is applied to every data element, and there is loose coupling between the elements.

In every case the central issue is how to manage the potentially large communication latency and limited bandwidth between the components of the application. These problems are complicated by the fact that the performance of the underlying network can vary significantly during the execution of an application, and the use of tightly coupled computations, which require efficient synchronizations and well balanced loads.

Supercomputing applications also can require voluminous input, output and interprocessor communication. The data transfers can have highly variable periods and durations in both directions. Consequently, the effects of limited or highly variable bandwidth can also have dramatic impact on application performance.

The development and use of more flexible algorithms (new latency-tolerant algorithms such as loosely synchronous algorithms, or algorithms which can adjust to time-varying resource availability), using specialized networks protocols or overlapping communication with computation can solve the bandwidth and latency issues (future very high performance computers may require similar innovations in latency tolerant algorithms, as computer speeds increase faster than memory speeds and as memory hierarchies get deeper).

Scheduling

Scheduling and application configuration are also significant challenges to distributed supercomputing applications. Obtaining peak application performance can depend on carefully selecting the type and number of processors used, base on application characteristics, the available network bandwidth and latency, and the location and volume of input and output data, and tuning the behavior of the application to the resources available.

Recalculating a data value depending on, for example, the amount of local memory, the location and volume of the remote data, can yield to faster execution than writing the result to disk and than reading it back into the memory, when it is needed. Balancing the number of nodes used against the network bandwidth and the ability to overlap computation and communication can also increase the performance of the application.

In the future, scheduling and tuning the applications should be done automatically, unlike today, when it is done by hand.

Fault recovery

Compute intensive applications often require many hours, days or even weeks of execution to solve a single problem. On traditional supercomputers runs of a few hours are submitted and intermediate results saved to be used as the starting point for the next run. Saving the state of the computation on secondary storage (called checkpointing) is also used to avoid having to rerun the entire problem in the event of system failures. Checkpointing a parallel application is complicated, while difficulties arise in determining a consistent application state. Generating a checkpoint in distributed supercomputing applications is complicated by the size of the application, the looser coupling of computational resources, and the fact that most wide area networks do not guarantee reliable or ordered delivery.

An alternate to checkpointing is to exploit the distributed nature of the grid. While in a single supercomputer application, the entire application fails if any node of the machine fails, in distributed supercomputing applications fault-tolerant computational algorithms used with group communication protocols could reduce or completely eliminate the need of checkpointing.

Grid development tools

Computational grids present extremely complex execution environments. Just as a programmer would not think of hand-optimizing assembly code for a modern multi-issue microprocessor, it can not be expected from the developers of distributed supercomputing applications to hand-optimize their code for the grid environment. Consequently, bringing distributed supercomputing into widespread use will require advances in grid-specific compilation systems, component composition systems and application level tools.

Interaction between application, middleware and network

Application-level scheduling techniques, requires greater interaction between an application, grid middleware (i.e., Globus, Legion), and underlying network. Fundamental to this interaction is determining what type of information and control needs to be passed between layers, how information about lower layers can be used to modify application behavior, and how information about the application can be used to control the underlying infrastructure. Examples of such information flow are: support for co-allocation, application-specific networking protocols, and network quality of service.

The European DataGrid Project

DataGrid is a project funded by European Union. The objective is to build the next generation computing infrastructure providing intensive computation and analysis of shared large-scale databases, from hundreds of TeraBytes to PetaBytes, across widely distributed scientific communities [EDG]. Such capabilities are required in many scientific disciplines, including particle physics, biology, and earth sciences.

Building on emerging computational Grid technologies the main purpose is to establish a research network that is developing the technology components essential for the implementation of a world-wide data and computational Grid on a scale not previously attempted. An essential part of this project is the phased development and deployment of a large-scale Grid testbed.

The primary goals of the first phase of the EDG testbed were:

- to demonstrate that the EDG software components could be integrated into a production-quality computational Grid;
- to allow the middleware developers to evaluate the design and performance of their software;
- to expose the technology to end-users to give them hands-on experience;
- to facilitate interaction and feedback between endusers and developers.

This first testbed deployment was achieved towards the end of 2001. [EDGGJRB]

The work of the project is divided into functional areas:

- workload management,
- data management,
- grid monitoring and information systems,
- fabric management,
- mass data storage,
- testbed operation,
- network monitoring.

Definitions

Virtual organizations (VO) are dynamic collections of individuals, institutions, and resources defined by the sharing rules of the Grid concept. They enable distinct groups of organizations and/or individuals to share resources in a controlled fashion, so that members may collaborate to achieve a shared goal.

Generally, in advanced networks, ***middleware*** consists of services and other resources located between both the applications and the underlying packet forwarding and routing infrastructure [RCCFLMMST].

Logical file names (LFN) are the generic name of a file, while ***physical file names (PFN)*** gives the physical location and name of a particular file replica.

Condor-G is a Condor-Globus joint project, which combines the inter-domain resource management protocols of the Globus Toolkit with the intra-domain resource and job management methods of Condor to allow high throughput computing in multi-domain environments [CP].

The ***Input Sandbox*** is a set of files transferred to a Worker Node by means of GridFTP by the Resource Broker, so that any file required for the job to be executed (including the executable itself if necessary) can be sent to the local disk of the machine where the job will run. Similarly the ***Output Sandbox*** is a set of files to be retrieved from the Worker Node after the job finishes (other files are deleted). The files in the Output Sandbox are stored on the RB node until the user requests them to be transferred back to a UI machine.

The architecture

The EDG architecture is based on the Grid architecture proposed by Ian Foster and Carl Kesselman [FK01], with a reduced number of implemented services.

The components of the architecture are organized in layers. Components within each layer share common characteristics but can build on capabilities and behaviors provided by any lower layer.

The Grid architecture is based on the “hourglass model” [RTIF]. The neck of the hourglass defines a fundamental set of core abstractions and protocols, onto which many different high-level services can be mapped (the top of the hourglass), and which themselves can be mapped onto many different underlying technologies (the base of the hourglass) [GGTKCG].

