Ruprecht-Karls Universität Heidelberg
Institute of Computer Science
Research Group Parallel and Distributed Systems

Bachelor's Thesis

Tracing the Connections Between MPI-IO Calls and their Corresponding PVFS2 Disk Operations

Name:	Stephan Krempel
Matriculation number:	2377365
Supervisor:	Prof. Dr. Thomas Ludwig
Date:	March 31, 2006

Ich versichere, dass ich diese Bachelor-Arbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

March 30, 2006, ...

Abstract

Fast and efficient input/output is a very important aspect of today's high performance computing. Often these systems are very complex so no prediction can be made what an I/O call at the program level will trigger at the system level. One kind of such complex systems are parallel programs based on MPICH2 for message passing and PVFS2 as parallel file system. This thesis is about enhancing the environment in a way that we can find the necessary information in the traces of a program run. MPI-IO calls can now be related to disk operations. The concept of our extension of the involved software packages is to generate a unique ID for every MPI-IO call and to pass it along to the I/O-servers. After they got written into the corresponding traces we can find matching events in them and visualize them in the Jumpshot trace viewer. An easy to follow example is presented and detailed hints for future work are given.

Contents

1. Introduction

Fast and efficient I/O is still a rising aspect of today's high performance computing. One alternative to deal with that is a combination of MPICH2 for message passing and PVFS2 as parallel filesystem, resulting in an easy to use and very flexible I/O software system. The flexibility comes from a high modularity that considerably increases the system's complexity. Every data to be written to or read from a disk has to pass through several layers, so that at the end we cannot see easily what operation at the program level has triggered what operation at the system level. This circumstance makes is very difficult to do performance optimisation and to find bottle-necks in the system.

There already exist some tools for analysing MPI programs. Vampir[1] for example is capable of writing out a trace of an MPI program run. It can then be used to visualize and analyse these traces. But, it only shows high level MPI function calls and gives no deep view into the underlying system behaviour. Another approach is TAU[2] which is a toolkit for instrumentation of parallel programs for performance analysis. It comes with an automatic instrumentor tool that can be used with MPI for analysing message passing, but there is nothing to read yet about automatic instrumentation for I/O analysis. On the other side we can find Karma that is part of the PVFS2 package [1]. Its purpose is to show PVFS2 system characteristics live at runtime but isn't able to save them permanently. None of the mentioned tools can be used easily to make connections between I/O request and I/O operations. Even if we cannot give the names of all existing tools here, we are not aware of one having this capability.

Therefore our idea is to take a trace tool and write all relevant data at each level into traces at the program runtime. So we get one trace for the clients making I/O requests and one trace for the servers making I/O operations. We then merge these traces and see what information we can get in that way. The last step is to visualize as much information as possible and analyze the I/O behavior.

This thesis is structured as follows: *Chapter 2* gives an in detail overview about the system that we use and modify. The goals of the thesis are specified in *Chapter 3*. *Chapter 4* describes the design ideas and methods for our changes of the system before *Chapter 5* shows the implementation of all modifications and enhancements described. *Chapter 6* guides through the installation of the used environment and describes a small example to see the results of the work. *Chapter 7* finally summarizes the work and gives a short conclusion. At the very end in *Chapter 8* some ideas for future work are presented.

[1] Visualisierung und Analyse von MPI-Resourcen, developed by ZAM Jülich and TU Dresden, distributed by Pallas GmbH, Germany, now called Intel® Trace Analyzer

[2] Tuning and Analysis Utilities, a joint project between the University of Oregon, the LANL, and the ZAM Jülich - http://www.cs.uoregon.edu/research/tau/home.php

2. System Overview

In this chapter we will give an overview over our system environment. From a global point of view this is a GNU/Linux Cluster with MPICH2 as the message passing environment and PVFS2 as the parallel filesystem. Here we take a closer look at the functional context of the relevant parts. A detailed description how to install and setup this environment is given in chapter 6 "Example" on page 61.

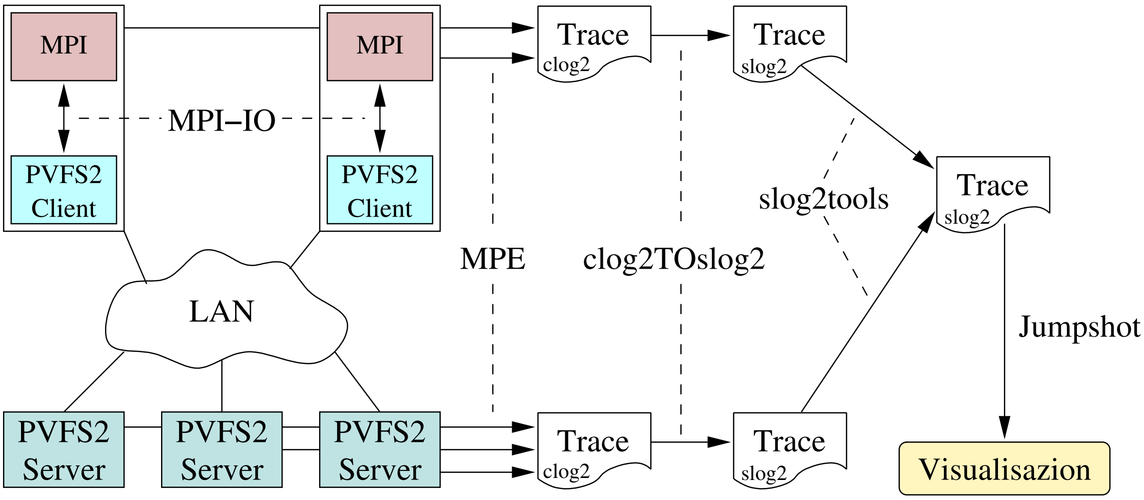


Figure 2.1.: Schematic plan of the system environment

We run parallel programs using MPI[1] for message passing. These programs do parallel input/output (I/O) using MPI-IO. The data is stored in a distributed way on hard disks using the Parallel Virtual Filesystem Version 2 (PVFS2). The operations in the clients (MPI) as well as in the servers (PVFS2 servers) are logged using the MPI Parallel Environment (MPE). The produced traces become merged and prepared using slog2tools[2] and finally get visualized with the tool Jumpshot. A schematic diagram is presented in figure 2.1.

[1] Message Passing Interface - `http://www.mpi-forum.org/`
[2] see section 2.4 "Trace Analysis Tools" on page 12 for details on the slog2tools package

2.1. MPICH2

The MPI standard [2] is the most commonly used programming method for parallel programs with message passing in our days. MPI-2 [3] including MPI-IO extends this standard with everything needed for efficient I/O in parallel programs. MPICH2[3] [5, 6] is the popular open source implementation by the Argonne National Laboratory that has attained wide acceptance in the world of high performance computing. It is not just a library, but it comes with an own compiler frontend (mpicc), daemon (mpd) and execution wrapper (mpiexec).

We are using MPICH2 version 1.0.3 released on November 23, 2005.

2.1.1. ROMIO

The part of MPICH2 that contains the functions specified in MPI-IO is called ROMIO[4] [8]. There is already support for many common filesystems and ROMIO has a modular structure to facilitate adding further ones. To support a filesystem ROMIO uses a so called ADIO[5] device implementing a couple of interface functions for one specific filesystem. An ADIO device for PVFS2 is already included.

2.2. MPE2

MPE[6] is a collections of tools for MPI programmers. A complete list of the many available tools under MPE2 can be obtained from [9]. The parts important for our development are the logging facilities at the one hand and the SLOG-2 SDK on the other.

Although MPE2 1.0.3 is included in MPICH2 1.0.3 we use the slightly newer version 1.0.3p1 from November 23, 2005.

2.2.1. Logging facilities

MPE2 provides a set of profiling libraries to collect information about the behavior of MPI programs.

[3] MPI Chameleon 2 - `http://www-unix.mcs.anl.gov/mpi/mpich2/` [4]

[4] `http://www-unix.mcs.anl.gov/romio/index.html` [7]

[5] Abstract-Device Interface for I/O

[6] MPI Parallel Environment - `http://www-unix.mcs.anl.gov/perfvis/download/index.htm#MPE`

The two of them we will use are

libmpe	-	providing a general logging interface for manual usage
liblmpe	-	providing wrappers around all MPI functions to log every MPI function call in a program.

The traces generated by MPE are written at the end of execution time for postmortem analysis. The writing is triggered by the call of `MPE_Finalize`.

2.2.2. SLOG-2 Software Development Kit

The SLOG-2 SDK provides both the Software Development and Runtime environments for SLOG-2 and Jumpshot-4.

SLOG-2

SLOG-2[7] is a visualization oriented logfile format for event logging. The data is represented by drawables internally organized in a tree structure [10]. Each drawable belongs to a category defining its type. Each time stamp (for example the start time of an event) has the time value as offset to the beginning time of the file and belongs to one time line. There can and almost always will be many time lines in as SLOG-2 formatted file. For example when MPE merges the traces from all MPI processes it creates one time line for each process.

There are four different kind of drawables:

event	-	event occurring at one single point in time
state	-	event taking a fraction of time
arrow	-	event taking a fraction of time and optionally beginning and ending on different time lines
composite	-	container to compose any drawables

The first three kinds are also called primitive drawables.

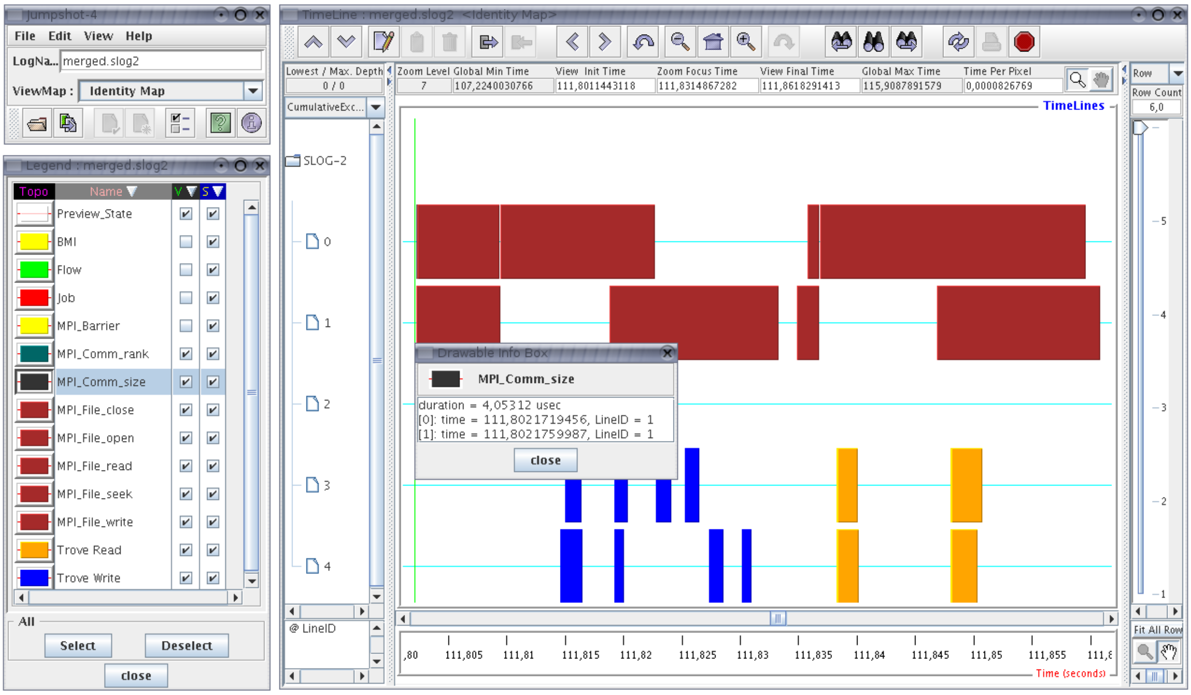
Jumpshot-4

Jumpshot[8] is a logfile viewer written in Java. It is able to show SLOG-2 formatted files.

Figure 2.2 on the next page shows the visualization of an slog2 file displayed by Jumpshot-4. The slog2 file is created using the scheme in figure 2.1 on page 7. Each of the time lines 1-4 represents one process. The rectangles on the time lines are the events that have occurred in the respective processes. For example these can be calls of MPI-IO functions (listed in

[7] `http://www-unix.mcs.anl.gov/perfvis/software/log_format/index.htm` [9]

[8] `http://www-unix.mcs.anl.gov/perfvis/software/viewers/index.htm` [9]



Topo	Name	V	S
	Preview_State	☑	☑
🟨	BMI	□	☑
🟩	Flow	□	☑
🟥	Job	□	☑
🟨	MPI_Barrier	□	☑
■	MPI_Comm_rank	☑	☑
■	MPI_Comm_size	☑	☑
🟥	MPI_File_close	☑	☑
🟥	MPI_File_open	☑	☑
🟥	MPI_File_read	☑	☑
🟥	MPI_File_seek	☑	☑
🟥	MPI_File_write	☑	☑
🟧	Trove Read	☑	☑
🟦	Trove Write	☑	☑

All
Select Deselect
close

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 0	7	107,2240030766	111,8011443118	111,8314867282	111,8618291413	115,9087891579	0,0000826769

Row Count 6,0

MPI_Comm_size
duration = 4,05312 usec
[0]: time = 111,8021719456, LineID = 1
[1]: time = 111,8021759987, LineID = 1

Figure 2.2.: Jumpshot program windows

appendix A on page 77). By doing a right click on such an event a pop-up window shows some information about the event. Each event has the color of its category as shown in the legend window on the left side. The legend window also provides the option to set the visibility and searchability of each category. For a more in detail description see [11].

2.3. PVFS2

PVFS2[9] is an open source parallel file system to be used on clusters running Linux Systems. It is a completely rewritten version of the first PVFS, both being developed at the Argonne National Laboratory.

PVFS2 has a modern modular layered design [12, 13, 14] to be flexible to changes and extensions. It provides to its clients an abstract view to a file system, hiding the real distributed nature of the data storage on multiple data servers. All the management is done by one or more metadata servers. For PVFS2 it does not matter where the client and server processes run physically.

