Towards an XML Based Data Grid Description Language

Rene Felder and Erich Schikuta
Institut für Informatik und Wirtschaftsinformatik, University of Vienna
Rathausstr. 19/9, A-1010 Vienna, Austria
erich.schikuta@univie.ac.at

Abstract

We present xDGDL, an approach towards a concise but comprehensive Datagrid description language. Our framework is based on the portable XML language and allows to store syntactical and semantical information together with arbitrary files. This information can be used to administer, locate, search and process the stored data on the Grid. As an application of the xDGDL approach we present ViPFS, a novel distributed file system targeting the Grid.

Keywords:

Datagrid, XML, meta information, distributed file systems, parallel and distributed I/O

1 Introduction

Today new and stimulating data-intensive problems in biology, physics, astronomy, space exploration, human genom research arise, which bring new high-performance applications with the need to store, administer and search intelligently gigantic data set spread over globally distributed storage resources [12].

We face a similar situation as in the well-known area of database systems [1], where data represents a model of the reality. Information must be searched, analyzed, administered easily and must be at hand efficiently for arbitrary applications. Consequentially data has to be attributed with meta information describing the specific semantics of the information in a standardized and processable way. This meta information allows applications to search the stored information intelligently.

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

Only a few similar approaches exist, but these are in a early state (e.g. [7]) or target mostly very specific application domains (e.g. [3] [8]).

The layout of the paper is as follows. In the next section we present xDGDL, the XML-based Data Grid Description Language, and give several examples for the usage of the language. Then we introduce shortly the Meta-ViPIOS system [6], which is a client server based I/O system supporting distributed applications on the Grid. Finally we present an prove-of-concept implementation of the xDGDL language within the ViPFS, the distributed file system component of the ViPIOS system.

2 xDGDL - the XML Data Grid Description Language

We propose the XML Data Grid Description Language (xDGDL) which aims to provide a convenient XML framework for the specification of meta information of data stored on the Grid. xDGDL is a derivative of PARSTORAGE [2], which was specifically designed as meta language for parallel IO data.

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

Focusing the Grid we have to specify a very general Grid architecture hosting our framework. From our point of view the Grid consists of an arbitrary number of *collaborations*, which are defined by an organizational domain [5], interconnected by WAN technology. In practice such a collaboration will be usually (but must not be) a coherent IT infrastructure represented by a cluster like system, which consists of a number of execution nodes. These *nodes* are *processing nodes* and/or *data (server) nodes*. The latter type provides data storage resources by a number of storage *devices* (e.g. disks, tapes, etc.). It is to note that a single data node can host an arbitrary number of devices.

2.1 The goals of xDGDL

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

2.1.1 Semantics of data

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

2.1.2 Persistency of data

Data stored without semantic information is lost (can not be reused), because the semantics is originally only in the program code of the application producing the data. Without the program the data is just a sequence of bytes without meaning. With the usage of a framework like xDGDL the data can be reused easily by any application understanding the meaning of the data. A practical Java-based example is given in [2].

2.1.3 Portability

In a distributed environment parts of data can migrate from one node/system/environment to another. On different hosting environments naturally the data formats change. However when moving data from one system to another, applications must still be able to read the data. By the description of the format the data can be interpreted and can be easily transformed to any proprietary format of the target machine [7].

2.1.4 Performance and efficiency

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

2.2 The xDGDL specification

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

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

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

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

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

It was a clear decision to choose XML as the basis for our framework due to its undeniable success within the Internet community and its acceptance as basis for beneath any standard movement in the Grid community (e.g. WSDL [4]).