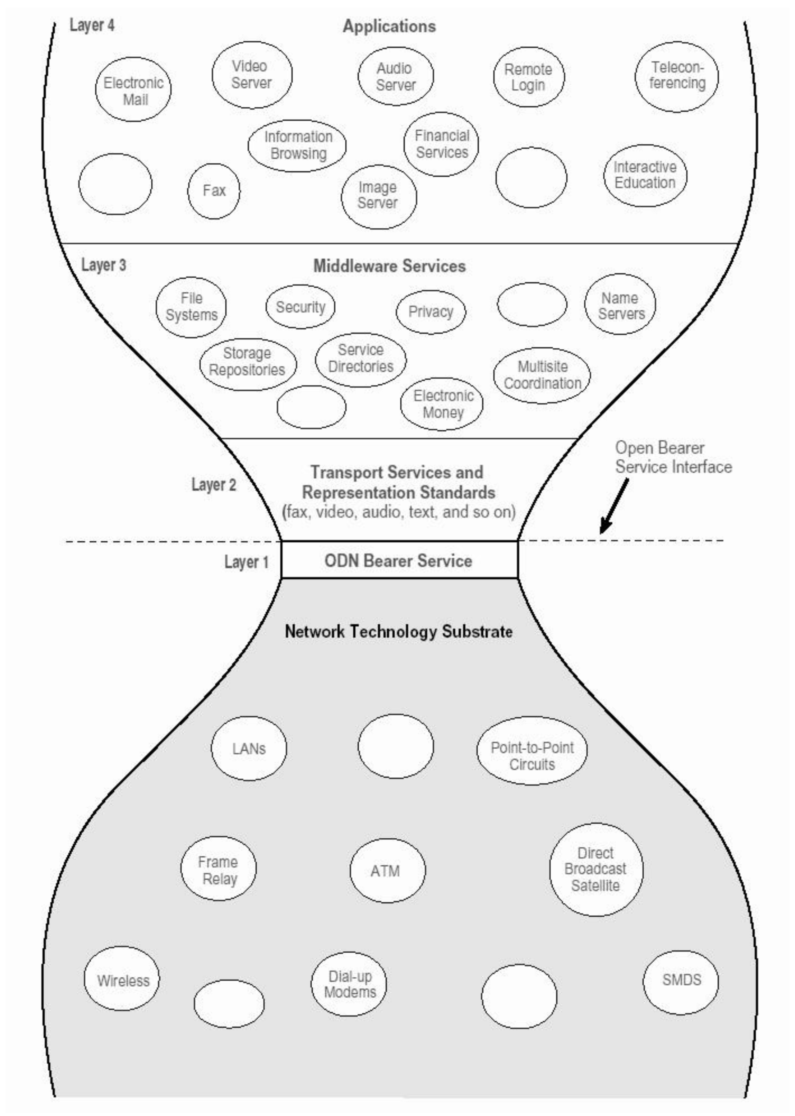

Figure 1. A four-layer model

The neck must consist of a small number of protocols. These are the ***Resource*** and ***Connectivity*** protocols, which facilitate the sharing of individual resources. These protocols are designed so that they can be implemented on top of a diverse range of resource types, defined at the ***Fabric layer***, and can in turn be used to construct a wide range of global services and application-specific behaviors at the ***Collective layer***.

The layered Grid architecture is based on the Internet Protocol Architecture, and has the following structure:

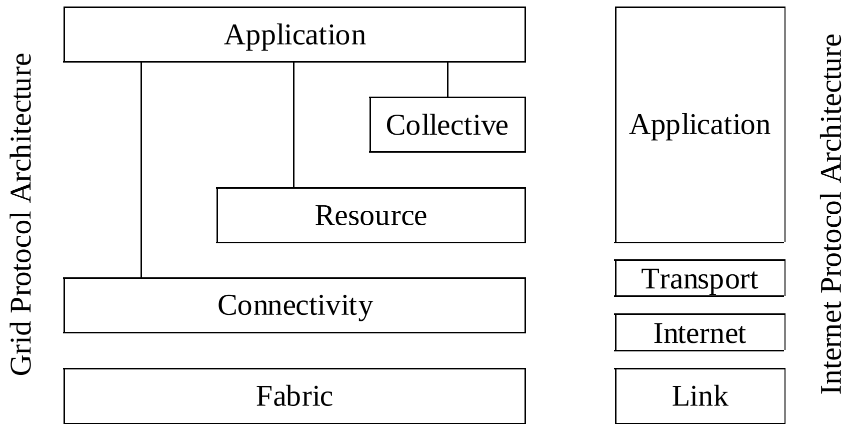

Figure 2. The layered Grid architecture and its relationship to the Internet protocol architecture.

The ***Fabric layer*** provides the resources which are shared among the Grid protocols (i.e. computational resources, storage systems, catalogs, network resources, and sensors. A “resource” may be a logical entity, such as a distributed file system, computer cluster, or distributed computer pool; in such cases, a resource implementation may involve internal protocols (e.g., the NFS storage access protocol or a cluster resource management system’s process management protocol), but these are not the concern of Grid architecture.

The ***Globus Toolkit*** has been designed to use (primarily) existing fabric components, including vendor-supplied protocols and interfaces. If the necessary Fabric-level behavior is not provided by a vendor, however, the Globus Toolkit includes the missing functionality. For example, enquiry software is provided for discovering structure and state information for various common resource types, such as computers (e.g., OS version, hardware configuration, load, scheduler queue status), storage systems (e.g., available space), and networks (e.g., current and predicted future load), and for packaging this information in a form that facilitates the implementation of higher-level protocols, specifically at the Resource layer. Resource management, on the other hand, is generally assumed to be the domain of local resource managers.

The ***Connectivity layer*** defines core communication and authentication protocols required for Grid-specific network transactions. Communication protocols enable the exchange of data between Fabric layer resources while authentication protocols cryptographically secure mechanisms for verifying the identity of users and resources. Communication requirements include transport, routing, and naming.

In the ***Globus Toolkit*** the Internet protocols are used for communication; the public-key based Grid Security Infrastructure (GSI) protocols are used for authentication, communication protection, and authorization. GSI builds on and extends the Transport Layer Security (TLS) protocols to address single sign-on, delegation, integration with various local security solutions (including Kerberos), and user-based trust relationships. X.509-format certificates are used. Stakeholder control of authorization is supported via an authorization toolkit that allows resource owners to integrate local policies via a Generic Authorization and Access (GAA) control interface.

The ***Resource layer*** building on the communication and authentication protocols of the Connectivity layer defines protocols (and APIs and SDKs) for the secure initiation, monitoring, and control of sharing operations on individual resources. Resource layer implementations of these protocols call Fabric layer functions to access and control local resources.

The ***Globus Toolkit*** defines client-side C and Java APIs and SDKs for a small and mostly standards-based set of protocols such as: Lightweight Directory Access Protocol (LDAP), HTTP-based Grid Resource Access and Management (GRAM) protocol, and an extended version of the File Transfer Protocol, the GridFTP. Server-side SDKs and servers are also provided for each protocol, to facilitate the integration of various resources (computational, storage, network) into the Grid.

The ***Collective layer*** contains global protocols and services (and APIs and SDKs) that ensure interactions between collections of resources. Because its components build on the narrow Resource and Connectivity layer “neck” in the protocol hourglass, they can implement a wide variety of sharing behaviors without placing new requirements on the resources being shared. For example: Directory services, Co-allocation, scheduling, and brokering services, Monitoring and diagnostics services, Data replication services, Grid-enabled programming systems, Software discovery services, Community authorization servers, Collaboratory services.

Many of the listed services build on Globus Connectivity and Resource protocols. Besides these the Meta Directory Service introduces Grid Information Index Servers (GIISs) to support arbitrary views on resource subsets, with the LDAP information protocol used to access resource-specific GRISs to obtain resource state and Grid Resource Registration Protocol (GRRP) used for resource registration.

The ***Application layer*** represents the final layer of the Grid architecture and consists of the user applications running within a Virtual Organization environment.

A sketch of the essential ***EDG architecture*** showing the relationship with the Operating System and the applications is shown in the following figure:

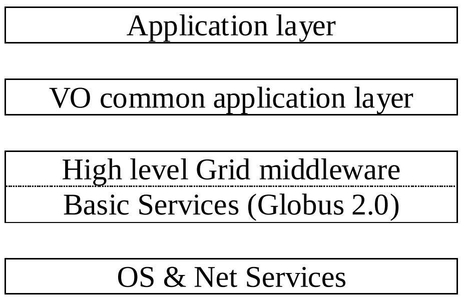
Application layer
VO common application layer
High level Grid middleware
Basic Services (Globus 2.0)
OS & Net Services

Figure 3. The schematic layered EDG architecture

The EDG architecture is a multi-layered architecture. At the lowest level is the operating system. Globus provides the basic services for secure and authenticated use of both operating system and network connections to safely transfer files and data and allow interoperation of distributed services. These services will be user by the user applications running on the Grid.