In figure 2.3 on the following page we can see the layered design of PVFS2. The implementation of PVFS2 is based on state machines handling separate tasks. Each layer is implemented by at least one state machine, often with several sub state machines. For a detailed description

[9] Second Parallel Virtual Filesystem - http://www.pvfs.org/pvfs2 [1]

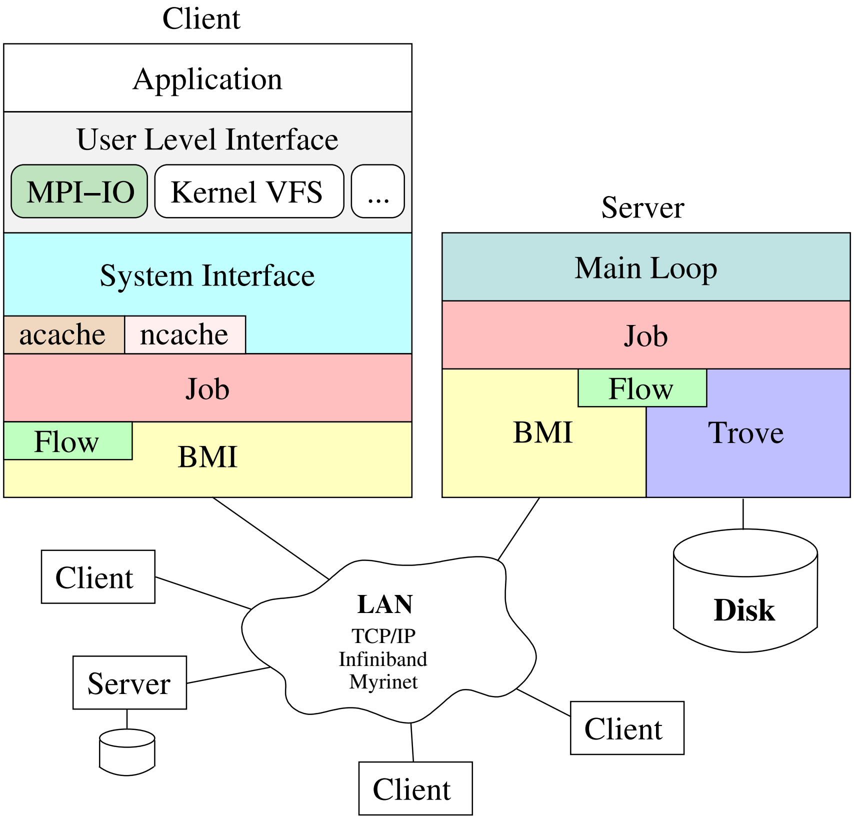


Figure 2.3.: Layer structure design of PVFS2

of the layers and their interfaces please refer to the documentation in the source package[10].

2.3.1. PVFS2 client library

The client side of PVFS2 is a library that is linked to a parallel program, providing methods to do all the usual stuff you wish to do with a filesystem. The library by itself does not use threads, every request is handled after the preceding request completed.

An application can either implement a new *User Level Interface* or use an already existing one such as the PVFS2 ADIO device for ROMIO in MPICH2. The second important *User Level Interface* implementation is the PVFS2 Linux kernel module connecting to the Linux kernel Virtual Filesystem Switch (VFS) and thus providing POSIX conform access to PVFS2. This feature uses an additional daemon on the client side providing the library functions for the kernel module to communicate with the servers. We do not consider this feature in the framework of this thesis.

The layers we can see for the client in figure 2.3 are much more important for the already mentioned daemon version than for the library we use. Job, Flow and BMI are similar to

[10] The documentation is in ${PVFS2-SRC}/doc, for the design first take a look at io-api-terms.

the layers we will see described for the PVFS2 server below. Client-side specific are only the caches in the system interface for minimizing the number of requests to the servers. For a more in detail description see the documentation included in the PVFS2 source code package and also [15, section 2].

2.3.2. PVFS2 server

The PVFS2 server is a daemon, writing the data received from a client to the underlying filesystem or reading and sending requested data to a client. The server needs specially prepared space to use as data storage. It is statically configured so no configuration changes are supported at runtime.

The server daemon runs four threads[11] trying to parallelize request handling:

- The **main thread** started first during server start-up and in the parent of all others. Later it manages the interworking of all parts by controlling the state machines.
- One thread is in charge of the **kernel device interface**, testing for the presence of unexpected messages and handles them.
- One thread is in charge of **Trove**, managing all disk operations, meaning all operations on the underlying filesystem.
- One thread is in charge of the **Buffered Message Interface (BMI)**, handling all network communication.

In the server two important parts are not realized as own threads:

- The **Flow layer** handles data transfers between Trove, BMI and the memory.
- The **Job layer** composes BMI, Flow and Trove operations into one API giving direct access to logical tasks.

2.4. Trace Analysis Tools

In 2005 Julian Kunkel and Dulip Withanage in Thomas Ludwig's group at the Universität Heidelberg developed several tools to prepare SLOG-2 formatted files for a better visualisation in Jumpshot. In particular, these tools are developed to bring MPE created logs from MPICH2+PVFS2 client and PVFS2 servers together and make them visible in a usable form

[11] The server threads are mapped onto pthreads in our environment

for performance analysis. The tools got packed and endued with a make system by the author of this document into one package called *slog2tools*[12].

2.4.1. Merging traces

MergeSlog2 is written by Julian Kunkel. It merges together two slog2 files in a way that the time lines from the second input file appear below the time lines of the first input file. It also provides the option to give a time difference to adjust the time lines of the two files. A simple automatic calculation of time difference is provided too, it just takes the first time in each file and set them as common time zero.

2.4.2. Getting better visual access

The three tools *Slog2ToCompositeSlog2*, *CompositeSlog2ToLineIDMap* and *Slog2Arrows* are developed by Dulip Withanage [16]. They change the inner structure of an slog2 file for better visualisation.

Slog2ToCompositeSlog2 puts together primitive drawables into composites. The user decides on the used criteria.

CompositeSlog2ToLineIDMap scans the input file for overlapping events on the same time line and splits the relevant time lines into sub time lines.

Slog2Arrows is intended only to be a reference to show how to create arrows. It simply creates time line spanning arrows from each drawable to the next in time.

2.4.3. Nicely colored pictures

To get visual access to our SLOG-2 formatted files we use the already described Jumpshot-4 by Anthony Chan at Argonne National Laboratory (see section 2.2.2 “SLOG-2 Software Development Kit” on page 9).

[12] The package will become available at the Internet in the near future.

3. Specification of the Goals

3.1. Starting Point

An MPICH2/PVFS2 system is a complex piece of software. The overall modularity guarantees great flexibility in development and usage, but it also makes it hard to understand what exactly happens with the data from a global perspective.

One first interesting point when we talk about performance analysis and program optimization is what happens at the end (on the physical storage) if we call one MPI-IO function or another in our program. We are already able to produce one single trace with all the MPI-IO calls made in any process of our program using the MPE wrapper library *liblmpe*. We can also get one single trace with events performing disk operations (Trove operations) from the PVFS2 data servers by executing them as MPI program. After the work of Julian Kunkel and Dulip Withanage for the slog2tools we actually can bring these traces together, prepare them and have a look at it in Jumpshot. But the only thing we can see is that some calls of MPI-IO functions led to some disk operations, nothing about what disk operation was initiated by what function call.

3.2. Goal of this Thesis

Our first goal is to become able to recognize connections between MPI-IO function calls and the disk operations they trigger. The disk operations are represented by the Trove events we get into the PVFS2 data server trace.

The second goal is to have an easy way to identify these connections in our visualization with Jumpshot. More precisely we want to see arrows in Jumpshot connecting the events that belongs together in that way. At the end this should look like in figure 3.1 on the next page.

As a third and last point it would be fine to have a tool producing some statistics about the trace from our environment. General statistics would be interesting and especially statistics concerning the connections possible due to the new CallID introduced here, for example the average number of triggered disk operations per MPI-IO function call.

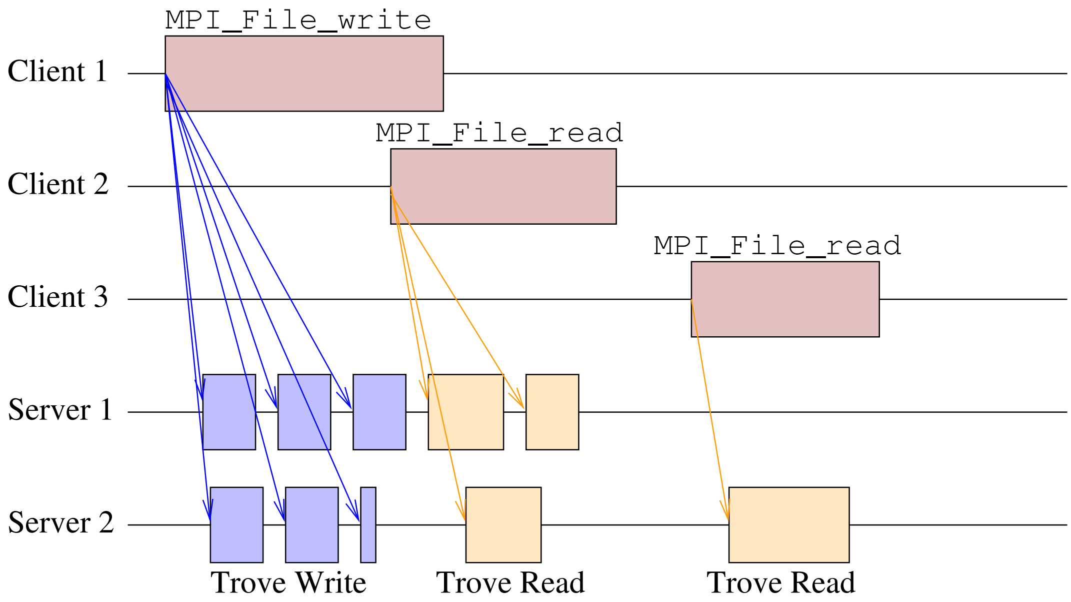


Figure 3.1.: The designated presentation of connections between events in Jumpshot

To reach this goals we first have to extend the tracing capabilities to give us enough information for producing the intended connections. We need to have some kind of unique identification that is only the same at all events causally belonging together.

After that we have to produce the arrows as arrow events in the SLOG-2 formatted trace. The feature to display these arrows at the end is already available in Jumpshot-4.

3.3. Approach

The main idea is to create an ID with each function call in MPI-IO and put it into the trace. Because it identifies the function calls in MPI-IO we will name it *CallID*. After creating it we will push this CallID through the whole system to log it together with all Trove events resulting of the MPI-IO function call it got created for.

Some relevant points that have to be discussed are

- There are many different MPI-IO functions (see appendix A "List of MPI-IO Functions" on page 77) defined in the MPI-2 standard [3] which are intended to do I/O in very different ways. To start we can care about only a few of them but in the future, research should be done to cover all these functions.

- We have to reflect on how many single disk operations can be triggered by on call of an MPI-IO function. The reverse case could also be possible, collective calls are designed to combine more than one request from MPI processes to one request to the file system.

- A very implementation specific decision to make is how it is possible to get the CallID from MPI-IO function call to the Trove layer of the PVFS2 data servers.

All the here mentioned points we will discuss in the next chapter about the design.

4. Design

This chapter is about the general design of the enhancements getting realized within the scope of this thesis. It describes the concept of what we will do.

4.1. General Issues

Different MPI-IO functions

Let us first discuss the differences between several MPI-IO functions. A complete list of all MPI-IO functions is given in appendix A "List of MPI-IO Functions" on page 77. We are especially interested in differences in the number of produces disk operation requests. An aspect changing the number of operations triggered by one function call may be the size of data to transfer. But as this should be the same with all functions we cannot use it as a criterium. Supposably the second great difference is between collective and non-collective versions of functions. Read and write should make little difference and also the kind of positioning (pointer or explicit offset) should not altering the request scheme so much.

Collective functions in MPI-IO are functions that have to be called by several MPI processes at almost the same time to do a collective I/O operation. With this kind of function we can run into big trouble with our approach using a CallID. In theory many calls could produce one request to the server and in that case one Trove event would have more than one CallID. Therefore we will first concentrate on the non-collective calls to get a feeling of the attempt.

A little deeper discussion of the collective call problem we will do in section 4.4 "Limitations of Design" on page 23.

Ratio of MPI-IO function calls to Trove events

Since we decided to begin with non-collective calls we will never come into the situation to produce one Trove event with more than one MPI-IO function call. But the reverse case is possible. Depending on the size of the data and the distribution function (see [15]) the data we want to read or write using one call of an MPI-IO function are spread over several PVFS2 data servers.

The *distribution function* is the algorithm used by PVFS2 to distribute the data over the available PVFS2 data servers. The concept of the standard distribution is striping with a fix stripe size. That means for a stripe size of X bytes, that the bytes 0 to $X-1$ of the virtual filesystem are handled by the first data server, the bytes X to $2X-1$ are handled by the second data server and so on. Assuming we have n data servers, the bytes $(n-1)X$ to $nX-1$ are handled by the last data server and the bytes nX to $(n+1)X-1$ are handled again by the first data server[1].

There is also another reason for one MPI-IO function call to become split into several requests. The size of PVFS2 messages is limited to a size of 64KB so larger requests have to be split over several messages to get the data from the client to one server and vice versa. This occurs even if there is only one PVFS2 data server.

Summarizing we can say that if using only non-collective functions each Trove event will be triggered by exactly one MPI-IO function call. But in the other direction one MPI-IO call can and mostly will trigger several Trove events.

4.2. Getting the Necessary Information

Before we are able to show connections between MPI-IO function calls and resulting disk operations we need to have the right information in the right places.

4.2.1. Logging of MPI-IO function calls

By linking our MPI program against *liblmpe* (see section 2.2 "MPE2" on page 8) we can get a trace with all MPI-IO calls invoked during the program run. MPE takes care for logging all invocations in each MPI process and puts this information together in one single trace.

4.2.2. Logging of PVFS2 trove events

In PVFS2 there is a separate part responsible for logging called *event manager*. Each part of PVFS2 tells the event manager when an event occurred. The event manager already comes with a backend for MPE2, so we can use the MPE capabilities to get a trace out of the system. In order to get one single trace file from all servers, we need to run the PVFS2 server daemon using MPI. This is possible with the method and patches developed by Julian Kunkel (see section 2.3.2 "PVFS2 Server" on page 12). We are particularly interested in all Trove events because they indicate real disk operations. The event manager innately logs trove read and write separately.

[1] In general byte number a is handled by Server $\lfloor a/X \rfloor \bmod n$

4.2.3. Making connections possible

Now we do have one combined trace from all clients with all MPI-IO calls and one combined trace from all servers with all Trove events. We can merge these two traces using MergeSlog2 written by Julian Kunkel and included in slog2tools package (see section 2.4.1 "Merging Traces" on page 13). Subsequently we have one single trace with all the events of interest.

To make a logical connecting of the MPI-IO calls with the triggered Trove events possible we need to have information about the triggering event in the triggered one. The MPI-IO call is the triggering event and always happens first. Consequently we generate a CallID for every MPI-IO function call and pass it through to whole system to Trove, finally doing the disk operation. Trove then gives the ID together with the event notification to the event manager. The same CallID then will get logged both with the MPI-IO function call and with all triggered Trove events. As we can see, the issue that one MPI-IO call can trigger several Trove events should make no problem, we just have to duplicate the CallID at the point of the system where the MPI-IO request gets also split.