2.3 The xDGDL document type definition

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

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

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

2.3.1 Document root

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

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

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

Example:

```
<PARSTORAGE VERSION="1.0"
            TIMESTAMP="testfile_twoserver">
    <TYPE>
        ...
    </TYPE>
    <ISLAND NAME="pri.univie.ac.at">
        ...
    </ISLAND>
</PARSTORAGE>
```

2.3.2 Island

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

```
<ISLAND NAME="pri.univie.ac.at">
    <SERVER HOST="vipios.pri.univie.ac.at">
    </SERVER>
</ISLAND>
```

2.3.3 Server

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

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

Example:

```
<SERVER HOST="vipios.pri.univie.ac.at">
    <DEVICE DEVICE_ID="/dev/vda1">
    </DEVICE>
</SERVER>
```

2.3.4 Device

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

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

Example:

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

2.3.5 View

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

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

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

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

Example:

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

2.3.6 Block

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

The BLOCK element consists of the following attributes:

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

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

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

corresponds to the definition on server 2:

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

2.4 xDGDL examples

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

2.4.1 A regularly distributed, two-server example

The first example introduces the structure of the xDGDL description. It uses two servers and writes data in round robin fashion to the local disks on each server: vipios.pri.univie.ac.at and vipclus9.pri.univie.ac.at.

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

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

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

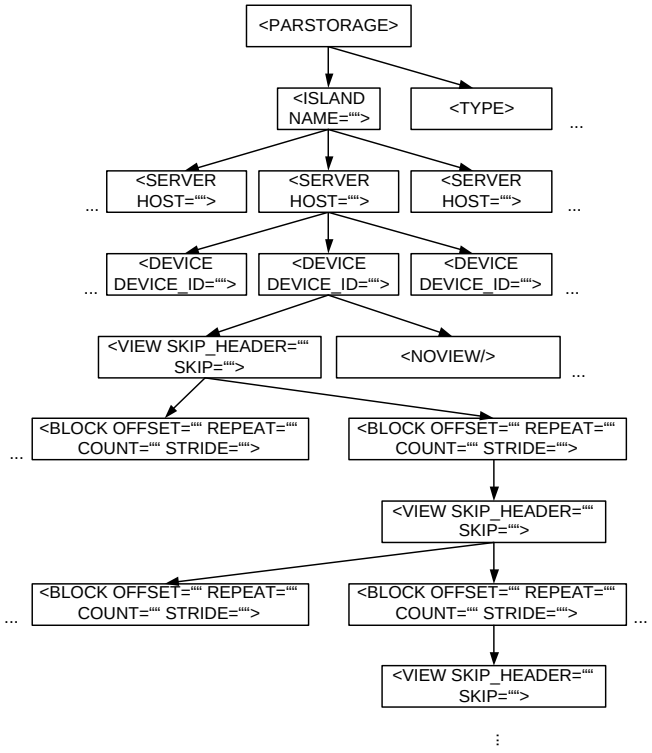

Figure 1: Example of a xDGDL tree

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

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

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PARSTORAGE SYSTEM "XDGDL.dtd">
<PARSTORAGE VERSION="1.0"
            TIMESTAMP="testfile_regular">
 <TYPE>
  <ETYPE TYPE="CHAR" LENGTH="1"/>
 </TYPE>
 <ISLAND NAME="island1.pri.univie.ac.at">
  <SERVER HOST="vipios.pri.univie.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="7">
     <BLOCK OFFSET="0" REPEAT="3"
            COUNT="5" STRIDE="7">
      <BYTEBLOCK/>
     </BLOCK>
    </VIEW>
   </DEVICE>
  </SERVER>
  <SERVER HOST="vipclus9.pri.univie.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="0">
     <BLOCK OFFSET="5" REPEAT="3"
            COUNT="7" STRIDE="5">
      <BYTEBLOCK/>
     </BLOCK>
    </VIEW>
   </DEVICE>
   </SERVER>
 </ISLAND>
</PARSTORAGE>
```

A graphical view of the regular distributed, two server example can be seen in Figure 2

2.4.2 A regular distributed, nested three-server example

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

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

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

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE PARSTORAGE SYSTEM "XDGDL.dtd">
<PARSTORAGE VERSION="1.0"
            TIMESTAMP="regular_multilevel">
 <TYPE>
  <ETYPE TYPE="CHAR" LENGTH="1"/>
 </TYPE>
 <ISLAND NAME="island3.pri.univie.ac.at">
  <SERVER HOST="vipios.pri.univie.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="12">
     <BLOCK OFFSET="0" REPEAT="2"
            COUNT="1" STRIDE="12">
      <VIEW SKIP_HEADER="0" SKIP="0">
       <BLOCK OFFSET="0" REPEAT="3"
              COUNT="5" STRIDE="7">
        <BYTEBLOCK/>
       </BLOCK>
      </VIEW>
     </BLOCK>
    </VIEW>
   </DEVICE>
  </SERVER>
  <SERVER HOST="vipclus9.pri.univie.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="12">
     <BLOCK OFFSET="0" REPEAT="2"
            COUNT="1" STRIDE="12">
      <VIEW SKIP_HEADER="0" SKIP="0">
       <BLOCK OFFSET="5" REPEAT="2"
              COUNT="7" STRIDE="12">
        <BYTEBLOCK/>
       </BLOCK>
      </VIEW>
     </BLOCK>
    </VIEW>
   </DEVICE>
  </SERVER>
  <SERVER HOST="vipclus10.pri.univie.ac.at">
   <DEVICE DEVICE_ID="/dev/vda1">
    <VIEW SKIP_HEADER="0" SKIP="0">
     <BLOCK OFFSET="29" REPEAT="2"
            COUNT="12" STRIDE="29">
      <BYTEBLOCK/>
     </BLOCK>
    </VIEW>
   </DEVICE>
  </SERVER>
 </ISLAND>
</PARSTORAGE>
```