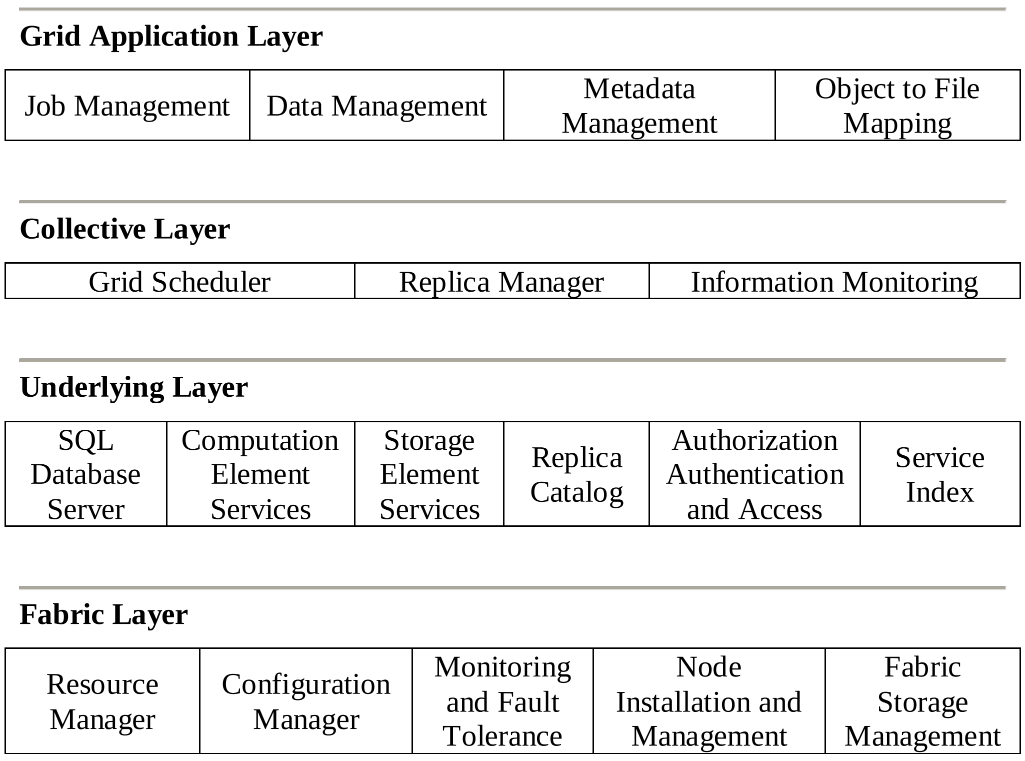

Figure 4. The multilayered EDG Grid architecture

There are sixteen services implemented in the EDG middleware. Most of them are based on the Globus 2 Toolkit (i.e. authentication (GSI), secure file transfer (GridFTP), information systems (MDS), job submission (GRAM) and the Globus Replica Catalogue), but the job submission system uses software from the Condor-G project [CP], and general open source software such as OpenLDAP is used too.

The service architecture of the EDG Grid is presented in the following figure:

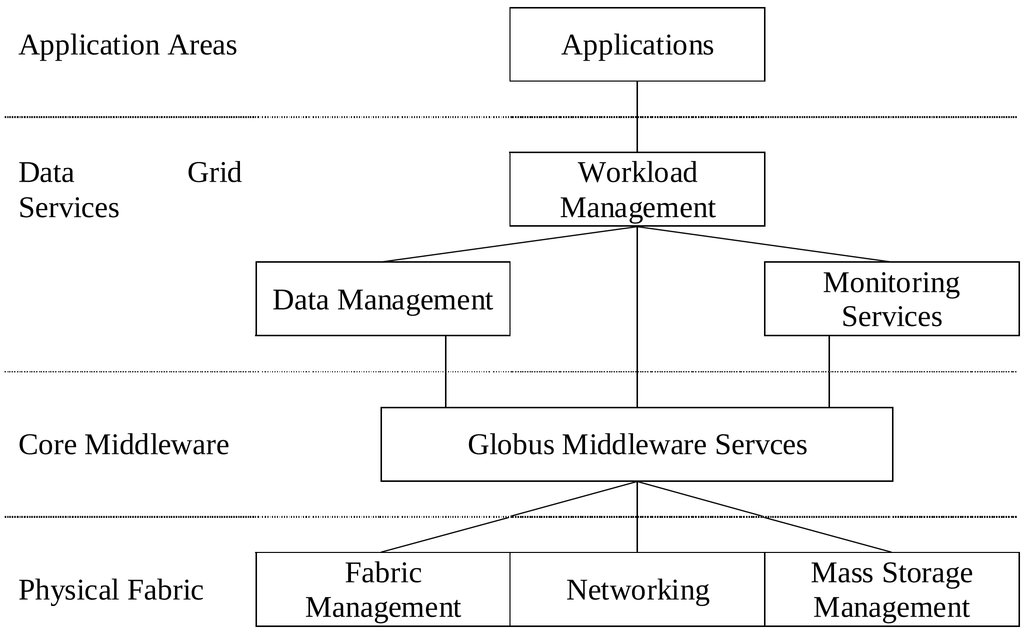

Figure 5. The EDG service architecture

Workload Management System (WMS)

The Workload Management System provides the architecture for distributed scheduling and resource management. It is composed from the following components:

- ***User Interface (UI)***, which represents the access point for the Grid user, allowing him to submit a job to the Resource Broker, and to retrieve the information about it and its output
- ***Resource Broker (RB)***, which performs match-making between the requirements of a job and the available resources, and attempts to schedule the jobs in an optimal way, taking into account the data location and the requirements specified by the user. The information about available resources is read dynamically from the Information and Monitoring System, while for resolving logical file names in physical file names the Replica Catalogue is used.

- ***Job Submission System (JSS)***, which is a wrapper for Condor-G [CP], an interface between the Grid and a Local Resource Management System (LRMS), usually a batch system like PBS, LSF or BQS.
- ***Information Index (II)***, which is a Globus MDS index that collects information from the Globus GRIS information servers running on the various Grid resources, published using LDAP, and read by the RB to perform the match-making.
- ***Logging and Bookkeeping (LB)***, which stores a variety of information about the status and history of submitted jobs using a MySQL database.

Job Description Language (JDL)

The JDL allows the various components of the Grid Scheduler to communicate requirements concerning the job execution. Examples of such requirements are:

- specification of the executable program or script to be run and arguments to be passed to it, and files to be used for the standard input, output and error streams
- specification of files that should be shipped with the job via Input and Output Sandboxes
- a list of input files and the access protocols the job is prepared to use to read them
- specification of the Replica Catalogue to be searched for physical instances of the requested input files
- requirements on the computing environment (OS, memory, free disk space, software environment etc) in which the job will run
- expected resource consumption (CPU time, output file sizes etc)
- a ranking expression used to decide between resources which match the other requirements

The ***Classified advertisements (ClassAds)*** language defined by the Condor project has been adopted for the Job Description Language because it has all the required properties.