Figure 4.1 on the following page anticipate some implementation issues for consistency reasons but we can already see some interesting things concerning the design in there:

I. The CallID becomes generated in the MPI-IO wrappers provided by MPE. MPE also puts the CallID together with the MPI-IO event into the client trace. *The CallID becomes stored globally in the client address space.*

II. The PVFS2 client takes the CallID and puts it together for each request into the server request protocol. *Since the whole client is not threaded the CallID cannot be overwritten until the client has completed the request. Since the PVFS2 client is in the same address space as MPE it can access the globally stored CallID directly.*

III. The CallID becomes transferred with each request to all PVFS2 servers involved.

IV. The CallID becomes transferred through the PVFS2 server passing the layers. The request comes over the network and is received by the networking layer BMI. Job becomes aware of a new message takes it and starts an appropriate Flow. The Flow then transfers the data from BMI to Trove (in case of a write request) or requests the data from Trove and waits to transfer it to BMI (in case of a read request). Trove reports the operation to the event manager. *On each crossed layer boundary a reference is assigned making it possible for the event manager to access the memory address where the right request protocol containing the CallID is stored.*

V. Trove handles the request and reads data from the disk respectively writes data to the disk.

VI. Finally the event manager puts the CallID together with the Trove event into the server trace.

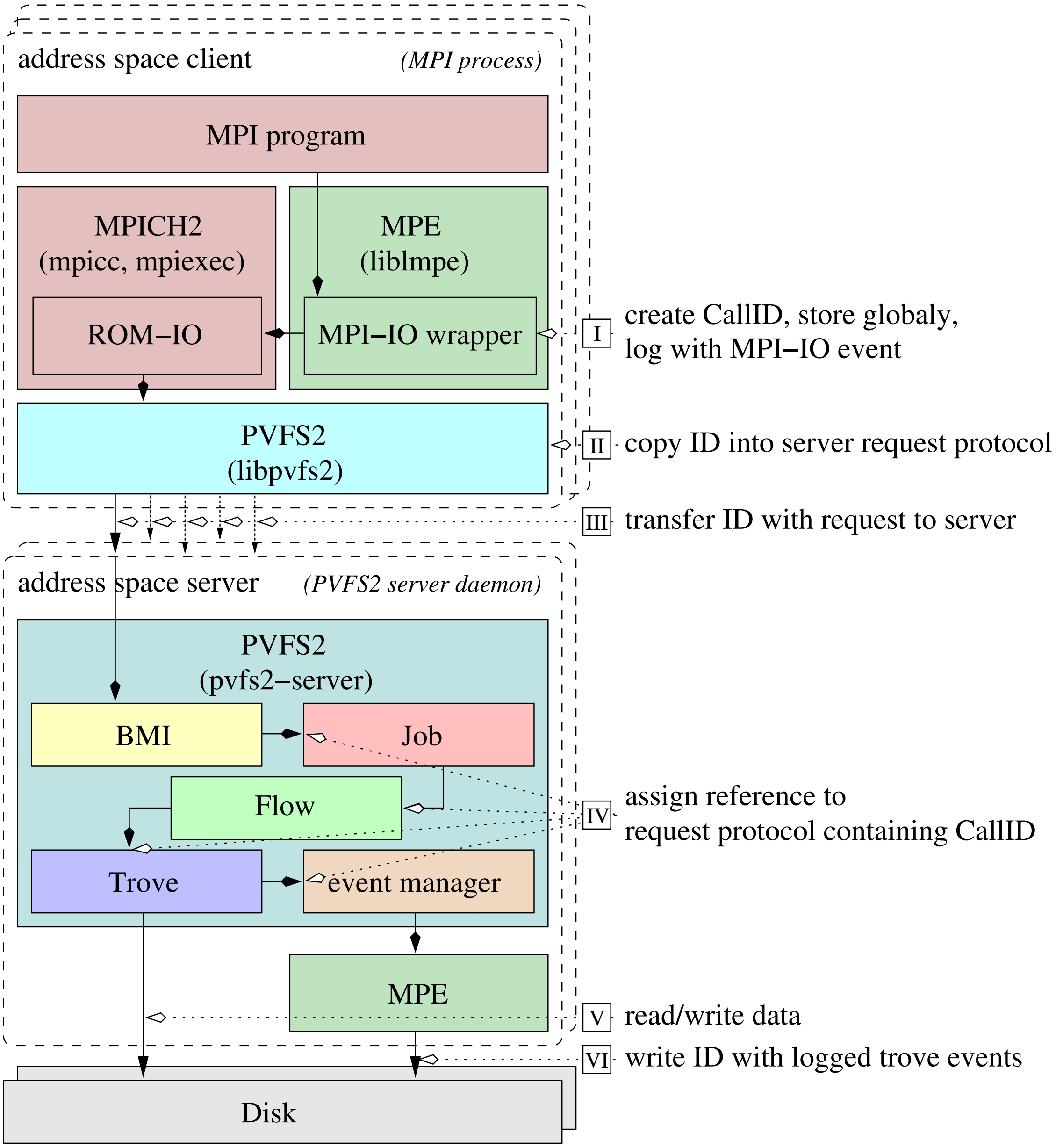


Figure 4.1.: The way every MPI-IO call takes through the system

4.3. Making Information Visible

After having traces with CallIDs, we need to find ways how to make expedient use of it.

4.3.1. Text based analysis

With the SLOG-2 SDK there comes a tool called *slog2print*. With this tool we can get all information in an SLOG2 trace printed to the console. This enables us to write a Perl script compiling some statistics about our SLOG2 trace. We will use Perl because it is an excellent choice for scripts mainly doing text parsing and processing.

Since the exact statistical information a user wants to have can highly vary, it would be difficult and intricate to write one tool for everyone. So we will confine ourself to just write a reference implementation that can grow with the requirements.

Some requirements to resolve already in the reference implementation are:

- The reference script should read in and provide all category information. Since the categories are extremely important in SLOG2 format this information will be needed and used very often.

- It should be structured in a manner that extensions can be applied easily.

- There should be a few simple statistical functions implemented: a counting of the categories with CallID, a calculation of the average number of trove operations one MPI-IO call produces.

The *slog2print* output is structured into five sections:

```
         SLOG-2 Header:
[...]

         CategoryMap:
[...]

         SLOG-2 Tree Directory
[...]

         LineIDMapList:
[...]



1: Primitive[ infobox[ TimeBBox(3.9186038970947266,3.9186408519744873) Category=
Category[ index=301, name=Job, topo=State, color=(255,0,0,255,true), isUsed=true
, width=1, vis=true, search=true, ratios=0.0890226,-0.3662462, count=76 ]: ] (3.
918604, 2) (3.9186409, 2) ] bsize=32
2: Primitive[ ...
[...]
```

Listing 4.1: Output sections of *slog2print*

The script should read its input line by line because most information parts are given in one single line. Each section contains other kinds of information and will need different parsing. So it would be fine for the script to know from what section the current line is. It then can restrict the parsing of the line to those rules valid in the current context. Each section border has a headline to identify it but the last that is introduced with three blank lines. We will use these border marks to let the script know in what context it is at every moment.

4.3.2. Graphical analysis

Also for graphical analysis there is already a Tool in the SLOG-2 SDK called Jumpshot-4 (see section 2.2.2 "SLOG-2 Software Development Kit" on page 9). As already mentioned Jumpshot can draw the in SLOG-2 format [10] defined arrows. So what we still need to do is adding arrow primitives with the appropriate end points into our trace to represent the connection of MPI-IO function calls with the respective triggered trove events. For adding arrows to slog2 traces Dulip Withanage has written a reference script Slog2ToArrowSlog2 in the course of his bachelor thesis [16]. Unfortunately this is just a very rudimentary implementation, so the big part of the work is still to be done, but there is a skeleton structure we can use as our basis.

The algorithmic principle we should use is:

```
FOR <all drawables in increasing start time order>
BEGIN
  IF <current drawable has CallID and this CallID is never seen before>
  BEGIN
    FOR <all drawables after current>
    BEGIN
      IF <current inner drawable has the same CallID as current outer drawable>
      BEGIN
        <create arrow from current outer to current inner drawable>
      END
    END
  END
END
```

Listing 4.2: Algorithm for creating arrows in *Slog2ToArrowSlog2*

The idea is that always the first occurrence in time must be the initiator of an operation and with it the MPI-IO call that created the CallID will be found as first drawable. So all arrows for one CallID will begin at the MPI-IO call and end at one Trove operation. Using this method we get a worst case runtime complexity of $O(\frac{n^2-n}{2}) \leq O(n^2)$.

To have a additional visual attribute already to see at the start point of an arrow what kind of trove operation it points to, the arrow should have the same color as the trove operation. That means orange for "Trove Read" and blue for "Trove Write". Arrows with unknown target type should have the color white. To have an indicator what arrow it is without having its begin or end in the viewport the CallID should become set for the arrow too.

4.4. Limitations of Design

Of course there are limitations that we cannot break with the described concept. Let us shortly discuss some critical aspects here:

Collective calls

MPI-IO provides so called collective calls. With them it is possible for multiple processes to access data in one file in a coordinated way. The idea behind this is to put the MPI implementation in a position for doing some optimization.

What happens with our CallID in that case? We already began this discussion in section 4.1 "General Issues" on page 17 at the beginning of this chapter. With collective calls it could happen that one Trove event (and so disk operation) is triggered by multiple MPI-IO function calls, or in other words by one collective call done in multiple MPI processes. The problem is that the Trove event could so get multiple CallIDs, one per process participating. The question in that case is which one the PVFS2 client takes to pass with the requests to the server. The answer to this question depends completely from the algorithm the PVFS2 backend of ROMIO uses. We will not analyse this here in detail. On the first view it seems not to be reasonable for PVFS2 to handle collective calls different from non-collective because the filesystem and so presumably the accessed data is spread over several PVFS2 data servers anyway. So perhaps the ADIO device for PVFS2 just map collective to non-collective calls and let PVFS2 do the complete optimization. This is a yet open issue in this context and will not be covered in this thesis (see also section 8.2.1 "Take Care of Collective Calls" on page 73 in chapter "Future Work").

5. Implementation

This chapter describes in detail the implementation of the concepts discussed in the last chapter about the design. There are many code fragments given so browsing the source code while reading this document should not be necessary. Nevertheless, it can be helpful sometimes, since the complex and geared structure especially of the PVFS2 server code makes it often impossible to show every detail.

5.1. General Issues

What we want to do here is to modify an existing software system that is itself under constant development. This circumstance leads to some potential problems, that we have to take care about.

Avoid changing the standard behaviour of the Software Even with our new patches included, the whole system should behave exactly as without. Moreover the user should have the choice to deactivate our modifications in the case some problems arise for him. To achieve that, we have several alternatives:

- We could try to produce an extra library. To get our functionality, someone would have to bind it to his program, together with MPE2 and PVFS2 libraries.
  This will *not* be practicable since our changes will not give dedicated extra functionality but enhance some functions already anchored in the system.

- We could use preprocessor macros to include or exclude the changes we will make using compiler flags.
  This seems to be a *good solution*. We can enclose all code we modify with

  ```
  #ifdef PVSHD_CALLID
  [our new or changed code]
  #else
  [the original code]
  #endif // PVSHD_CALLID
  ```

> *PVSHD* stands for “Parallele und Verteilte Systeme, Heidelberg” which is the German name of our research group “Parallel and Distributed Systems” at the Universität Heidelberg, Germany.

Avoid producing name conflicts with existing or future changes Not to come into trouble with that issue, we will use the prefix ’`pvshd_`’ for all global variable names. To use the same proceeding for local variables too, would make the code more illegible than being helpfully. We will not do that.

Hold modifications as minimal as possible As already mentioned the base systems PVFS2, MPICH2 and MPE2 themselves are under permanent development. So unfortunately the risk is relatively high, that patches will not work without producing conflicts after short time. To keep that risk as small as possible, we should hold our modifications as small as possible.
In particular that means:

- Avoid modification of parameter lists.
- Avoid copying many lines of code.

Even though these principles will not guarantee patches to apply without conflicts in the future, they will reduce the risk dramatically and make it even simpler to adjust the patches.

Now we start making a step by step modification of the system. We will begin in section 5.2 “Creation of CallID for MPI-IO Function Calls” on the following page with the CallID generation and logging in the wrapper functions for MPI-IO calls in *liblmpe*. After this, section 5.3 “Getting the ID to the Right Place” on page 33 will guide us through all modifications necessary to pass the generated CallID through the System to the event manager in the PVFS2 servers where it is put into the trace. Section 5.3 is cut into three parts, each for one section of the CallID’s path: “Client Side: From MPE2 to BMI”, “From Client Side to Server Side” and “Server Side: From BMI to Event Manager”. This completes the modifications in this part of the system so in section 5.4 “Statistic and Visualisation Tools” on page 47 we will develop two trace analysis tools, one text-based to produce statistics about the trace and one to add arrows to the slog2 file presenting the connections between MPI-IO function calls and Trove operations. At the end we will shortly discuss the limitations caused by our implementation in section 5.5 “Limitations of Implementation” on page 57, address some problems in section 5.6 “Implementation Problems” on page 58 and give a final summary in section 5.7 “Summary” on page 60.

5.2. Creation of CallID for MPI-IO Function Calls

The first decision to make is where exactly to implement the creation of our CallID. The only purpose for our ID is tracing with MPE2. So this should be the best place to start. General logging of MPI functions is done with *liblmpe* that consists mainly of wrappers around MPI functions using the MPI profiling library. We are especially interested in the part for MPI-IO functions (`MPI_File_*`). The related code is in log_mpi_core.c[1] and log_mpi_io.c[2].

In the MPE2 logging concept for start-stop events (in opposition to single events) the first step is to declare the logging states (see section 2.2 on page 8 about MPE2). For logging the MPI functions *liblmpe* innately uses logging states without the optional info statement. Indeed the internal declaration function `MPE_Describe_known_state` has a parameter prepared for passing the format of an info field, but currently set to `NULL`. To log information with the events, we change the function call for all logging states. Later we are going to define `NULL` as initial value for `format`, so for all states we do not set an info format nothing will change.