A graphical view of the regular distributed, nested three-server example can be seen in Figure 3

3 An Application of xDGDL

3.1 The ViPIOS island

ViPIOS - the Vienna Parallel Input Output System - is an I/O system that tries to solve the well-known I/O bottleneck of high-performance computing [11]. ViPIOS was originally designed as a client-server system satisfying parallel I/O needs of high performance applications. Due to the requirements of the Datagrid initiative ViPIOS was extended to Meta-ViPIOS, which harnesses distributed I/O resources [6].

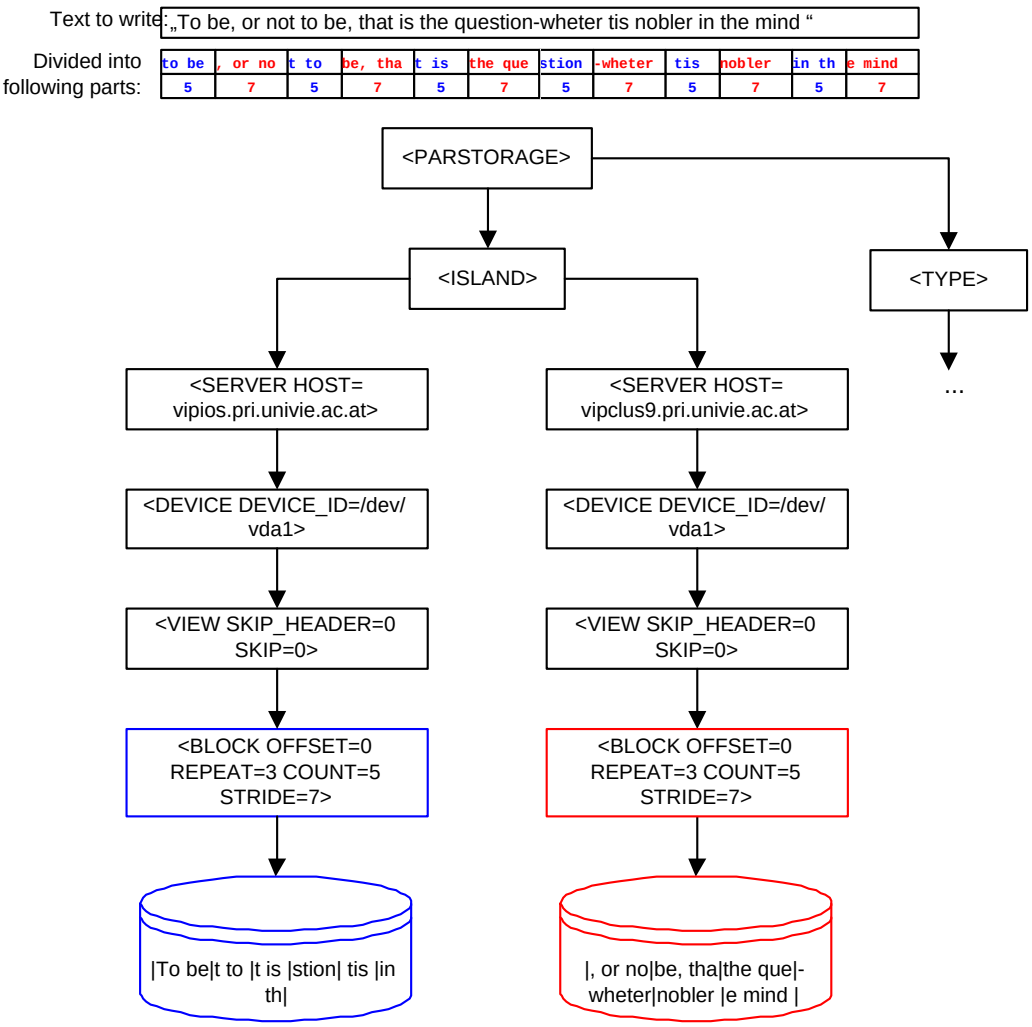

Figure 2: Tree representation of a regular distributed, two-server xDGDL distribution