Data Management System (DMS)

The goal of the Data Management System is to specify, develop, integrate and test tools and middleware to coherently manage and share petabyte-scale information volumes in high-throughput production-quality grid environments. The emphasis is on automation, ease of use, scalability, uniformity, transparency and heterogeneity. Its components are:

- ***Replica Manager***, which is still under development, but it will manage the creation of file replicas between different Storage Elements, simultaneously updating the Replica Catalogue, and optimizing the creation of file replicas by using network performance information and cost functions, according to the file location and size. different instances of the Replica Manager will be running on different sites, and will be synchronized to local Replica Catalogues, which will be interconnected by the Replica Location Index.

- ***Replica Catalogue***, which is used to resolve Logical File Names (LFN) into a set of corresponding Physical File Names (PFN) which locate each replica of a file providing a Grid-wide file catalogue for the members of a given Virtual Organization.
- ***GRID Data Mirroring Package (GDMP)***, which is a generic file replication tool used to automatically mirror file replicas securely and efficiently from one Storage Element to a set of other subscribed sites using several Globus Grid tools. It also supports pre- and post-processing plugins. It is also currently used as a prototype of the general Replica Manager service.
- ***Spitfire***, which provides a secure, Grid-enabled interface for access to relational databases using Globus GSI authentication.

Grid Monitoring and Information Systems

The goal of the Grid Monitoring and Information Systems is to provide easy access to current and archived information about the Grid itself (information about resources - Computing Elements, Storage Elements and the Network) using Globus MDS, about job status (as implemented by the WMS Logging and Bookkeeping service) and about user applications running on the Grid, for performance monitoring. This permits job performance optimization as well as allowing for problem tracing, and is crucial to facilitating high performance Grid computing. The main components are as follows:

- ***MDS (Globus Monitoring and Discovery Service)***, which is a monitoring service based on soft-state registration protocols and LDAP, collecting its information from GIISs (Grid Information Index Servers), which aggregate the information from the GRISs (Grid Resource Information Servers) running on each resource.
- ***Ftree***, which is an EDG-developed alternative to the Globus LDAP with improved caching.
- ***R-GMA (Relational Grid Monitoring Architecture)***, which makes information from producers available to consumers as relations (tables).
- ***GRM/PROVE***, which is an application monitoring and visualization tool of the P-GRADE graphical parallel programming environment, modified for application monitoring in the DataGrid environment.

Fabric Installation and Job Management Tools

The EDG collaboration has developed a complete set of tools for the management of PC farms (fabrics), in order to make the installation and configuration of the various nodes automatic and easy for the site managers, and for the control of jobs on the Worker Nodes.

The fabric installation and configuration management tools are based on a remote install and configuration tool called LCFG (Local Configurator), which, by means of a server, installs and configures remote clients, starting from scratch, using a network connection to download the required RPM files for the installation, after using a disk to load a boot kernel on the client machines.

The Storage Element

The Storage Element has an important role in the storage of data and the management of files in the Grid domain.

A Storage Element is a complete Grid-enabled interface to a Mass Storage Management System, tape or disk based, so that mass storage of files can be almost completely transparent to Grid users. A user should not need to know anything about the particular storage system available locally to a given Grid resource, and should only be required to request that files should be read or written using a common interface. All existing mass storage systems used at testbed sites will be interfaced to the Grid, so that their use will be completely transparent and the authorization of users to use the system will be in terms of general quantities like space used or storage duration.

Initially the supported storage interfaces will be UNIX disk systems, HPSS (High Performance Storage System), CASTOR (through RFIO), and remote access via the Globus GridFTP protocol. Local file access within a site will also be available using Unix file access, e.g. with NFS or AFS. EDG are also developing a grid-aware Unix filing system with ownership and access control based on Grid certificates rather than local Unix accounts.

Data Access

Grid applications need different types of resources, such as CPU, memory, network bandwidth and secondary storage. The grid middleware tries to allocate these resources to the applications in a way the execution can be as fast and as efficiently as possible.

Many applications read and write large data files during their operations. Often these files are not available on the computing element where the application runs. The optimal strategy for accessing these files depends on the kind of the application: computation-intensive applications may spend long periods of time between successive data accesses, while data-intensive applications may need to access large files very frequently. Consequently a small puffer may be enough for storing a local copy of the data needed in the next step of computation or a complete copy of a huge file may be necessary in advance. The access method as well as the granularity of the read/write operations may vary with each input and output file used. Therefore the optimal data access may require the coping of some files as a whole, as well as accessing other files in parts.

In the EDG architecture three services must be mentioned regarding the data access:

- the Resource Broker, which is responsible for scheduling,
- the Replica Catalogue, the data files are access trough, and
- the Replica Manager, which manages the creation of file replicas on different Storage Elements.

The scheduling and match-making algorithms used by the RB are the key to making efficient use of Grid resources. The job can then be sent to the site which minimizes the cost of network bandwidth to access the files.

Data Access Patterns

Both logical and physical files can carry additional metadata in the form of "attributes". Logical file attributes may include items such as file size, CRC check sum, file type and file creation timestamps.

In order to optimize the resource allocations of the different jobs additional information is required. The introduction of the ***data access patterns*** [HKU] may represent the source for this information. They will store the name of the files that are accessed by the application, the type of the operation (read/write/append), the amount of data accessed in one step and the frequency and type (sequential or irregular order) of data access operations.

To provide a simple and powerful interface for the caching of data files and for the scheduling the description of file accesses is block-based. In the case of read and write operations the data access pattern contains the following information:

- ***offset*** – the beginning of the first block to be accessed
- ***length*** – the length of one block
- ***stride*** – the distance between two consecutive blocks
- ***ratio*** – the ration of the amount of data accessed in one block and the block length
- ***method*** – the way the job will access the input blocks, specified by the keywords: ***seq*** in case of sequential processing and ***rand*** in case of undefined access.

In the possession of this block-based description of the file accesses the Resource Broker can optimize the overall performance of the grid by the sending and retrieving only the required file data blocks to and from the Storage Elements. Obviously this optimization is based also on the information about the other resources, but without the ability to partition the data files in a correct manner only the complete replication of files can be used for speeding up the job executions.

The information stored in these data access patterns may be different according to the type of the operations executed on the data files.

Every input file is associated with a data access pattern, witch will store the fields presented lately.

In the case of the output files there are two kinds of write methods:

- modify, which means that the job changes data in the file
- append, which shows that the job will add data to the end of the output file.

In the first case the data access pattern associated with the output file will store the same fields as in the case of the input files, while when the write method is append, the data access pattern associated with the output file will store only the ***length*** field, specifying the amount of data to be written.

Operation patterns

Similarly to the data access patterns can be introduced the ***operation patterns***. These will store information about the operations executed in a job in the following fields:

- ***fixed*** – the number of fixed point basic operations completed by the operation (i.e. fixed point addition)
- ***floating*** – the number of floating point basic operations completed by the operation (i.e. floating point division)
- ***file:amount*** – the input file and the amount of data that will be processed by the operation (multiple file:amount descriptions are allowed).