The declaration of the logging states for MPI functions is done in the wrapper `MPI_Finalize` in log_mpi_core.c:

```
int  MPI_Finalize( )
{
    MPE_State       *state;
[...]
    if (procid_0 == 0) {
        fprintf( stderr, "Writing␣logfile....\n" );
        for ( idx = 0; idx < MPE_MAX_KNOWN_STATES; idx++ ) {
            if (totcnt[idx] > 0) {
                state  = &states[idx];
                MPE_Describe_known_state( CLOG_CommIDs4World, 0,
                                          state->stateID,
                                          state->start_evtID,
                                          state->final_evtID,
                                          state->name, state->color,
#ifdef PVSHD_CALLID
                                          state->format);
#else
                                          NULL );

#endif
            }
        }
    }
[...]
}
```

Listing 5.1: Modified `MPI_Finalize` in log_mpi_core.c

[1] ${MPE2-SRC}/src/wrappers/src/log_mpi_core.c
[2] ${MPE2-SRC}/src/wrappers/src/log_mpi_io.c

For this to become working, we need to add the field `format` to the structure `MPE_State` which is the type of `state`. In addition we adjust the initialization of `state` in the wrapper `MPI_Init`. As already indicated we make `NULL` the default value of `format`.

All these code is in log_mpi_core.c:

```
typedef struct {
    int  stateID;      /* CLOG state ID */
    int  start_evtID;  /* CLOG Event ID for the beginning event */
    int  final_evtID;  /* CLOG event ID for the ending event */
    int  n_calls;      /* Number of times this state used */
    int  is_active;    /* Allows each state to be selectively switched off */
    int  kind_mask;    /* Indicates kind of state (message, environment) */
    char *name;        /* Pointer to name */
    char *color;       /* Color (or B&W representation) */
#ifdef PVSHD_CALLID
    char *format;      /* Info format */
#endif
} MPE_State;
```

Listing 5.2: Modified *struct* `MPE_State` in log_mpi_core.c

```
/*
 * Replacement for MPI_Init.  Initializes logging and sets up basic
 * state definitions, including default color/pattern values
 */
int  MPI_Init( argc, argv )
int * argc;
char *** argv;
{
  MPE_State  *state;
[...]
  /* Initialize all states */
  for ( idx = 0; idx < MPE_MAX_KNOWN_STATES; idx++ ) {
      state               = &states[idx];
      state->stateID      = MPE_Log_get_known_stateID();
      state->start_evtID  = MPE_Log_get_known_eventID();
      state->final_evtID  = MPE_Log_get_known_eventID();
      state->n_calls      = 0;
      state->is_active    = 0;
      state->name         = NULL;
      state->kind_mask    = 0;
      state->color        = "white";
#ifdef PVSHD_CALLID
      state->format       = NULL;
#endif
  }
[...]
}
```

Listing 5.3: Modified `MPI_Init` wrapper in log_mpi_core.c

The next step is to set a format of the info field for the MPI-IO functions. The filling of the `state` structures for MPI-IO logging is done in function `MPE_Init_MPIIO` defined in log_mpi_io.c.

As an example we take the one for `MPI_File_write`:

```
void MPE_Init_MPIIO( void )
{
  MPE_State *state;
[...]
  state = &states[MPE_FILE_WRITE_ID];
  state->kind_mask = MPE_KIND_FILE;
  state->name = "MPI_File_write";
  state->color = "brown:gray2";
#ifdef PVSHD_CALLID
  state->format = "CallID=0x%x";
#endif
[...]
}
```

Listing 5.4: Modified `MPE_Init_MPIIO` in log_mpi_io.c

The format tag '`%x`' means to print the CallID in hexadecimal notation. Unfortunately the printf style tag for leading zeros is not supported by MPE.

As we now have the format, it is time to fill something into the field. To get a unique ID for each function call we will use a global counter. But as MPI is for parallel use and so there is usually more than one process with independent address spaces, we also should include something unique for each process. MPI provides exactly what we need, the process id. In MPE it is stored globally in `procid_0`.

We implement our counter as an *`unsigned int`* to have a larger range of positive values. It can be *`static`* because we will only use it in log_mpi_io.c and we do not want it to be changed from somewhere else.

```
#ifdef PVSHD_CALLID
/* counter for function calls to create unique ids */          2
static unsigned int pvshd_callcounter = 0;
//TODO: find better place for that                              4
#include "mpe_misc.h"
#endif                                                          6
```

Listing 5.5: Modified head of log_mpi_io.c

Note the *`#include`* `"mpe_misc.h"` in line 5 for now, we will need that later in section 5.3.1 "Client Side: From MPE2 to BMI" on page 33.

Now the wrapper for each function has to be modified to create and log the CallID. As each wrapper has a similar structure we use again `MPI_File_write` as example:

This is the original wrapper:

```
int MPI_File_write( MPI_File fh,
                    void * buf,                                                  2
                    int  count,
                    MPI_Datatype datatype,                                       4
                    MPI_Status * status )
{                                                                                6
  int returnVal;
                                                                                 8
/*
    MPI_File_write - prototyping replacement for MPI_File_write                  10
    Log the beginning and ending of the time spent in MPI_File_write calls.Col80
*/                                                                               12
  MPE_LOG_STATE_DECL
                                                                                 14
  MPE_LOG_STATE_BEGIN(MPE_COMM_NULL, MPE_FILE_WRITE_ID)
                                                                                 16
  returnVal = PMPI_File_write( fh, buf, count, datatype, status );
                                                                                 18
  MPE_LOG_STATE_END(MPE_COMM_NULL)
                                                                                 20
  return returnVal;
}                                                                                22
```

Listing 5.6: Original `MPI_File_write` wrapper in log_mpi_io.c

The MPE way of getting the info value into the traces is to first pack it into a buffer in the format defined while state declaration and then assign a pointer to that buffer to the logging function.

Let us take care about the second issue first: As we can see in lines 15 and 19 there are macros used to expand the real logging function calls at preprocessing time.

These macros are defined in log_mpi_core.c:

```
#define MPE_LOG_STATE_BEGIN(comm,name) \
    if (trace_on) { \
        state = &states[name]; \
        if (state->is_active) { \
            commIDs = CLOG_CommSet_get_IDs( CLOG_CommSet, comm ); \
            MPE_Log_commIDs_event( commIDs, 0, state->start_evtID, NULL ); \
        } \
    }
#define MPE_LOG_STATE_END(comm) \
    if (trace_on && state->is_active) { \
        MPE_Log_commIDs_event( commIDs, 0, state->final_evtID, NULL ); \
        state->n_calls += 2; \
    }
```

Listing 5.7: Original macros `MPE_LOG_STATE_*` in log_mpi_core.c

These macros are also used in wrappers for non MPI-IO functions, so changing them would require many other changes, too. It is better to define two new macros for our purposes. Later on, to integrate the whole things deeper into the code, one more parameter in the common macros would be a much better solution, in particular since the last argument of `MPE_Log_commIDs_event` currently set to `NULL` is exactly for assigning an info value.

For now we define two new macros:

```
/*
   These macros are to use in log_mpi_io.c instead of the ones above for
   assigning the info value holding the CallID.
   TODO: Think about changing the originals to pass the info buffer through
*/
#define MPE_PVSHD_LOG_STATE_BEGIN(comm,name,bytebuf) \
    if (trace_on) { \
        state = &states[name]; \
        if (state->is_active) { \
            commIDs = CLOG_CommSet_get_IDs( CLOG_CommSet, comm ); \
            MPE_Log_commIDs_event( commIDs, 0, state->start_evtID, bytebuf ); \
        } \
    }
#define MPE_PVSHD_LOG_STATE_END(comm,bytebuf) \
    if (trace_on && state->is_active) { \
        MPE_Log_commIDs_event( commIDs, 0, state->final_evtID, bytebuf ); \
        state->n_calls += 2; \
    }
```

Listing 5.8: New macros `MPE_PVSHD_LOG_STATE_*` in log_mpi_core.c

We still have to create a buffer, pack the CallID into it and assign it using our new macros.

The new wrapper should then look like that[3]:

```
int MPI_File_write( MPI_File fh,
                    void * buf,
        int count,
        MPI_Datatype datatype,
        MPI_Status * status )
{
  int returnVal;

/*
    MPI_File_write - prototyping replacement for MPI_File_write
    Log the beginning and ending of the time spent in MPI_File_write calls.
*/
  MPE_LOG_BYTES  bytebuf;      // buffer for logging of CallID
  int            bytebuf_pos;  // position pointer for buffer
```

[3] For readability, in this code fragment all lines are already removed that the preprocessor would remove due to PVSHD_CALLID compiler flag set.

```
  // set CallID as described in src/misc/include/mpe_misc.h                    16
  pvshd_callid = (procid_0 << PVSHD_COUNTBITS) |
                   (++callcounter << PVSHD_PROCBITS >> PVSHD_PROCBITS);          18

  MPE_LOG_STATE_DECL                                                              20

  // pack callid into buffer                                                      22
  bytebuf_pos = 0;
  MPE_Log_pack( bytebuf, &bytebuf_pos, 'x', 1, &pvshd_callid );                  24

  // log start of operation including callid into trace                           26
  MPE_PVSHD_LOG_STATE_BEGIN(MPE_COMM_NULL, MPE_FILE_WRITE_ID, &bytebuf)
                                                                                  28
  // call true function in MPI profiling library
  returnVal = PMPI_File_write( fh, buf, count, datatype, status );               30

  // log end of operation including callid into trace                             32
  MPE_PVSHD_LOG_STATE_END(MPE_COMM_NULL, &bytebuf)
                                                                                  34
  return returnVal;
}                                                                                 36
```

Listing 5.9: Modified `MPI_File_write` in log_mpi_io.c

The setting of `pvshd_callid` (line 17 et seq.) looks somewhat strange. What the expression does is just to create one single integer structured like shown in figure 5.1. Even `pvshd_callid` is not declared now, we will care about that later in section 5.3.1 “Client Side: From MPE2 to BMI” on page 33.

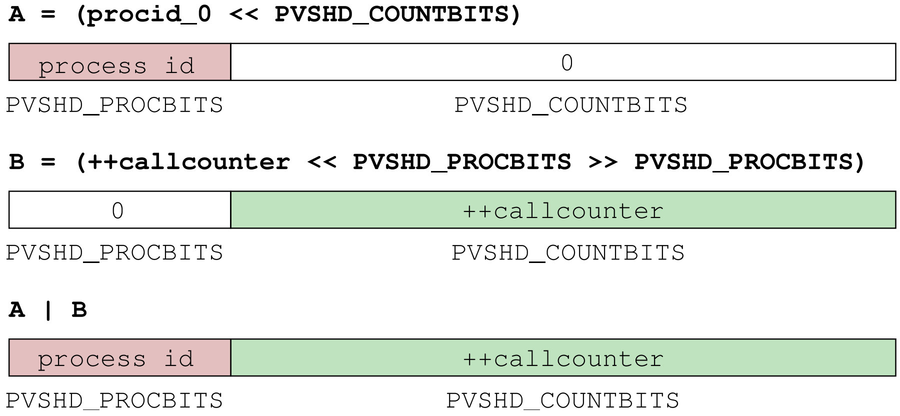


Figure 5.1.: Structure of integer containing CallID

To avoid the necessity of modifying each single wrapper and state declaration by hand we now write a short sed script[4] to do that all at once (`change_log_mpi_io.sed`, listing 5.10 on the following page). In addition this enables us to react on changes faster and less work intensive.

[4] For those not familiar with sed script syntax please refer to sed manual page [17] or accept that this script just does the job.

The script is to be called as described at its beginning:

```
#!/bin/sed -f

# use: $1 origfile > changedfile
# or: $1 -i origfile

/^void MPE_Init_MPIIO( void )$/i\
\#ifdef PVSHD_CALLID\
\/* counter for function calls to create unique ids *\/\
static unsigned int pvshd_callcounter = 0;\
\/\/TODO: find better place for that\
\#include "mpe_misc.h"\
\#endif\
\

/^  state->color = ".*";$/a\
\#ifdef PVSHD_CALLID\
  state->format = "CallID=0x%x";\
\#endif

/^  MPE_LOG_STATE_DECL$/i\
\#ifdef PVSHD_CALLID\
  MPE_LOG_BYTES  bytebuf;      // buffer for logging of callid\
  int            bytebuf_pos;\
  \
  \/\/ set callid as described in src/misc/include/mpe_misc.h\
  pvshd_callid = (procid_0 << PVSHD_COUNTBITS) |\
                    (++pvshd_callcounter << PVSHD_PROCBITS >> PVSHD_PROCBITS);\
\#endif

/^  MPE_LOG_STATE_BEGIN(MPE_COMM_NULL,/{
i\
\#ifdef PVSHD_CALLID\
  \/\/ pack callid into buffer\
  bytebuf_pos = 0;\
  MPE_Log_pack( bytebuf, &bytebuf_pos, 'x', 1, &pvshd_callid );\
  \
  \/\/ log start of operation including callid into trace
h
s/MPE_LOG_STATE_BEGIN/MPE_PVSHD_LOG_STATE_BEGIN/
s/)/, \&bytebuf)/
p
g
i\
\#else
a\
\#endif
}

/^  returnVal = PMPI_File_/i\
  // call true function in MPI profiling library
```

```
/^  MPE_LOG_STATE_END(MPE_COMM_NULL)$/{
i\
\#ifdef PVSHD_CALLID\
  \/\/ log end of operation including callid into trace\
  MPE_PVSHD_LOG_STATE_END(MPE_COMM_NULL, &bytebuf)\
\#else
a\
\#endif
}
```

Listing 5.10: sed script to automatically modify original log_mpi_io.c

5.3. Getting the ID to the Right Place

We have created one unique CallID for every MPI-IO function call and taken care of getting it into the trace. Now we need to bring that very same CallID to the PVFS2 server trove layer and event manager so we can see what disk operations are triggered by each call.

5.3.1. Client side: From MPE2 to BMI

As we can see in figure 5.2 on the next page the PVFS2 server request in *libpvfs2* is set up in sys-io.sm[5], so we need the CallID to be transferred to this place.

One clear way to do that is to pass it from function to function all the way shown in figure 5.2 on the following page. Unfortunately this would require many changes of parameter lists and could become very complicated in certain cases (see section 5.1 “General Issues” on page 24).

Since our MPI program with all its linked libraries (*liblmpe*, *libpvfs2*) is not multithreaded and moreover has one single address space, we have another alternative: putting the ID into a global variable. For now it should be sufficient just to find an adequate place to put this variable in, later a better implementation should be developed (also see section 8.3.2 “Better Inclusion of the Changes” on page 76).

Such a place for now is mpe_io.c[6] and mpe_misc.h[7]:

- The CallID is to be set in log_mpe_io.c. mpe_misc.h itself is very small. It can be included into log_mpe_io.c without producing any conflicts.

  As you remember we already did that in the last section (listing 5.5 on page 28).