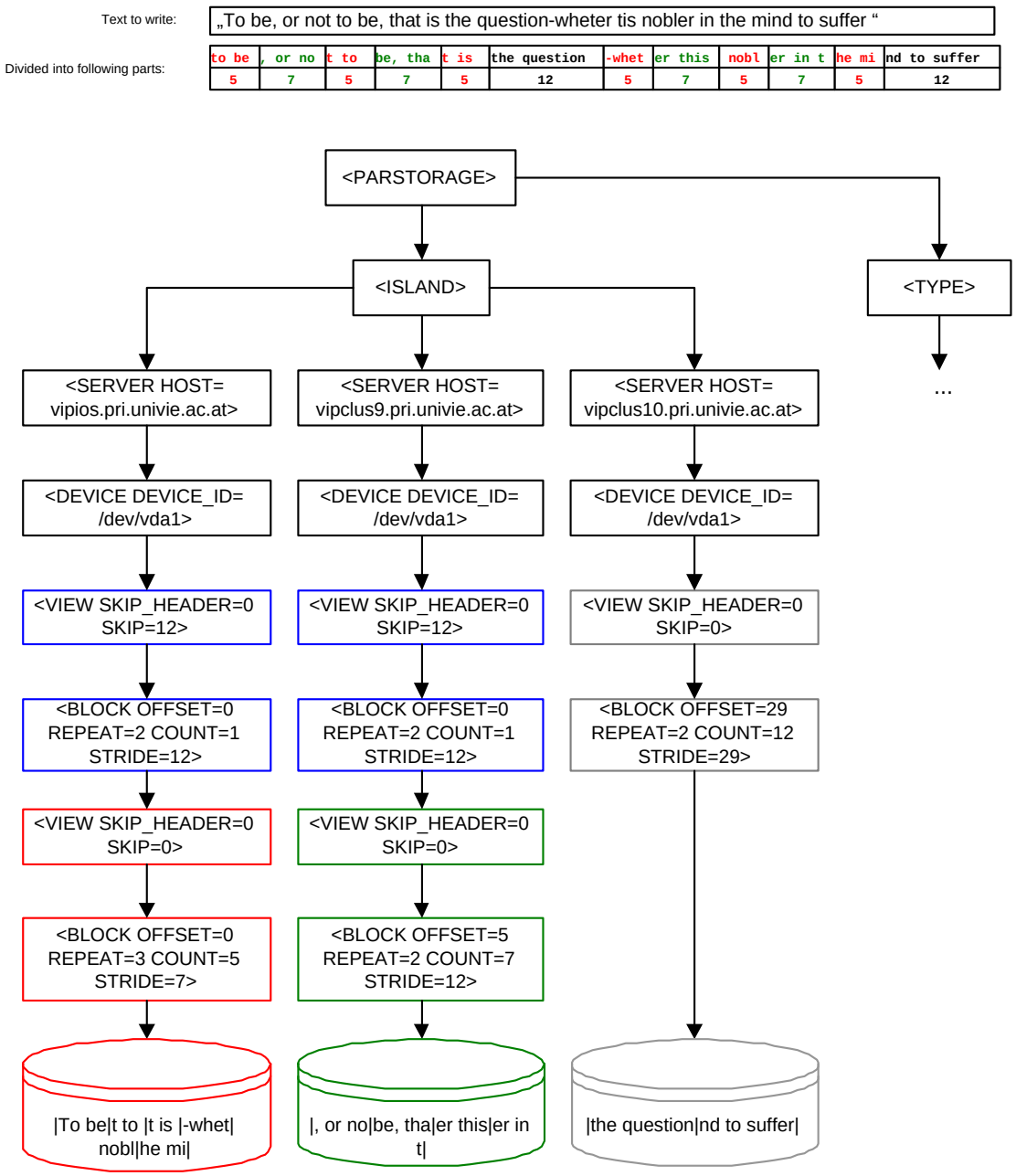

Figure 3: Tree representation of a regular distributed, nested three-server xDGDL distribution

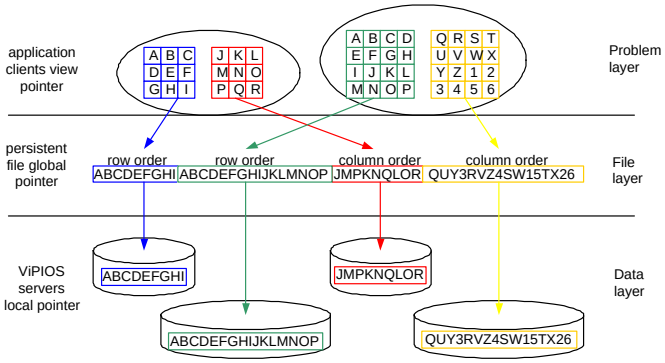

Figure 4: Different point of views: The ViPIOS layers

A *ViPIOS island* (resembling roughly a collaboration within our Grid architecture) can be seen as a logically independent system, residing on a defined set of processing nodes. Conventionally this is a typical cluster system, but it can also be an arbitrary set of world-wide distributed machines. An island comprises an arbitrary number of ViPIOS servers processing the I/O requests of connected applications. To reach such an island the client needs to know the hostname (or IP-address) of a dedicated connection server responsible for that island (for more information see [10]).

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

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

ViPIOS provides a data independent view of the stored data to the application process. It is based on a three-tier model. The three specific ViPIOS layers are the following (see Figure 4):

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

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

3.2 The ViPIOS interfaces

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

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

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

3.3 ViPFS

Basically ViPFS is a library which overloads the standard file calls in UNIX. This methods allows users easily and efficiently to employ transparently services provided by ViPIOS. Thus all Unix tools for file accesses can be used without recompiling. The idea is to redirect the calls with "conventional" data files to the standard I/O library and to redirect the calls with ViPFS data files to the ViPIOS system. This approach is similar to PVFS [9].

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

For users it is very easy to define the meta information for the data file in focus. A respective xDGDL file has to be created and stored in the same directory as the data file, which has the same name as the data file, but with the prefix "`.vd.`"[2]. With an open statement the ViPFS library checks if there is a corresponding xDGDL file for the given file. The prefixed dot is used because these files are not visible with the common `ls` command. It is also quite common to use the dot for configuration files and to a certain extent the "`.vd.*`" files can be seen as configuration files. When it is parsed, its is checked against the given data type definition (DTD). If the file is erroneous or does not exist the respective data file will be distributed with the standard distribution of ViPFS which is a cyclic distribution among the available ViPIOS servers.