Extending the JDL

A job is defined using the JDL language, which specifies the input data files, the code to execute, the required software environment, and lists of input and output files to be transferred with the job. The user can also control the way in which the broker chooses the best-matching resource.

The Job Description Language

The JDL is a fully extensible language, hence it is allowed to use whatever attribute for the description of a job. Anyway only a certain set of attributes that we will refer as "supported attributes" from now on, is taken into account by the Workload Management System components in order to schedule a submitted job.

The supported attributes can be grouped into two main categories:

- ***resources attributes***, which are used to build the expressions of the ***Requirements*** and ***Rank*** attributes in the job class-ad and which have to be effective, i.e. to be actually used for selecting a resource, and have to belong to the set of characteristics of the resources that are published in the Grid Monitoring and Information Service (MDS) (such as the operating system required, the amount of memory required, the amount of time required, etc.)
- ***job attributes***, which represent instead job specific information and specify in some way actions that have to be performed by the RB to schedule the job (such as the job name, command to execute, command line options, etc.). Some of these attributes are provided by the user by editing the job description file while others (needed by the RB) are inserted by the User Interface (UI) before submitting the job.

A small subset of the attributes that are inserted by the user are mandatory, i.e. necessary for the RB to work correctly and can be split in two categories:

- ***mandatory***: the lack of these attributes does not allow the submission of the job
- ***mandatory with default value***: the UI provides default value for these attributes if they are missing in the job description [JDL01].

The ***resources attributes*** include the Computing Element, Close Storage Element, Storage Element and Storage Element Protocol entities attributes. Anyway some of the attributes published in the MDS shall not be used by the user to build the ***Requirements*** and ***Rank*** expressions since they are automatically taken into account by the RB for carrying out the match-making algorithm.

The Resource Broker is sensitive to upper/lower case of attribute names, so the requirements and rank attributes are always passed lower case by the UI while other JDL attributes are passed so as they are written by the user in the job description.

The ***Requirements*** attribute is a boolean ClassAd expression that using C-like operators represents job requirements on resources. In order for a job to run on a given queue, this ***Requirements*** expression must evaluate to true on it.

The ***Rank*** attribute is a floating-point ClassAd expression that defines preference, a higher numeric value meaning better rank. The RB will give to the job the Compute Element (CE) queue with the highest rank.

The format of the JDL must be human readable, relatively easy to understand and create with a simple line editor, and easily parsed.

The extension

In order to optimize job scheduling and file access the JDL must be extended. The extended JDL description should be able to specify the amount of computing and secondary storage resources required by a job. Moreover, this description should enable the scheduling service to calculate the required network bandwidth. The description should be able to specify the secondary storage access method as well, allowing the Replica Manager (or the future EDG Grid FS implementation) to optimize file access parallel to the job execution.

The JDL extension is based on a simplified job model. According to this model, the jobs consist of processing steps. During a processing step, the job opens its input files, reads the input data, completes operations on the input data, and writes the result of processing operations. The JDL extension will describe the resource (computing and secondary storage) consumption of the processing steps. [HKU]

The optimal data access strategy can be determined only if the Resource Broker and the Replica Manager gets sufficient amount and quality of information about the data access patterns of the application. To communicate this information the following ***job attributes*** were introduced:

- ***InputPattern*** – a list of the input files and the block-based description of the read operations executed on each of them
- ***OutputModifyPattern*** – a list of the output files and the block-based description of the write operations executed on each of them
- ***OutputAppendPattern***. – a list of the output files and the amount of data written to each of them.
- ***OperationPattern*** – a list of the operation descriptions of the given job

The block-based description of the read and write operations contain the same information as in the case of the data access patterns mentioned above:

- ***offset***
- ***length***
- ***stride***
- ***ratio***
- ***method.***

The description of the operations is also identical with the one of the operation patterns:

- ***fixed***
- ***floating***
- ***file:amount.***

The elements of the lists have the following form:

```
(“file name”, offset, length, stride, ratio, method)
```

in the case of ***InputPattern*** and ***OutputModifyPattern,***

```
(“file name”, length)
```

in the case of the ***OutputAppendPattern*** and

```
(fixed, float, files1:amount1, file2:amount2, …)
```

in the case of the ***OperationPattern***.

Examples:

```
InputPattern = { ("example1.dat", 0, 40000, 0, 1, seq) }
```

The example specifies a description of a simple input file access: the job sequentially reads the whole "example1.dat" file (the size of the file is 40000 byte).

Analyzing this description, the Replica Manager can decide on not to replicate the whole input file on the Compute Element that runs the job, but to pre-fetch the beginning of the file and read the rest into a relatively small buffer parallel with the job execution. This strategy can minimize the time from job submission to job termination.

```
OutputAppendPattern = { ("example2.dat", 4000) }
```

According to the example, the results will be appended to the output file "example2.dat" (the amount of data to be written is 4000 bytes long).

```
OperationPattern = { (100, 0, "example1.dat":40000) }
```

The example shows that the job reads the "example1.dat" input file. In each processing step 100 fixed point basic operations are completed, and 40000 bytes are processed.

The whole example is the following:

```
InputPattern = { ("example1.dat", 0, 40000, 0, 1, seq) }
OutputAppendPattern = { ("example2.dat", 4000) }
OperationPattern = { (100, 0, "example1.dat":40000) }
```

The example JDL fragment specifies a job that sequentially reads "example1.dat" into a 40000 bytes long buffers, calculating for example in each processing step the sum of 100 bytes (integers) stored in the buffer, and writing out these sums into "example2.dat".

Implementation

Concerning grid application development the extension of the JDL has two major goals:

- make the work of the specialists (physicist, biologists, etc.) developing applications for Grid easier,
- enable Grid services to apply further optimization techniques in order to achieve shorter job execution times.

Each grid application has to solve two very different problems:

- it has to provide the results required by the scientists
- it has to be able to efficiently use and interact with the grid services.

Today, these aspects of grid application are not separated. For example, the MPI2 communication codes are mixed with the "real" application code computing the required scientific results. In addition, scientists have to deal with the problems of parallel and distributed computing, and the communication between distributed processes.

In the future the grid interaction has to be separated from the scientific calculations. So the scientists should concentrate on the implementation of scientific calculations, and use pre-implemented grid interaction. Consequently, a future grid application will have two major modules:

- the grid interaction code module and
- the problem code module.

The final code of the application will be generated from these modules by a weaver. The created code can be compiled than with the chosen compiler.

The grid interaction code modules may form an extensible library. As a result scientists will not have to deal with this aspect of their applications but choose the appropriate interaction code to be woven with their application code.

The grid interaction code of a given application could produce the information required for optimization. This information will be used to generate the extended JDL description for the job, which will contain:

- the JDL description provided by the user and
- the JDL extension generated automatically.

Definition of Data Access Patterns

The definition of data access patterns can be accomplished in different ways:

- can be specified by the programmer or
- can be derived from data collected by a monitoring tool.

Those applications which have meta-descriptor files attached to them can be included in the first case too. These descriptor files were written by the programmers. The information required for creating the data access patterns can be achieved than from these files with a suitable tool (the tool has to know both the form of the meta-descriptor files and the data access patterns).