[5] ${PVFS2-SRC}/src/client/sysint/sys-io.sm
*.sm files contains mostly native C code and are used to generate the code for state machines

[6] ${MPE2-SRC}/src/misc/src/mpe_io.c

[7] ${MPE2-SRC}/src/misc/include/mpe_misc.h

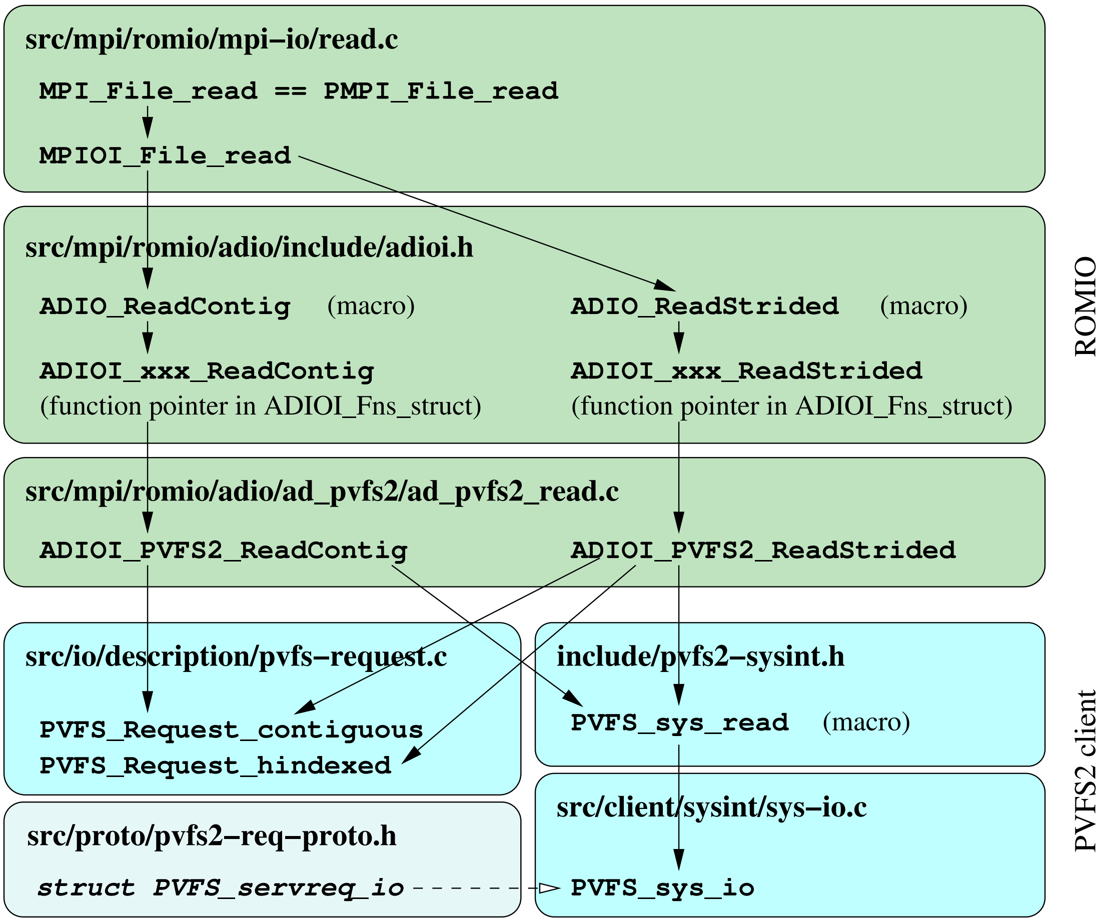

src/mpi/romio/mpi–io/read.c
MPI_File_read == PMPI_File_read
MPIOI_File_read
src/mpi/romio/adio/include/adioi.h
ADIO_ReadContig (macro)
ADIO_ReadStrided (macro)
ADIOI_xxx_ReadContig
(function pointer in ADIOI_Fns_struct)
ADIOI_xxx_ReadStrided
(function pointer in ADIOI_Fns_struct)
ROMIO
src/mpi/romio/adio/ad_pvfs2/ad_pvfs2_read.c
ADIOI_PVFS2_ReadContig
ADIOI_PVFS2_ReadStrided
src/io/description/pvfs–request.c
PVFS_Request_contiguous
PVFS_Request_hindexed
include/pvfs2–sysint.h
PVFS_sys_read (macro)
src/proto/pvfs2–req–proto.h
struct PVFS_servreq_io
src/client/sysint/sys–io.c
PVFS_sys_io
PVFS2 client

Figure 5.2.: The way of function calls from liblmpe to libpvfs2

- The situation in *libpvfs2* is a bit more complicated. Since PVFS2 is to be compiled alone after MPE2 we must have our variable in a header to be included in the *libpvfs2* code. With mpe_misc.h this is possible as it becomes a part of mpe.h (the header for *libmpe*, see below). At the moment, including mpe.h in the appropriate places in *libpvfs2*'s code makes no problems, but this is a point to take care of in the future. Furthermore this makes *libpvfs2* depend on *libmpe* after our changes. That is also no problem for us since without MPI and MPE the changes do not make sense at all.

Both of these files, mpe_misc.h and log_mpe_io.c are part of *libmpe*[8].

[8] We will link our programs against *liblmpe* and *libmpe*.

Not to have a double declared variable `pvshd_callid`, we mark it as *`extern`* in mpe_misc.h. So *libpvfs2*'s sys-io.sm and *liblmpe*'s log_mpi_io.c will know about it but no extra memory will be reserved:

```
[...]
#ifndef _MPE_MISC                                                              2
#define _MPE_MISC
                                                                               4
#ifdef PVSHD_CALLID
#include <stdint.h>                                                            6
/* choose a multiple of 4 here to get process id separated in hex notation */
#define PVSHD_PROCBITS 8 /* max #processes is 256 */                           8
#define PVSHD_COUNTBITS ( sizeof(uint32_t) * 8 - PVSHD_PROCBITS )
extern uint32_t pvshd_callid;                                                  10
#endif
[...]                                                                          12
#endif
```

Listing 5.11: Modified mpe_misc.h

The *`#include`* `<stdint.h>` in line 6 is necessary for using `uint32_t` type. The macros `PVSHD_PROCBITS` (line 8) and `PVSHD_PROCBITS` (line 9) we have already seen in figure 5.1 on page 31 and used in the wrapper modifications (listings 5.6 to 5.10 on pages 29–32). Their purpose is to set the number of bits from the whole integer to use for the process and the counter. Since the sum is definite, setting one is sufficient.

In our case we always set the process bits and calculate the counter's proportion. In general the numbers to take are not limited within the space of bits used for `uint32_t` ($[0, 32]$), but taking a multiple of four has one great advantage because we will have the CallID in hexadecimal notation in the trace: With a split on a multiple of four in binary notation there is also a split between two digits in hexadecimal, since always four digits are combined. (see figure 5.3)

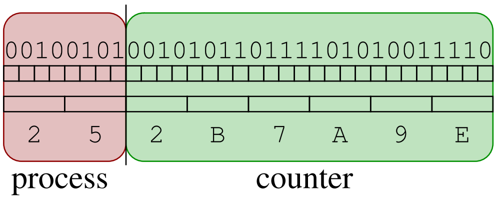


Figure 5.3.: CallID in binary and hexadecimal notation

We still need to declare the real variable `pvshd_callid` in mpe_io.c:

```
#include "mpe_misc_conf.h"
#include "mpe_misc.h"
[...]
#ifdef PVSHD_CALLID
//TODO: Find a better place for that
uint32_t pvshd_callid;
#endif
[...]
```

Listing 5.12: Declaration of `pvshd_callid` in mpe_io.c

and to make our variable known in the PVFS2 client state machine (sys-io.sm):

```
[...]
#ifdef PVSHD_CALLID
#include "mpe.h"
#endif
[...]
```

Listing 5.13: Modified head of sys-io.sm

5.3.2. From client side to server side

Now that we have the CallID at the place where the server requests are created, we should modify the *I/O server request protocol* defined in pvfs2-req-proto.h[9]. The protocol as a whole consists of a structure holding the data and some functions for filling and encoding.

First we extend *struct* PVFS_servreq_io:

```
struct PVFS_servreq_io
{
    PVFS_handle handle;        /* target datafile */
    PVFS_fs_id fs_id;          /* file system */
    /* type of I/O operation to perform */
    enum PVFS_io_type io_type; /* enum defined in pvfs2-types.h */

    /* type of flow protocol to use for I/O transfer */
    enum PVFS_flowproto_type flow_type;

    /* relative number of this I/O server in distribution */
    uint32_t server_nr;
    /* total number of I/O servers involved in distribution */
    uint32_t server_ct;

    /* distribution */
    PINT_dist *io_dist;
    /* file datatype */
    struct PINT_Request * file_req;
    /* offset into file datatype */
    PVFS_offset file_req_offset;
    /* aggregate size of data to transfer */
    PVFS_size aggregate_size;
#ifdef PVSHD_CALLID
    /* mpiio function call id created by function wrapper in liblmpe */
    uint32_t pvshd_callid;
#endif
};
```

Listing 5.14: Modified *struct* PVFS_servreq_io in pvfs2-req-proto.h

[9] ${PVFS2-SRC}/src/proto/pvfs2-req-proto.h

The PVFS2 requests have to become encoded to deal with the requirements of heterogeneous environments. Of cause they also have to become decoded after transmission, but as this is exactly the same flow as encoding except of the names we will only take a closer look at encoding.

At the moment there comes only one encoding type with PVFS2: little endian bytefield (lebf), implemented in PINT-le-bytefield.c[10]. The call succession when using this encoding type for I/O server requests is diagrammed in figure 5.4.

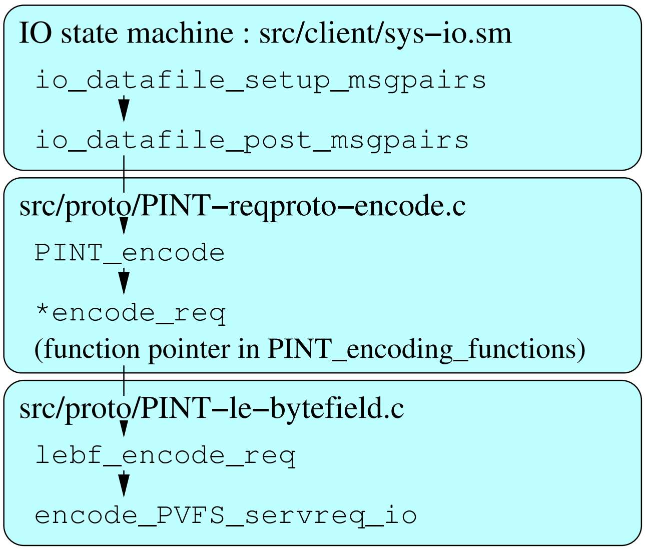

IO state machine : src/client/sys–io.sm
io_datafile_setup_msgpairs
io_datafile_post_msgpairs
src/proto/PINT–reqproto–encode.c
PINT_encode
*encode_req
(function pointer in PINT_encoding_functions)
src/proto/PINT–le–bytefield.c
lebf_encode_req
encode_PVFS_servreq_io

Figure 5.4.: Flow of encoding I/O server request

We now have to adjust this macro. We have luck as there already exists an encoding function for `uint32_t`[11] and we can simply use it.

```
#define encode_PVFS_servreq_io(pptr,x) do { \
    encode_PVFS_handle(pptr, &(x)->handle); \
    encode_PVFS_fs_id(pptr, &(x)->fs_id); \
    encode_skip4(pptr,); \
    [...]
    encode_PVFS_offset(pptr, &(x)->file_req_offset); \
    encode_PVFS_size(pptr, &(x)->aggregate_size); \
    encode_uint32_t(pptr, &(x)->pvshd_callid); \
} while (0)
```

Listing 5.15: Modified macro `encode_PVFS_servreq_io` in `pvfs2-req-proto.h`

`pptr` is a pointer to the particular current position in the encoded protocol and is increased by the correct value with every encoded component. `x` is the respective structure holding the request data to be encoded.

[10] ${PVFS2-SRC}/src/proto/PINT-le-bytefield.c

[11] `encode_uint32_t` is a macro in ${PVFS2-SRC}/src/proto/endecode-funcs.h

So we have adjusted the protocol data structure and the encoding for it. Still outstanding is the filling of the data structure. For this task there is a macro (Fancy that!). But for the moment we will not use it since we would have to copy it for having two independent versions then. If the changes sometimes become fully integrated the macro should be modified.

For now we set the CallID at the server request protocol structure in sys-io.sm after the call for `PINT_SERVREQ_IO_FILL` in `io_datafile_setup_msgpairs`:

```
static int io_datafile_setup_msgpairs(PINT_client_sm *sm_p,
                                      job_status_s *js_p)
{
[...]
        PINT_SERVREQ_IO_FILL(
            sm_p->u.io.contexts[i].msg.req,
            *sm_p->cred_p,
            sm_p->object_ref.fs_id,
            sm_p->u.io.contexts[i].data_handle,
            sm_p->u.io.io_type,
            sm_p->u.io.flowproto_type,
            sm_p->u.io.datafile_index_array[i],
            attr->u.meta.dfile_count,
            attr->u.meta.dist,
            sm_p->u.io.file_req,
            sm_p->u.io.file_req_offset,
            PINT_REQUEST_TOTAL_BYTES(sm_p->u.io.mem_req));

#ifdef PVSHD_CALLID
        // set pvshd_callid in server request protocol after macro for not
        // having to copy/change it. (TODO: fully integrate into fill macro?)
        (sm_p->u.io.contexts[i].msg.req).u.io.pvshd_callid = pvshd_callid;
#endif
[...]
}
```

Listing 5.16: Modified `io_datafile_setup_msgpairs` in sys-io.sm

5.3.3. Server side: From BMI to event manager

Before handling the last step on the CallID's long way up-down through the whole system let us take a deeper look into the PVFS2 servers structure. Here are some facts that will be helpful to remember:

- The PVFS2 server is doing its work with multiple threads, so the use of global variables is not advisable.
- There is one thread for each of the layers BMI, MainLoop/Job and Trove.

- For every task in the server the MainLoop/Job thread starts a state machine. It then do permanent checks for completed steps and and puts the respective state machine to the next state. Each state is implemented as a function that's return value decides about the following state.

- Each layer provides a data structure from which every instance represents one piece of work to be handled by the layer. An instance of such a struct in this document is called one *data object*. In each layer there can be multiple data objects at the same time.

- The layers make *use* of each other. This is where the name *user pointer* comes from. In the data object of the used layer it points to the responsible data object for the using layer.

Once a request is received by an PVFS2 server process a reference to it is available at the server's I/O state machine in io.sm[12]. From there on this request data is accessible from within each layer the request passes, since we can always climb up using the user pointers. The way an I/O request takes through the layers is illustrated in figure 5.5 on the next page. On the left side we see the important functions that are called in bottom-up order. On the right the data objects structure in each layer is shown. The arrows on the right side represent the references in the user pointers of the data objects. To get the reference to the CallID we have to go on the right side top-down through the user pointers and data objects.