Copy Example

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

The preconditions for using ViPFS are the following:

- Start of ViPIOS

[2]The prefix stands for *ViPIOS description*

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

```
MAX_APP 5 MAX_SRV_FILE 32 DATA_BUFLEN 4096
SRV_GROUP_NAME "vipios_server" SRVR_DEVICE_LIST 3
  /home/felder/ViPIOS/dev1/
  /home/felder/ViPIOS/dev2/
  /home/felder/ViPIOS/dev3/
VIP_DIR "/home/felder/vipios"
```

- Setting of Unix environment variable that points to the ViPIOS configuration file (e.g. `VIP_CONF=/home/felder/vipios/ViPIOS.conf`). The environment could be set up with the command `export`.
- Setting up the `LD_PRELOAD` environment variable. The variable must point to the `vipfsinvoke.so` shared object. In our example we set it up as follows: `export LD_PRELOAD=/home/felder/vipfs/vipfsinvoke.so`

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

In case of the example above the user can copy a data file simply by the commands shown in figure 3.3

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

In the first line we print out all .vd.* files. In our example only one distribution file is present. We used the distribution file presented in 3. That means, that the testfile was distributed among three servers with one device on each server. If we did not declare a .vd. file the testfile would have been written sequentially to the first disk on the current server.

4 Conclusions and Future Work

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

The xDGDL language is the starting point for a new way of defining data access paths on the Grid. We work on a research project to define Grid I/O patterns, which allow to define I/O data streams on the Grid easily. A stream can be seen as a graph where the vertices are modules, which are instantiated from Grid I/O patterns, and the edges are the data streams. Data is moved along such streams and carries along from vertex to vertex its self-describing information based on the xDGDL language. This allows the modules, which in fact are active I/O resources (Grid fabrics), as distributed file systems, database systems, etc., to interpret and to process the data. We work on a method for the automatic generation of such Grid I/O graphs based on heuristic methods [13].

Acknowledgement

The work described in this paper was partly supported by the Special Research Program SFB F011 AURORA of the Austrian Science Fund.

Appendix: xDGDL DTD

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

<!-- (c) Andras Belokosztolszki-->
<!-- 2000 -->
<!-- (c) Rene Felder -->
<!-- 2001 -->


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

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


<!-- hpf data structure -->
<!-- Intrinsic Data Types -->
<!ELEMENT TYPE (ETYPE|ARRAY|TYPE)+>
<!ATTLIST TYPE TYPENAME CDATA #IMPLIED>
<!ATTLIST TYPE NAME CDATA #IMPLIED>

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

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

```
felder@vipios:~/vipfstests > ls -al .vd.*
-rw-r-----   1 felder   users  1177 Oct 14  2001 .vd.testfile

felder@vipios:~/vipios > cp testfile /home/felder/vipios   # copy in
felder@vipios:~/vipios > cp /home/felder/vipios/testfile . # copy out

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

Figure 5: ViPFS copy of a data file

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


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


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

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

<!-- Model Device-Descriptor -->
<!ELEMENT DEVICE (VIEW|NOVIEW)>
<!ATTLIST DEVICE DEVICE_ID CDATA #REQUIRED>

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

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

References

[1] S. Sudarshan Abraham Silberschatz, Henry F. Korth. *Database System Concepts*. McGraw-Hill, 1996.

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

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

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

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

[6] Thomas Fuerle, Oliver Jorns, Erich Schikuta, and Helmut Wanek. Meta-vipios: Harness distributed i/o ressources with vipios. *Iberoamerican Journal of Research "Computing and Systems", Special Issue on Parallel Computing*, 4(2):124–142, October–December 2000.

[7] Feitelson Dror G. and Klainer Tomer. *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter XML, Hyper-media, and Fortran I/O. John Wiley and Sons, November 2001.

[8] The ncsa hdf home page. http://hdf.ncsa.uiuc.edu/.

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

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

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

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

[13] Helmut Wanek and Erich Schikuta. A blackboard method for automatic parallel i/o performance optimization. In Springer, editor, *Fifth International Conference on Parallel Computing Technology PaCT'99*, St. Petersburg, September 1999.