Monitoring data access

In many cases data access patterns for an application will not be specified. One possible reason is that the author of the application is not able to specify them. Another possible reason is that the exact behavior and source code of the application are not known to the scientist who wants to execute the application on the grid. In such cases monitoring can still be used to define data access patterns.

Many applications are executed not only once but several times, for different input values. Parameter sweep applications are a good example. Jobs originating from the same application are likely to share data access patterns. If such jobs can be identified, monitoring the first few executions of the application can provide information to the grid middleware to optimize resource allocation for any subsequent execution. Middleware products supporting parameter sweep applications already exist for the grid. Using such a middleware can make the identification of jobs originating from the same application transparent to the scientists.

The monitoring of jobs can be done in the following ways:

- altering the source code of the application
- altering the compiler
- altering the run-time system
- altering the operating system.

There are a number of possibilities to merge monitoring code with the code of an application.

The software provided by the EDG collaboration currently is based on the Linux Red Hat 6.2 platform. So in the following these possibilities will be discussed focusing on the programming language C (other languages should be handled in a similar way).

Monitoring the data access is based on monitoring standard file handling operations defined in the ***stdio***, ***fcntl*** and ***unistd*** libraries.

The IO functions of the ***stdio*** library [STDIO] that need monitoring are (a short description of the given functions is also presented):

- tmpfile – creates a temporary file
- fclose – closes a stream
- fcloseall – closes all the opened streams
- fopen – opens the file whose name is pointed by a string and associates a stream with it
- freopen – opens the file whose name is pointed by a string and associates a stream with it; the original stream (if it exists) is closed
- fdopen – creates a new stream that refers to an existing system file descriptor
- fopencookie – creates a new stream that refers to the given magic cookie, and uses the given functions for input and output
- fprintf – writes formatted output to a stream
- vfprintf – writes formatted output to a stream from the given argumentum list

- vdprintf – writes formatted output to a file descriptor from the given argumentum list
- dprintf – writes formatted output to a file descriptor
- fscanf – reads formatted input from a stream
- vfscanf – reads formatted input from a stream into an argument list
- fgetc – reads a character from a stream
- getchar – reads a character from stdin
- _IO_getc – reads a character from a stream
- getc_unlocked – is equivalent with fgetc_unlocked
- getchar_unlocked – reads a character from stdin without locking the stream
- fgetc_unlocked – reads a character from a stream without locking the stream
- fputc – writes a character to a stream
- putchar – writes a character to stdout
- _IO_putc – writes a character to a stream
- fputc_unlocked – writes a character to a stream without locking the stream
- putc_unlocked – is equivalent with fputc_unlocked
- putchar_unlocked – writes a character to stdout without locking the stream
- getw – reads a word (int) from a stream
- putw – writes a word (int) to a stream
- fgets – reads a newline-terminated string of finite length from a stream
- fgets_unlocked – reads a newline-terminated string of finite length from a stream without locking the stream
- gets – reads a newline-terminated string from stdio
- __getdelim – is equivalent with getdelim
- getdelim – reads up to (and including) a delimiter from a stream into a string pointer
- getline – reads up to (and including) a newline from a stream into a string pointer
- fputs – writes a string to a stream
- fputs_unlocked – writes a string to a stream without locking the stream
- puts – writes a string, followed by a newline, to stdout
- ungetc – pushes a character back onto the input buffer of a stream
- fread – reads elements of data from a stream
- fwrite – writes elements of data to a stream
- fread_unlocked – reads elements of data from a stream without locking the stream
- fwrite_unlocked – writes elements of data to a stream without locking the stream
- fseek – seeks to a certain position on a stream
- rewind – rewinds to the beginning of a stream
- fseeko – seeks to a certain position on a stream

- fsetpos – sets a stream’s position
- popen – creates a new stream connected to a pipe running the given command
- pclose – closes a stream opened by popen

The monitoring of the “_unlocked” version of the functions is also very important, while these functions are faster compared to the normal ones, due to fact that in these cases no file locking is performed.

Though the _IO_getc and _IO_putc functions are defined in the ***libio*** library, they should be monitored as well, since the C standard explicitly says that the getc and putc functions defined in the ***stdio*** library can be macros, and their current implementation calls directly the _IO_getc and _IO_putc functions.

The information gathered from the monitored functions may be different form function to function. They are:

- the stream id for the tmpfile function
- the stream id and the result code for the fclose and pclose functions
- the result code for the fcloseall function
- the stream id, the filename, the opening mode and the file size for the fopen and freopen functions
- the stream id and the amount of the bytes written for the fprintf, vfprintf, fputc, putchar, _IO_putc, fputc_unlocked, putc_unlocked, putchar_unlocked, putw, fputs, fputs_unlocked, puts, ungetc, fwrite and fwrite_unlocked functions
- the file descriptor and the amount of the bytes written for the dprintf and vdprintf functions
- the stream id and the amount of the bytes read for the fscanf, vfscanf, fgetc, getchar, _IO_getc, getc_unlocked, getchar_unlocked, fgetc_unlocked, getw, fgets, fgets_unlocked, gets, __getdelim, getdelim, getline, fread and fread_unlocked functions
- the stream id, the offset and the position whence the offset is measured for the fseek, rewind, fseeko and fsetpos functions
- the stream id, the command and the opening mode for the popen function.

The IO functions of the ***fcntl*** library [FCNTL] that need monitoring are (a short description of the given functions is also presented):

- open – opens a file and returns a new file descriptor for it
- creat – creates and opens a file and returns a new file descriptor for it

The information gathered for these functions are the file descriptor, the filename and the opening mode flags.

The IO functions of the ***unistd*** library [UNISTD] that need monitoring are (a short description of the given functions is also presented):

- lseek – seeks to a certain position on a file descriptor
- close – closes a given file descriptor
- read – reads data from a file descriptor
- write – writes data to a file descriptor
- pread – reads data from a file descriptor from a given offset
- pwrite – writes data to a file descriptor at a given offset
- pipe – creates a pair of file descriptors, pointing to a pipe inode
- dup – create a copy of the given file descriptor
- dup2 – makes the new file descriptor be the copy of the old file descriptor, closing the new file descriptor first if necessary
- truncate – truncates a file given by its name to the given length
- ftruncate – truncates a file given by its file descriptor to the given length

The information gathered for these functions are:

- the file descriptor, the offset and the position whence the offset is measured for the lseek function
- the file descriptor and the result code for the close function
- the file descriptor and the amount of the bytes read for the read function
- the file descriptor and the amount of the bytes written for the write function
- the file descriptor, the offset and the amount of the bytes read for the pread function
- the file descriptor, the offset and the amount of the bytes written for the pwrite function
- the file descriptors for the pipe, dup and dup2 functions
- the filename and the length for the truncate function
- the file descriptor and the length for the ftruncate function.

The data collected by monitoring the jobs must be further processed in order to get the required definition of the data access patterns. These definitions can be used than to extend the descriptor of the job (using an extension of the JDL language), which in turn can be used by the grid middleware services to perform the optimization.

For processing the collected data I have written a special ***parser tool***, which analyzes the monitored data and generates the extensions for the current JDL description of the application.

Altering the source code

A possibility to extend an application with monitoring code is to ask the author of the application to modify the original source code. Obviously this modification should not be subtle, otherwise a scientist will not be able to perform it. Fortunately, with an appropriate set of C macros much of the necessary code transformations can be performed. Using such a set of macros and a set of new, monitored libraries the required modification would be to replace some include statements with other ones.