One could think now we can easily get the CallID at least until the Trove layer. As we can also see in figure 5.5 on the following page this is unfortunately not as easy because all user pointers (and differently named pointers with the same functionality) are of the type `*void`. That means we need exactly one cast for each layer boundary the request passed. In addition we need to have all layer's data object structure to be known in the same place. This is normally neither the case nor intended.

First let us collect all the necessary information about structures by following the path in figure 5.5 and make the modifications necessary to get all the information into the Trove layer functions.

[12] ${PVFS2-SRC}/src/server/io.sm

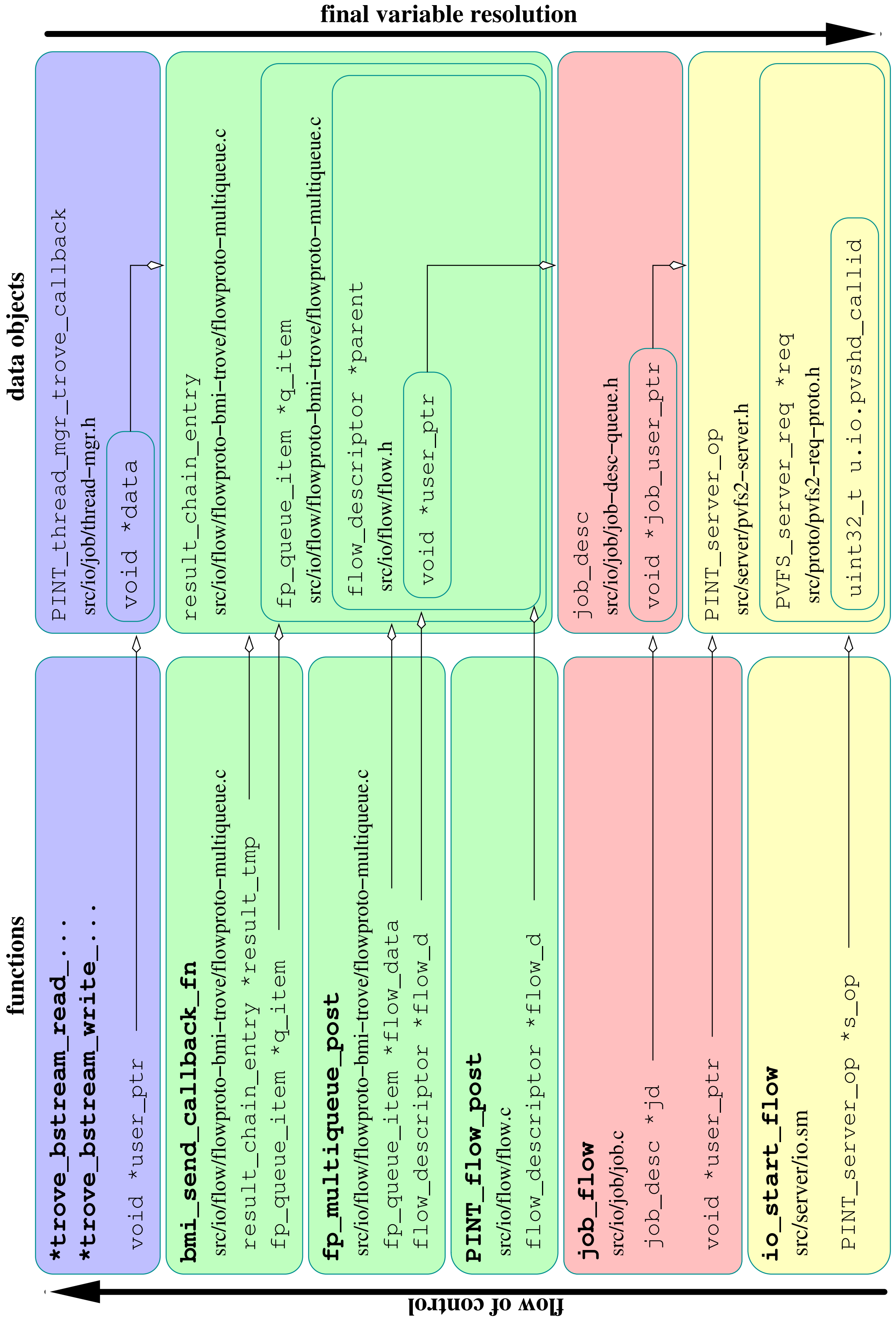


Figure 5.5.: Important functions on an I/O request's path through a PVFS2 server and accessibility of the CallID.

As we can see, we will need the definitions for

- result_chain_entry located in ${PVFS2-SRC}/src/io/flow/flowproto-bmi-trove/flowproto-multiqueue.c
- job_desc located in ${PVFS2-SRC}/src/io/job/job-desc-queue.h
- PINT_server_op located in ${PVFS2-SRC}/src/server/pvfs2-server.h

The filename extension of flowproto-multiqueue.c implies that this file is not designed to be included into other files. So it will be best to move the structure definitions into a new header file flowproto-multiqueue.h.

The newly created header looks like this:

```
#ifdef PVSHD_CALLID
/* put in header for easier inclusion in other files */
#ifndef __FLOWPROTO_MULTIQUEUE_H
#define __FLOWPROTO_MULTIQUEUE_H

#include "pvfs2-types.h"       // for PVFS_id_gen_t, PVFS_size, PVFS_offset
#include "pint-request.h"      // for PINT_Request_result
#include "thread-mgr.h"        // for PINT_thread_mgr_trove_callback,
                               //     PINT_thread_mgr_bmi_callback
#include "quicklist.h"         // for qlist_head
#include "src/io/flow/flow.h"  // for flow_descriptor

#define MAX_REGIONS 64

struct result_chain_entry
{
    [...]
};

extern struct PINT_perf_counter* PINT_server_pc;

/* fp_queue_item describes an individual buffer being used within the flow */
struct fp_queue_item
{
    [...]
};

#endif /* __FLOWPROTO_MULTIQUEUE_H */
#endif /* PVSHD_CALLID */
```

Listing 5.17: New header flowproto-multiqueue.h

We can include this header now into the original file instead of the outsourced definitions:

```
#ifdef PVSHD_CALLID
#include "src/io/flow/flowproto-bmi-trove/flowproto-multiqueue.h"
#endif
[...]
#ifndef PVSHD_CALLID
struct result_chain_entry
{
    [...]
};
[...]
struct fp_queue_item
{
    [...]
};
#endif
```

Listing 5.18: Modified head of flowproto-multiqueue.c

All the other headers are simple to include into the file we want to have the CallID in.

In figure 5.5 on page 40 we can also see that the interesting functions in Trove layer are those *trove_bstream_read_* and *trove_bstream_write_* point to. In fact the only two really used function pointers are *trove_bstream_read_list and *trove_bstream_write_list. The only trove implementation currently in the release is DBPF (DB Plus Files). So the functions we are looking for in there are dbpf_bstream_read_list and dbpf_bstream_write_list. They both just call the function dbpf_bstream_rw_list in dbpf-bstream.c[13].

We include the ascertained headers in the file and initialize a new variable pvshd_callid in the function:

```
[...]
#ifdef PVSHD_CALLID
/* for result_chain_entry */
#include "src/io/flow/flowproto-bmi-trove/flowproto-multiqueue.h"
/* for job_desc */
#include "src/io/job/job-desc-queue.h"
/* for PINT_server_op */
#include "src/server/pvfs2-server.h"
#endif
[...]
```

Listing 5.19: Modified head of dbpf-bstream.c

[13] ${PVFS2-SRC}/src/io/trove/trove-dbpf/dbpf-bstream.c

```
static inline int dbpf_bstream_rw_list(TROVE_coll_id coll_id,
                                       TROVE_handle handle,
                                       char **mem_offset_array,
                                       TROVE_size *mem_size_array,
                                       int mem_count,
                                       TROVE_offset *stream_offset_array,
                                       TROVE_size *stream_size_array,
                                       int stream_count,
                                       TROVE_size *out_size_p,
                                       TROVE_ds_flags flags,
                                       TROVE_vtag_s *vtag,
                                       void *user_ptr,
                                       TROVE_context_id context_id,
                                       TROVE_op_id *out_op_id_p,
                                       int opcode)
{
[...]
#ifdef PVSHD_CALLID
    /* getting callid directly from request protocol */
    struct PINT_thread_mgr_trove_callback *pvshd_trove_callback = user_ptr;
    struct result_chain_entry *pvshd_result_tmp = pvshd_trove_callback->data;
    struct job_desc *pvshd_jd = pvshd_result_tmp->q_item->parent->user_ptr;
    struct PINT_server_op *pvshd_sd = pvshd_jd->job_user_ptr;
    uint32_t pvshd_callid = pvshd_sd->req->u.io.pvshd_callid;
#endif
[...]
}
```

Listing 5.20: Modified `trove_bstream_rw_list` in dbpf-bstream.c

In the function `dbpf_bstream_rw_list` we can also find a call of the macro `DBPF_EVENT_START`. This is to tell the event manager the start of a new Trove operation. It calls the event manager's entry function `PINT_event_timestamp`. The event manager already gets informed of start and stop events occurring in Trove (and also in other layers) and it is enabled to log these events using MPE. What we want it to do in addition is for Trove events to put our CallID into the MPE info field. Fortunately it is sufficient to pass the ID only with the start event. The assignment between start and stop events belonging together would be difficult.

The mentioned function `PINT_event_timestamp` provides no possibility to pass the CallID to it. Now we are in a real bad situation: We cannot change the parameter list because the function is also used by the other layers and we do not want to touch everything in the PVFS2 code for now. After we will know about the benefit of all the things we are implementing at the moment, some deeper changes could be an option. Indeed since Frank Panse in our group in Heidelberg is still working on a new structure for the whole tracing this definitely will be an option (see also section 8 "Future Work" on page 73, [18]). For the moment however we have no good alternative and thus will copy the whole function and rename it to pass our CallID to the event manager.

Exactly the same problem is occurring in the event manager again with the function `__PINT_event_mpe` originally called from `PINT_event_timestamp`, so we have to copy and rename it too. The event managers functions are placed in pint-event.c[14]

Here are the copied and renamed macro ...

```
#define DBPF_PVSHD_EVENT_START(__op, __id, __callid)                     \
 PINT_PVSHD_event_timestamp(PVFS_EVENT_API_TROVE, __op, 0, __id,         \
 PVFS_EVENT_FLAG_START, __callid)
#endif
```

Listing 5.21: New copied macro `DBPF_PVDHS_EVENT_START` in dbpf.h

... and functions:

```
#ifdef PVSHD_CALLID
void __PINT_PVSHD_event_timestamp(enum PVFS_event_api api,
          int32_t operation,
          int64_t value,
          PVFS_id_gen_t id,
          int8_t flags,
          void* pvshd_ptr)
{
    gen_mutex_lock(&event_mutex);

#if defined(HAVE_PABLO)
    __PINT_event_pablo(api, operation, value, id, flags);
#endif

#if defined (HAVE_MPE)
    __PINT_PVSHD_event_mpe(api, operation, value, id, flags, pvshd_ptr);
#endif

    __PINT_event_default(api, operation, value, id, flags);

    gen_mutex_unlock(&event_mutex);

    return;
}
#endif
```

Listing 5.22: New copied `PINT_PVSHD_event_timestamp` in pint-event.c

[14] ${PVFS2-SRC}/src/common/misc/pint-event.c

```
#ifdef PVSHD_CALLID
void __PINT_PVSHD_event_mpe(enum PVFS_event_api api,          2
          int32_t operation,
          int64_t value,                                      4
          PVFS_id_gen_t id,
          int8_t flags,                                       6
          void* pvshd_ptr)
{                                                             8
  MPE_LOG_BYTES    bytebuf;       // buffer for logging of callid
  int              bytebuf_pos;                               10
  uint32_t callid = *((uint32_t *) pvshd_ptr);
                                                              12
  switch(api) {
    case PVFS_EVENT_API_BMI:                                  14
      // never coming here
      break;                                                  16
    case PVFS_EVENT_API_JOB:
      // never coming here                                    18
      break;
    case PVFS_EVENT_API_TROVE:                                20
      if (flags & PVFS_EVENT_FLAG_START) {
        bytebuf_pos = 0;                                      22
        MPE_Log_pack( bytebuf, &bytebuf_pos, 'x', 1, &callid);
        if (operation == PVFS_EVENT_TROVE_READ_LIST) {        24
          MPE_Log_event(PINT_event_trove_rd_start, 0, bytebuf);
        } else if (operation == PVFS_EVENT_TROVE_WRITE_LIST) {  26
          MPE_Log_event(PINT_event_trove_wr_start, 0, bytebuf);
        }                                                     28
      } else if (flags & PVFS_EVENT_FLAG_END) {
        // never coming here                                  30
      }
      break;                                                  32
    case PVFS_EVENT_API_ENCODE_REQ:
    case PVFS_EVENT_API_ENCODE_RESP:                          34
    case PVFS_EVENT_API_DECODE_REQ:
    case PVFS_EVENT_API_DECODE_RESP:                          36
    case PVFS_EVENT_API_SM:
      break; /* XXX: NEEDS SOMETHING */                       38
    }
}                                                             40
#endif
```

Listing 5.23: New copied __PINT_PVSHD_event_mpe in pint-event.c

In lines 22 to 27 in listing 5.23 we try to give MPE an info value for trove states that it does not expect yet. As we know (see section 5.2 “Creation of CallID for MPI-IO Function Calls” on page 26) the first step to put something into the info field is to set the format in MPE state declaration.