In many cases this little modification of the source code is affordable – though, in many other cases it is not. For example, if the scientist who wants to run the application on the Grid has limited programming skills, he might not be able to perform even such a simple modification. Despite this, the major drawback of this approach is the requisiteness that the source code of the original application has to be available. Finally, the expressive power of C macros might not be sufficient for inserting certain kinds of monitoring code into the code of the application.

During the research phase of the current paper I have written the following monitored libraries: ***mon_stdio***, ***mon_fcntl*** and ***mon_unistd***. Every function mentioned above (that needs monitoring) has a correspondent in these libraries. The name of these functions is composed from the name of the original function and the "_mon_" prefix. To preserve full compatibility with the original libraries the monitored functions are only defined if the correspondent functions are defined too — this depends on the definition of a set of macros, using the #define directive in the source code or preprocessor options during compilation (e.g. -Dmacro in case of gcc); for example:

```
#ifndef __USE_FILE_OFFSET64
/* Open a file and create a new stream for it.  */
FILE *_mon_fopen (__const char *__restrict __filename,
             __const char *__restrict __modes) __THROW;
/* Open a file, replacing an existing stream with it. */
FILE *_mon_freopen (__const char *__restrict __filename,
               __const char *__restrict __modes,
               FILE *__restrict __stream) __THROW;
#endif
#ifdef __USE_LARGEFILE64
FILE *_mon_fopen64 (__const char *__restrict __filename,
               __const char *__restrict __modes) __THROW;
FILE *_mon_freopen64 (__const char *__restrict __filename,
               __const char *__restrict __modes,
               FILE *__restrict __stream) __THROW;
#endif
```

The information collected by the functions is written in a log file. The body of the new functions consists of the following steps:

- execute the original function
- open the log file
- write the name of the function and the gathered information to the log file
- close the log file
- return the result obtained in the first step.

Beside the libraries I have also created three header files: ***monstdio.h***, ***monfcntl.h*** and ***monunistd.h***. They contain the macro definition to transfer calls to the ***stdio***, ***fcntl*** and ***unistd*** functions to the corresponding monitoring functions; for example:

```
#ifndef __USE_FILE_OFFSET64
#define open _mon_open
#endif
#ifdef __USE_LARGEFILE64
#define open64 _mon_open64
#endif
```

In order to create a monitored version of the given application the source files must include ***monstdio.h*** instead of ***stdio.h***, ***monfcntl.h*** instead of ***fcntl.h*** and ***monunistd.h*** instead of ***unistd.h***, and the monitoring libraries must be linked with the application too.

Altering the compiler

Another possibility is to leave the source code untouched, but intervene during the compilation process. The source code of the application can be altered by a special phase during the execution of the compiler.

This kind of code transformation probably has more expressive power than the previous approach. Another advantage is that it does not require the scientist to make any changes to the source code of the application. To obtain a monitored version of his application he only needs to compile the application with an appropriate compiler. It is the task of the site administrator, a person with suitable skills, to provide the necessary compiler or makefile settings.

The drawback of this approach is that the source code of the application should still be available and that the recompilation must be done on a suitably configured computer.

Many tools exists that enable parsing and transforming C source code. Using a wide-spread tool, like javacc can be achieved the same functionality as that presented in the previous section.

Altering the run-time system

The third possibility is to interfere during run-time, when shared libraries are linked to the application. It is possible to hack the run-time system on a Computing Element node in a way that certain IO function calls are not dispatched to the default IO libraries, but to a custom shared library which performs monitoring as well as executing the actual IO functions.

This is a very flexible solution. Only an environment variable has to be modified in the run-time system from non-monitored to monitored applications. Another advantage is that the source code of the original application is not needed, since no recompilation is necessary. A special shared library must be available on the Computing Element nodes, and an environment variable must be set (according e.g. to the job description), in order the data accesses of the job to be monitored or not. This kind of monitoring is completely hidden in the Grid middleware, and is transparent to the scientist who executes the application.

A disadvantage of this approach is that it is platform-dependent. The techniques required by this approach can be used on Linux and Solaris machines; other operating systems require further investigations.

Using the same technique as the fakeroot tool of Debian Linux systems [FAKEROOT], the shared libraries containing the monitoring versions of the file manipulation functions are preloaded during the run-time linking of the Grid job on a Computing Element node. As result the application calls to standard IO functions will be dispatched to these shared libraries, which at there turn will dynamically load and execute the appropriate IO functions using the same API as used by ld.so, the dynamic linker.

Altering at the operating system level

Changing the operating system slightly also offers a solution. This can be done by extending the grid filesystem, which is a part of the operating system of the Computing Element nodes, with the possibility of monitoring file accesses. A drawback of this approach is that only the file accesses performed through the grid filesystem (namely remote file accesses) could be monitored this way.

Generating the description of the resources

Gathering the data

During the execution of a job the written monitored libraries (***mon_stdio***, ***mon_fcntl*** and ***mon_unistd***) will generate a logfile. The lines of this logfile have the following form:

```
<function_name param_name1=”value1” param_name2=”value2” … >
```

where:

- ***function_name*** is the name of the monitored function
- ***param_name1, param_name2, …*** are the names of the monitored information
- ***value1, value2, …*** are the values of the monitored information.

Processing the gathered data

The logfile generated in the previous phase is processed here by parsing it line by line. During this procedure two lists are generated for every file:

- one for the read and
- one for the written data.

The program detects if one or more streams and / or one or more file descriptors refer to the same file (the logged read, write, seek, etc. operations are always linked to streams or file descriptors) and treats these references as one. It also tracks the multiple open-close cycles performed on the same file.

These lists are composed from data blocks. Every block stores the following information:

- the starting position of the block in the file
- the ending position of the block in the file
- the amount of bytes being accessed in the current block
- the type of the access (sequential or random)

The ***parser tool*** has to main parts:

- first it parses the logfile and generates the two lists mentioned above for every accessed file,
- than a special algorithm analyses these lists and generates the extensions for the current JDL description of the monitored application.

The algorithm used in the first phase starts parsing the logfile and:

- if the function in the current line is a ***file open*** related function then it links to the file the current stream or file descriptor specified as parameter for the current function and creates two empty block-lists registering them to the current file if no lists were registered for the file yet, or it sets the status of the file to “opened status” if there are lists already registered to the file;
- if the function in the current line is a ***file close*** related function then it sets the status of the file to “closed status” and unlinks the streams and file descriptors associated with the file if the parameter stream or file descriptor links to a file or does nothing if it don’t;
- if the function in the current line is a ***file seek*** related function then it updates the current file position pointer of the file linked to the stream or file descriptor parameter of the current line, if such link exists, or does nothing if it don’t;
- if the function in the current line is a ***file read / write*** related function and the stream or file descriptor parameter of the current line links to a file then it creates a new sequential block in the correspondent block-list if the range of the current operation does not intersect with another block already defined or it updates the range and the type of the intersected block (or eventually merges multiple blocks) if such intersection occurs.

The algorithm does not contain any error checking for detecting the wrong order of the operations (close before open, read after close, etc.), however if this would be needed the required error detections and error feedbacks could be added very easily to the parsing tool.