In the event manager that means to modify the state describing function calls in `init_mpeLogFile` defined in pint-event.c:

```
void init_mpeLogFile(void)
{
[...]
#ifdef PVSHD_CALLID
    MPE_Describe_info_state(PINT_event_trove_rd_start, PINT_event_trove_rd_stop,
      "Trove␣Read", "orange", "CallID=0x%x");
    MPE_Describe_info_state(PINT_event_trove_wr_start, PINT_event_trove_wr_stop,
      "Trove␣Write", "blue", "CallID=0x%x");
#else
    MPE_Describe_state(PINT_event_trove_rd_start, PINT_event_trove_rd_stop,
      "Trove␣Read", "orange");
    MPE_Describe_state(PINT_event_trove_wr_start, PINT_event_trove_wr_stop,
      "Trove␣Write", "blue");
#endif
[...]
}
```

Listing 5.24: Modified `init_mpeLogFile` in pint-event.c

Only to play safe after adding an info tag to the MPE state declaration, we will also modify the original `__PINT_event_mpe` to pass a CallID to MPE. We cannot be sure that it is not still called with Trove signature from somewhere else in the system, although this should not be the case. We take ’-1’[15] to identify these cases in the trace.

```
void __PINT_event_mpe(enum PVFS_event_api api,
          int32_t operation,
          int64_t value,
          PVFS_id_gen_t id,
          int8_t flags)
{
#ifdef PVSHD_CALLID
  MPE_LOG_BYTES  bytebuf;      // buffer for logging of callid
  int            bytebuf_pos;
  uint32_t nocallid = -1;
#endif
  switch(api) {
    case PVFS_EVENT_API_BMI:
      [...]
    case PVFS_EVENT_API_JOB:
      [...]
    case PVFS_EVENT_API_TROVE:
      if (flags & PVFS_EVENT_FLAG_START) {
#ifdef PVSHD_CALLID
        bytebuf_pos = 0;
        MPE_Log_pack( bytebuf, &bytebuf_pos, 'x', 1, &nocallid);
        if (operation == PVFS_EVENT_TROVE_READ_LIST) {
          MPE_Log_event(PINT_event_trove_rd_start, 0, bytebuf);
```

[15] ’-1’ is the highest unsigned integer available, if CallID reaches that value it runs our of range anyway

```
        } else if (operation == PVFS_EVENT_TROVE_WRITE_LIST) {
          MPE_Log_event(PINT_event_trove_wr_start, 0, bytebuf);
        }
#else
        if (operation == PVFS_EVENT_TROVE_READ_LIST) {
          MPE_Log_event(PINT_event_trove_rd_start, 0, NULL);
        } else if (operation == PVFS_EVENT_TROVE_WRITE_LIST) {
          MPE_Log_event(PINT_event_trove_wr_start, 0, NULL);
        }
#endif
      } else if (flags & PVFS_EVENT_FLAG_END) {
        if (operation == PVFS_EVENT_TROVE_READ_LIST) {
          MPE_Log_event(PINT_event_trove_rd_stop, value, NULL);
        } else if (operation == PVFS_EVENT_TROVE_WRITE_LIST) {
          MPE_Log_event(PINT_event_trove_wr_stop, value, NULL);
        }
      }
      break;
    case PVFS_EVENT_API_ENCODE_REQ:
    case PVFS_EVENT_API_ENCODE_RESP:
    case PVFS_EVENT_API_DECODE_REQ:
    case PVFS_EVENT_API_DECODE_RESP:
    case PVFS_EVENT_API_SM:
      break; /* XXX: NEEDS SOMETHING */
    }
}
```

Listing 5.25: Modified `__PINT_event_mpe` in pint-event.c

Now we created a CallID for each MPI-IO function call and pushed it through the whole system. The same CallID value becomes logged with the MPI-IO event and all resulting Trove events. So we are able to produce one trace containing all client MPI-IO function calls and all Trove events having CallIDs and connecting them is now possible. The time has come to make the connections really visible.

5.4. Statistic and Visualisation Tools

First we write a text-based tool to process statistics from the output of *slog2print*. Afterwards we create another tool to put arrows representing our connections into the merged slog2 trace file. These arrows are for visualization in Jumpshot.

5.4.1. Tool for analyse output of *slog2print*

As described in section 4.3.1 "Text Based Analysis" on page 21 we will write a reference Perl script in order to make it easy for users to get exactly that statistical information they really need. For that, they just have to use our reference implementation and extend some functions to make it collecting the data they require.

We start with a inline description of the script and declare some global variables:

```
#!/usr/bin/perl -tw
#
###
#
# This script is to create statistics from the output of slog2print.
# It reads in the output of slog2print from stdin and writes some statistics to
# stdout. It is designed to process the output slog2print creates by default.
#
# Call: slog2print <SLOG2_TRACE> | slog2stats
#
# For the moment this should just be a reference for easy implementation of
# further statistic generating functionality:
#
# * The categories get read into a hash of hashes to make them easy to access.
# * The categories containing the CallID info tag get counted.
# * The drawables get counted (this should result the same number as the last
#   output line from slog2print)
# * The arrows Slog2ToArrowSlog2 would create are counted for each different
#   CallID value and an average number is calculated.
#
###
[...]

use strict;
use warnings;
use diagnostics;

# currently parsing part
my $currentPart = "";

# for counting the categories with CallID info type
my $infocat_counter = 0;

# for counting drawable objects
my $drawable_counter = 0;

# for counting of occurrence of each CallID
my %arrow_counters;

# hash of hashes for categories
my %categories;

# count consecutive blank lines, after three we are in drawables list
my $blcount;
```

Now the script reads from stdin line by line. First it checks if the line identifies the start of a new section in the *slog2print* output. If so the section identifier is set to the new section and it goes to process the next line, else the normal line content processing starts:

```
# read stdin line by line
while(<STDIN>) {
  chomp;

  if ( $_ =~ /SLOG-2 Header:/ ) { $currentPart="Header"; }
  elsif ( $_ =~ /CategoryMap:/ ) { $currentPart="CategoryMap"; }
  elsif ( $_ =~ /SLOG-2 Tree Directory/ ) { $currentPart="TreeDirectory"; }
  elsif ( $_ =~ /LineIDMapList:/ ) { $currentPart="LineIDMapList"; }
  elsif ( $_ =~ /^$/ ) { $currentPart="DrawableList" if ++$blcount==3; next; }
  else { $blcount = 0; goto contentcheck; }
  $blcount = 0;
  next;
```

The script does only the processing specified for the current section. This improves speed and avoids mismatches if there are similar lines in different sections with different meanings.

In the *Header* section we only check if the SLOG2 version is the one the script is written for and print a warning if not the case:

```
  contentcheck:

  if ( $currentPart eq "Header" ) {
    if ( $_ =~ /^version/ ) {
      $_ =~ /= SLOG 2.0.6$/ || print "Warning:␣Input␣has␣wrong␣version\n";
    }
  }
```

In *Category Map* section we fill the categories hash to have this information still in later sections available. Also we increase the counter for categories with CallID for each found one.

```
  if ( $currentPart eq "CategoryMap" ) {
    next unless $_ =~ /^[0-9]+: Category\[ (.*) \]$/;

    my %category;
    my $category_tmp = $1;
    # special handling for info_fmt, it could contain ',' or '='
    if ( $category_tmp =~ /info_fmt=< (.*) >, vis/ ) {
      $category{"info_fmt"} = $1;
      $category_tmp =~ s/$&/vis/;
    }
    my @keyvalues = split /, /, $category_tmp;
    foreach ( @keyvalues ) {
      my ( $key, $value );
      ($key, $value) = split /=/, $_, 2;
      $category{ $key } = $value;
    }
    # count all categories with CallID info type
    if ( exists $category{"info_fmt"} ) {
      $infocat_counter++ if $category{"info_fmt"} =~ /CallID/;
    }
    # add reference to that category to categories
    $categories{$category{"index"}} = \%category;
   }
```

In the sections *TreeDirectory* and *LineIDMapList* the script will do nothing for the moment, although we prepare them for later processing:

```
 elsif (  $currentPart eq "TreeDirectory" ) {
  # nothing to analyse here for the moment
 }

 elsif (  $currentPart eq "LineIDMapList" ) {
  # nothing to analyse here for the moment
 }
```

In *DrawableList* section we use the same algorithm as described for the use in Slog2ToArrowSlog2 in section 4.3.2 “Graphical Analysis” on page 22 listing 4.2 on page 22. The difference is that we do not create arrows here but count how many we would create for each CallID.

```
 elsif (  $currentPart eq "DrawableList" ) {
  next unless
     $_ =~ /^[0-9]+: (?:Primitive)|(?:Composite)\[ (.*) \] bsize=[0-9]+$/;

  # count the drawables
  $drawable_counter++;

  # count arrows of each CallID value
  # the first occurrence becoming number 0 is the begin of all arrows
  if ( $_ =~ /CallID=0x([0-9A-F]+)/ ) {
   if ( exists $arrow_counters{$1} ) { $arrow_counters{$1}++; }
   else { $arrow_counters{$1} = 0; }
  }
 }
}
```

Finally we calculate how many arrows would be created on average for each CallID and print out the compiled statistics:

```
## calculate average number of arrows per CallID
my $arrows = 0;
my $arrowstarts = 0;
foreach my $callid ( keys %arrow_counters ) {
  $arrowstarts++ if $arrow_counters{$callid} > 0;
  $arrows += $arrow_counters{$callid};
}

# print statistics
print "\nStatistics:\n";
print "Found␣".$infocat_counter."␣categories␣with␣CallID␣info␣key.\n";
print "Counted␣".$drawable_counter."␣drawables.\n";
print "Found␣". keys ( %arrow_counters ) ."␣different␣CallID␣values.\n";
if ( $arrowstarts != 0 ) {
  print "On␣average␣there␣are␣". $arrows/$arrowstarts
                                        ."␣arrows␣per␣CallID␣having␣any.\n";
} else {
  print "There␣could␣no␣arrow␣be␣found.\n";
}
print "\n";
```

5.4.2. Tool creating arrows for visualisation in *jumpshot*

For implementing *Slog2ToArrowSlog2* we will use the functionality of the SLOG2-SDK Java API. Since we do not have any API documentation we can use as a reference only the implementations of several tools[16] and some comments is the SDK code itself. For our part we will make extensive use of Javadoc, and code annotation in general to make our implementation as well documented as possible, even the parts adopted form other code. Here we will only discuss some code fragments essential for the core functionality of *Slog2ToArrowSlog2* to create arrows connecting CallIDs.

[16] *slog2filter*, *slog2print*, *MergeSlog2*, *Slog2ToCompositeSlog2* etc.

After reading the input slog2 file and having all the contents accessible we create new categories for the arrows. We need three types, one for arrows pointing to "Trove Read", one for arrows pointing to "Trove Write" and one for arrows pointing to unknown:

```
// generate new categories for the arrows and put them into maps
String[] arrowInfoKeys = { "CallID=0x" };
InfoType[] arrowInfoTypes = {InfoType.BYTE4};

Category readArrowCategory = getNewArrowCategory("Read␣Path",
                                                 new ColorAlpha(255, 165, 0));
readArrowCategory.setInfoKeys(arrowInfoKeys);
readArrowCategory.setInfoTypes(arrowInfoTypes);
category.put(new Integer(readArrowCategory.getIndex()), readArrowCategory);

Category writeArrowCategory = getNewArrowCategory("Write␣Path",
                                                  new ColorAlpha(0, 0, 255));
writeArrowCategory.setInfoKeys(arrowInfoKeys);
writeArrowCategory.setInfoTypes(arrowInfoTypes);
category.put(new Integer(writeArrowCategory.getIndex()), writeArrowCategory);

Category miscArrowCategory = getNewArrowCategory("Misc␣Path",
                                                 new ColorAlpha(255, 255, 255));
miscArrowCategory.setInfoKeys(arrowInfoKeys);
miscArrowCategory.setInfoTypes(arrowInfoTypes);
category.put(new Integer(miscArrowCategory.getIndex()), miscArrowCategory);
```

Listing 5.26: *Slog2ToArrowSlog2*: Creating new categories for arrows in `main()`

We use the method `getNewArrowCategory` to create a new arrow category. There exists no such method yet, so we also have to implement it:

```
/**
 * Returns a new category with arrow topology. Takes care about getting an
 * unused category index.
 *
 * <P>Better to put this in Category class, at least the part to get the index.
 * Perhaps better to get _one_ unused index, not the lowest higher than all
 * used ?!?</P>
 *
 * @param topo_name
 *            name for the new category
 * @param arrow_color
 *            color for the arrows
 * @return the created category
 */
private static Category getNewArrowCategory(String topo_name,
        ColorAlpha arrow_color) {

    // find unused category index, take lowest higher than all existing
    Iterator category_itr;
    Category objdef;
    int new_idx = 0;
    category_itr = category.values().iterator();
    while (category_itr.hasNext()) {
        objdef = (Category) category_itr.next();
        if (new_idx <= objdef.getIndex())
            new_idx = objdef.getIndex() + 1;
    }

    // create new arrow category
    Category newCategory = new Category(new_idx, topo_name, Topology.ARROW,
            arrow_color, 1);
    return newCategory;
}
```

Listing 5.27: *Slog2ToArrowSlog2*: Method implementation of `getNewArrowCategory`

Having categories for the arrows we will now implement our algorithm to find where to create arrows (see listing 4.2 on page 22) developed in section 4.3.2 "Graphical Analysis" on page 22:

```
// sort drawables in start time order
// arrows should go from start to start
Collections.sort(drawObject, Drawable.INCRE_STARTTIME_ORDER);

// create arrows from the first occurrence of each CallID to all
// following.
ArrayList newArrows = new ArrayList();
ArrayList handledIds = new ArrayList();
Drawable currentStart;
Drawable currentEnd;
Integer currentStartCallId;
Integer currentEndCallId;
for ( int idx_start=0; idx_start < drawObject.size(); ++idx_start) {
  // get CallID of current start drawable, if none test next
  currentStart = (Drawable) drawObject.get(idx_start);
  currentStartCallId = getCallIdFromDrawable(currentStart);
  if ( currentStartCallId == null )
    continue;
  // if id is already handled we found an endpoint, try next
  if ( handledIds.contains(currentStartCallId))
    continue;
  // save id as handled
  handledIds.add(currentStartCallId);
  // search for all following with same CallID and create arrows
  for ( int idx_end=idx_start+1; idx_end < drawObject.size(); ++idx_end ) {
    // get CallID of current end drawable, if none test next
    currentEnd = (Drawable) drawObject.get(idx_end);
    currentEndCallId = getCallIdFromDrawable(currentEnd);
    if ( currentEndCallId == null )
      continue;
    // if current end CallID doesn't match current start CallID
    // test next
    if (currentEndCallId.compareTo(currentStartCallId) != 0)
      continue;
    // create arrow according to arrow end category
    if (currentEnd.getCategory().getName().compareTo("Trove␣Read") == 0) {       36
      newArrows.add(getNewArrowDrawable(readArrowCategory, currentStart,
                                        currentEnd, currentEndCallId));        38
    }
    else if (currentEnd.getCategory().getName().compareTo("Trove␣Write") == 0) {  40
      newArrows.add(getNewArrowDrawable(writeArrowCategory, currentStart,
                                        currentEnd, currentEndCallId));        42
    }
    else {                                                                     44
      newArrows.add(getNewArrowDrawable(miscArrowCategory, currentStart,
                                        currentEnd, currentEndCallId));        46
    }
  }                                                                            48
}
```

Listing 5.28: *Slog2ToArrowSlog2*: Finding where to create arrows in `main()`

In lines 16 and 28 in listing 5.28 on the preceding page we use the method `getCallIdFromDrawable` to get the CallID value for one drawable. In the lines 37, 41 and 45 we use `getNewArrowDrawable` to create a new arrow drawable. These are both methods we have to implement by ourself, too.