After finishing the parsing every list will contain the final set of blocks, having these properties:

- every block is separated from the others in the aspect of access time
- if i < j then block[i] is accessed none but before block[j]

The second phase a special algorithm examines every generated block-list, determining the data access pattern for every file and for every operation. The definition of data access patterns divides the files in data blocks. The nth block starts at offset+(n-1)*(length+stride) and ends at length+ the starting position. Every valid data access pattern must correspond to the followings:

- every data block should contain valid data (I will call ***valid data*** the data the application access through read or write operations)
- no valid data should be found outside the blocks

The optimal data access pattern must also have the smallest block length out of the definable data access pattern.

The algorithm tries to found the ***offset***, ***length*** and ***stride*** that will define this optimal data access pattern. This is based on the sequential structure of the file (block[i] is accessed none but before block[j] if i < j).

A new block concept is introduced here. This block has neither necessarily fixed size, nor necessarily fixed starting offset. It is describe using four fields:

- blockMin – the minimal size of the
- blockSize – the current size of the block
- minExt – the minimal possible extension of the current block
- maxExt – the maximal possible extension of the current block

The (possible) extension of a block affects both the size of the block and its starting offset.

The algorithm is given by a depth first search. For every sub-tree a new block is created (n is the number of the sub-tree):

- blockMin – is initialized with the difference between the ending position of the nth block in the list and the value of firstPos
- blockSize – is initialized with the difference between the starting position of the block after the nth block in the list and the value of firstPos
- minExt – is initialized with 0
- maxExt – is initialized with value of firstPos;

where firstPos is the starting position of the first block in the list.

The program then tries to fit the current block to the whole file (the rest of the list). The starting and ending positions of the nth occurrence of this block are given by the formulas:

```
start[n] = firstPos - minExt + (n-1) * blockSize

end[n] = start[n] + blockSize + maxExt
```

If the nth occurrence of the current block does not contain any valid data (i.e. a block from the list), the blockSize is wrong and the algorithm must step back.

If this occurrence intersects with one or more blocks from the list, the current block is split into multiple parts or its fields are changed depending on the type of the intersection, and the algorithm continues with the modified block or it will fork according the new blocks.

The program runs until the first block is found that can be fitted to the whole file. Then it calculates the minimal ***distance*** between the ending position of the nth occurrence of the result block and the ending position of the last block element from the list that intersects with the current occurrence.

The data access pattern for the current file has the following parameters:

```
offset = firstPos – extMin

stride = distance

length = blockSize – stride

ratio = the ratio between the maxmimum amount of data accessed
in any of the occurrences of the result block and the value of
the length paramter

method = seq if all of the blocks in the list where accessed
sequentially or rand otherwise
```

When all the data access patterns are calculated they are transformed in the form of an extension for the JDL, and this JDL extension is written to the standard output of the tool.

Conclusions

The grid middleware (the Resource Broker and the Replica Manager) should ensure that applications are executed on the grid as efficiently as possible. Optimizing data access is an important aspect in this respect.

The introduction and the specification of data access patterns for applications should provide sufficient amount and quality of information for both the Resource Broker and the Replica Manager to determine and use the optimal data access strategy for the applications.

The extension of the Job Description Language with the proposed attributes as well as the solutions for gathering this information (through the presented monitoring methods) and transforming it in a useable form (either with the written tool or with other special tools) can be used to communicate the required data in the grid system.

However it should be further investigated what kind of information about data access patterns would be best to use for the optimization by the grid middleware. The usefulness of different kinds of information could be measured through simulations (e.g. OptorSim is a simulation package for Optor, the optimizer component of the Reptor replica manager). According to the results the created extension of the JDL should be redefined and the written monitoring libraries should be also extended to collect the necessary information to incrementally build JDL files for parameter-sweep applications.

The program designer may help also to determine the data/resource access pattern of an application by annotations, which could be used than to create the JDL specification.

And finally the use of small filtering agents in the case of applications which access huge remote files in an irregular way, should be also a good solution for optimizing the data access.

Bibliography

[BAL01] Balaton Z., Kacsuk P., Podhorszki N.: From Cluster Monitoring to Grid Monitoring Based on GRM and PROVE. Report of the Laboratory of Parallel and Distributed Systems, LPDS – 1/2000.

[CP] Condor Project
http://www.cs.wisc.edu/condor/

[EDG] The DataGrid Project
http://eu-datagrid.web.cern.ch/

[EDGGJRB] European DataGrid Project: Experiences of deploying a large scale Testbed for e-Science applications
http://hep-proj-grid-tutorials.web.cern.ch/hep-proj-grid-tutorials/presentations/Perf2002Paper.pdf

[FAKEROOT]fakeroot, Debian GNU/Linux manual
http://www.clc.unibe.ch/cgi-bin/man2html?fakeroot+1

[FCNTL] fcntl – manipulate file descriptor, Linux manual
http://www.clc.unibe.ch/cgi-bin/man2html/usr/share/man/man2/fcntl.2.gz

[FK01] I. Foster, C. Kesselman: The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann; 1999
ISBN: 1558604758

[FK02] I. Foster, C. Kesselman: The Anatomy of the Grid
Technical Report, Global Grid Forum, 2001
http://www.globus.org/research/papers/anatomy.pdf

[GGTKCG] The Grid, Globus Tool Kit, Condor – G and What?
http://charm.cs.uiuc.edu/users/mani/globus.ppt

[GUY01] Guy, L. at al: Replica Management in Data Grids, 2002.

[HKU] Horváth Z. - Kozsik T. - Ulbert A.: Towards a better optimization of resource allocation in Grid
Technical Report, 2002
http://aszt.inf.elte.hu/~grid/publ/proposals.ps

[JDL01] Fabrizio Pacini (fpacini@datamat.it): JDL Attributes – Note
http://server11.infn.it/workload-grid/docs/DataGrid-01-NOT-0101-0_6-Note.pdf

[LIV01] Livny, M. High-Throughput Resource Management. In Foster, I. and Kesselman, C. eds. The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1999, 311-337.

[RCCFLMMST] Aiken, R., Carey, M., Carpenter, B., Foster, I., Lynch, C., Mambretti, J., Moore, R., Strasnner, J. and Teitelbaum, B. Network Policy and Services: A Report of a Workshop on Middleware, IETF, RFC 2768, 2000.
http://www.ietf.org/rfc/rfc2768.txt.

[RTIF] Realizing the Information Future: The Internet and Beyond. National Academy Press, 1994
ISBN: 0309050448

[STDIO] stdio – standard input/output library functions, Linux manual
http://www.clc.unibe.ch/cgi-bin/man2html?stdio+1

[STO01] Stockinger, H. et al.: File and Object Replication in Data Grids. HPDC 2001.

[UNISTD] unistd, Linux manual
http://www.clc.unibe.ch/cgi-bin/man2html?lseek+2
http://www.clc.unibe.ch/cgi-bin/man2html?close+2
http://www.clc.unibe.ch/cgi-bin/man2html?read+2
http://www.clc.unibe.ch/cgi-bin/man2html?write+2
http://www.clc.unibe.ch/cgi-bin/man2html?pread
http://www.clc.unibe.ch/cgi-bin/man2html?pipe
http://www.clc.unibe.ch/cgi-bin/man2html?dup
http://www.clc.unibe.ch/cgi-bin/man2html?truncate