Let us first take a look at implementing `getCallIdFromDrawable`:

```
/**
 * Returns value of CallID in the drawable's info field. Returns
 * null if no CallID exists in drawable.
 *
 * @param in_draw
 *           drawable to get CallID from
 * @return CallID
 */
private static Integer getCallIdFromDrawable(Drawable in_draw) {
    Integer info_idx;
    InfoValue currentCallId;
    // if the drawable has no info field return null
    if(in_draw.getInfoLength() < 1)
        return null;
    // if categories with CallIDs are not identified yet do it
    if ( callIdCategories == null ) {
        callIdCategories = new HashMap();
        identifyCallIdCategories();
    }
    // if category of current has no CallID info return null
    if(!callIdCategories.containsKey(in_draw.getCategory()))
        return null;
    info_idx = (Integer) callIdCategories.get(in_draw.getCategory());    24
    currentCallId = in_draw.getInfoValue(info_idx.intValue());
    return Integer.valueOf(currentCallId.toString(),16);                 26
}
```

Listing 5.29: *Slog2ToArrowSlog2*: Method implementation of `getCallIdFromDrawable`

We use another method here in line 18 when the method is called for the first time. It then calls `identifyCallIdCategories` to create a list of all these categories that actually have a CallID info field:

```
/**
 * Identifies each category with CallID info field and map it to the index
 * of the info value. This method fills the static variable
 * callIdCategories.
 *
 * @see callIdCategories
 */
private static void identifyCallIdCategories() {
    System.out.println("identifyCallIdCategories()");
    Iterator itr_category = category.values().iterator();
    Category checkCategory;
    String[] checkInfoKeys;
    while(itr_category.hasNext()) {
        checkCategory = (Category) itr_category.next();
        checkInfoKeys = checkCategory.getInfoKeys();
        // if has no info keys check next categorie;
        if( checkCategory.getInfoKeys() == null
              || checkCategory.getInfoKeys().length == 0)
            continue;
        for( int i = 0; i < checkInfoKeys.length; ++i ) {
            // if is not a CallID info check next info key
            if ( checkInfoKeys[i].compareTo("CallID=0x") != 0 )
                continue;
            // map category to index of CallID info
            callIdCategories.put(checkCategory,Integer.valueOf(i));
        }
    }

}
```

Listing 5.30: *Slog2ToArrowSlog2*: Method implementation of `getCallIdFromDrawable`

There was a second method used in the code above (listing 5.28 on page 53). Let us implement the method actually creating the arrows `getNewArrowDrawable`:

```
 /**
 * Creates an arrow drawable of the given category and set its CallID. The
 * arrow will begin at the start time of in_begin and end at the start time
 * of in_end. The info value is set to the given CallID.
 *
 * @param in_category
 *            arrow category with first and only info key for CallID
 * @param in_begin
 *            drawable the arrow shell begin at
 * @param in_end
 *            drawable the arrow shell and at
 * @param callId
 *            CallID to fill info field
 */
private static Primitive getNewArrowDrawable(Category in_category,
        Drawable in_begin, Drawable in_end, Integer callId) {

    // set begin and end time
    double begintime = in_begin.getEarliestTime();
    double endtime = in_end.getEarliestTime();

    double[] time_coords = new double[2];
    time_coords[0] = begintime;
    time_coords[1] = endtime;

    // set begin and end line
    int[] y_coords = new int[2];
    y_coords[0] = in_begin.getStartVertex().lineID;
    y_coords[1] = in_end.getStartVertex().lineID;

    // reencode CallID to byte[]
    ByteArrayOutputStream bary_ous = new ByteArrayOutputStream();
    DataOutputStream d_ous = new DataOutputStream(bary_ous);
    try {
         d_ous.writeInt(callId.intValue());
    }
    [...]
    byte[] byte_infovals = bary_ous.toByteArray();

    // have to do this for everything to be present in Primitive
    // TODO: better way ???
    Primitive prime = new Primitive(in_category.getIndex(), begintime,
            endtime, time_coords, y_coords, byte_infovals);
    Primitive prime2 = new Primitive(in_category, prime);
    prime2.setInfoBuffer(byte_infovals);

    return prime2;
}
```

Listing 5.31: *Slog2ToArrowSlog2*: Method implementation of `getNewArrowDrawable`

The somewhat strange looking creating of a temporary `Primitive` at line 42 is necessary due to the fact that there is no constructor taking category reference and info value and also no method to set category reference after creation.

Now we have got all the new arrows created and put in our list of arrows. Finally we merge this list of arrows to the list of the drawables from the input file and write it together with everything else, especially with the new category information into the output slog2 file:

```
// add arrows to output drawables
drawObject.addAll(newArrows);

// sort drawables in end time order
Collections.sort(drawObject, Drawable.INCRE_FINALTIME_ORDER);

// put all drawables into the tree for output
for (int i = 0; i < drawObject.size(); i++) {
    addDrawable(treetrunk, (Drawable) drawObject.get(i));
}

// write everything to output file
treetrunk.flushToFile();
category.removeUnusedCategories();
slog_outs.writeCategoryMap(category);
slog_outs.writeLineIDMapList(lineIDmap);
slog_outs.close();
```

Listing 5.32: *Slog2ToArrowSlog2*: Creating new categories for arrows in `main()`

All methods and parts of code not explicitly mentioned here we could copy and use without or with only few modification from other tools in slog2tools package (see section 2.4 “Trace Analysis Tools” on page 12). To this adapted code we mainly had to extend annotation.

5.5. Limitations of Implementation

Scalability

Due to the choice of the **data type for our CallID** we have a natural limitation in scalability. With the `uint32_t` that we use to store and transfer, we have only 32 bits for our complete CallID. We split these 32 bits into an 8 bit part for the process and an 24 bit part for the call counter (see figure 5.1 on page 31). That means the maximum of MPI processes we can have with unique CallIDs is $2^8 = 256$. The maximum of MPI-IO function calls one process can invoke until the counter overflows is $2^{24} = 16.777.216$. If someone use to run more processes, he simply could increase the number of bits to use for the process part (`PVSHD_PROCBITS`). But this automatically would decrease the number of bits for the counter part by the same amount. So for example using the ratio 12 : 20 would result in $2^{12} = 4096$ potential processes but therefore the counter would overflow after $2^{20} = 1.048.576$ yet.

Another point is the **amount of memory used for logging**. All tools in the *slog2tools* package using the SLOG-2 SDK (concerning *Slog2Merge* and *Slog2ToArrowSlog2*) are first loading the whole input files into the memory. To make a good proposition about scalability problems with that point, first some analysis of the resulting size of SLOG-2 formatted traces have to be done. An important point that has to be handled when doing this analysis in the future will be in what proportion the number of MPI-IO calls is to the number of resulting Trace operations in real applications. With such information one can say if the memory or the limitation in the CallID value range is the more serious scalability problem.

Overlapping events in the PVFS2 server

MPE2 creates the trace for the PVFS2 servers as a clog2 file. This is then to be converted into SLOG-2 format using *clog2TOslog2* (see figure 2.1 on page 7). The currently used method is not able to correctly handle cases of overlapping events of the same type (like "Trove Read" or "Trove Write"). This occures most notably if there are several clients sending requests to one server in parallel. This is a very important point to take care about. Frank Panse has already began to resolve this problem in the scope of his diploma thesis [18] (see also section 8.1 “Work in Progress” on page 73).

5.6. Implementation Problems

While implementing the described functionality we also faced to some problems. Mainly they come from a big lack of annotation in the code and detailed documentation of the code in its current state. Here are a few words to the several modified parts:

MPE2

The modifications in MPE2 have not been very difficult. The code is easy to understand and the documentation is sufficient.

PVFS2

An overall problem we had to deal with is the lack of annotation in the PVFS2 code.

To implement the modularity the programmers have used many preprocessor macros that cannot be handled by common IDEs and are even difficult to trace by hand. Much time was necessary to understand some single steps in the code. To point it, difficulties mainly arise at the interfaces of the layers and modules.

1.4.0 Issue: small I/O With the PVFS2 1.4.0 release a new request type was introduced: *small I/O*. This request type is to be used for very small (<16KB using TCP) amount of I/O data. The main difference to the normal I/O request is the handling in the PVFS2 server. The state machine for small I/O does not create a Flow but passes the data that are all included in one message directly to Trove.

This new feature is problematic for us because it creates a second path that an I/O request can take through the server. That means for us at the end we cannot say which path to climb up back through the data objects to get the CallID. Since our implementation had been too advanced at the time of the new release we could not change the whole design anymore to meet this new situation. So we decide to deactivate the new option now and add the section 8.2.2 "Enable Small I/O" on page 74 to chapter 8 "Future Work".

The deactivation is done by disabling the creation of small I/O requests in the PVFS2 client if `PVSHD_CALLID` is set:

```
[...]
#ifndef PVSHD_CALLID // XXX: never do small I/O with CallIDs

    /* look at sio_array and sio_count to see if there are any
     * servers that we can do small I/O to, instead of setting up
     * flows.  For now, we're going to stick with the semantics that
     * small I/O is only done if all of the sizes for the target datafiles
     * are small enough (sio_count == target_datafile_count).  This can
     * be changed in the future, for example, if sio_count is some
     * percentage of the target_datafile_count, then do small I/O to
     * the sio_array servers, etc.
     */
    if(sio_count == target_datafile_count)
    {
        gossip_debug(GOSSIP_IO_DEBUG, "␣␣%s:␣doing␣small␣I/O\n", __func__);

        sm_p->u.io.small_io = 1;
        js_p->error_code = IO_DO_SMALL_IO;
        goto sio_array_destroy;
    }
#endif
[...]
```

Listing 5.33: Deactivate small I/O in `io_datafile_setup_msgpairs` in sys-io.sm

slog2stats

There have been no problems with implementation. The output of *slog2print* is self-explanatory.

Slog2ToArrowSlog2

As already mentioned there is no documentation for the SLOG-2 SDK API. Also the annotation in the source code is only insufficient and no use is made of the great documentation feature for Java: Javadoc.

Furthermore the access methods for some object presenting classes in the SDK are deficient. For example it seems to be impossible to set the category or the info file to a drawable after creation time and even at creation time it is difficult to set both.

5.7. Summary

Despite these difficulties we now have an MPICH2+PVFS2 system that can produce traces with CallID information in MPI-IO and PVFS2 server Trove events. This information is useful to make connections between MPI-IO function calls and triggered disk operations. MPE is used as logging facility. It produces clog2 files and merges the traces from the single MPI processes as well as the traces from the single PVFS2 server processes for us since we already run the PVFS2 server daemon as MPI program. The two traces, one from the client, one from the server side, we can convert into the visualization oriented SLOG-2 format and merge them together into one trace. We can then use *slog2print* and our new tool *slog2stats* to get some statistical information about the trace, including the average number of Trove events triggered by one MPI-IO function call. Furthermore, using the completely rewritten tool *Slog2ToArrowSlog2*, we are able to create arrows in the trace that represent the connections between MPI-IO function calls and the respectively resulting Trove operations.

Yet unclear issues are the handling of collective MPI-IO calls and a better integration of the changes into the code in almost every part of the MPICH2+MPE2+PVFS2 system. Also outstanding is the reactivation of small I/O requests in PVFS2 and their handling concerning CallIDs.

6. Example

This chapter gives a step by step introduction to get a working system environment. After this an example is described to illustrate the development made in the scope of this thesis.

The concept of installing and running MPI programs with MPE and PVFS2 in the manner described here has been initially developed by Julian Kunkel[1]. The author of this thesis has extended it by the use of VPATH concept and formalized and translated the description.

The software versions to start with:

MPICH2:	1.0.3
MPE2:	1.0.3p1
PVFS2:	1.4.0
slog2tools	0.9.1

Our test cluster is running Debian GNU/Linux 3.1 "sarge" i686 with original Debian kernel image 2.6.8-2-686-smp in version 2.6.8-16sarge1.

6.1. Installation of the Environment

First we make some assumptions:

${MPICH2_SRC}	–	MPICH2 source tree
${MPICH2_PLAIN_BUILD}	–	build directory for MPICH2 *without* PVFS2
${MPICH2_PLAIN}	–	target directory for MPICH2 *without* PVFS2
${MPICH2_BUILD}	–	build directory for MPICH2 *with* PVFS2
${PVFS2_SRC}	–	PVFS2 source tree
${PVFS2_BUILD}	–	build directory for MPICH2 *with* PVFS2
${PREFIX}	–	target directory for MPICH2 *with* PVFS2 and for PVFS2
${JAVADIR}	–	toplevel directory of a working Java SDK (>=1.5)
${WORKDIR}	–	directory for configurations, the example and traces

The only important point to consider is that ${PREFIX} and ${WORKDIR} must be accessible on every node in the same path.

[1] Many thanks to Julian for his work on that

In MPICH2 *1.0.3* there is MPE2 *1.0.3* included. Since we use MPE2 *1.0.3p1* the one in MPICH2 needs to be replaced. To do this

1. unpack mpe2-1.0.3p1.tar.gz somewhere, thus creating the directory mpe2-1.0.3p1

2. delete or rename ${MPICH2_SRC}/src/mpe2

3. create a symbolic link ${MPICH2_SRC}/src/mpe2 pointing to the mpe2-1.0.3p1 directory

6.1.1. Patching the sources

The enhancement on PVFS2 was originally done on base of the CVS version from 2005/12/07 21:10:16 (see [1]). However when version 1.4.0 was released on 2006/02/15 we adapted everything to the new version. Nevertheless there are some patches needed in addition to our modifications because some functionality we base our development on is not in official release yet.

The necessary patches are[2]:

mpich2-1.0.3_fixmpdboot	fixes a problem while starting MPD ring with mpdboot
mpe2-1.0.3p1_callid	our changes for CallID logging in MPE2
pvfs2-1.4.0_fixevents	fixes two issues in event logging: Trove Reads are no longer logged also as Writes; BMI and Jobs are no longer logged repeatedly.
pvfs2-1.4.0_mpirun	makes running as MPI program possible: adds `--with-mpiexec` compiler option and host-specific server.conf
pvfs2-1.4.0_writelog	enables trace writing after reset of event-mask
pvfs2-1.4.0_callid	our changes for CallID logging in PVFS2

They can be applied as follows:

```
$ cd ${MPICH2_SRC}
$ gunzip -c mpich2-1.0.3_fixmpdboot.patch.gz | patch -Np1


$ cd ${MPICH2_SRC}/src/mpe2
$ gunzip -c mpe2-1.0.3p1_callid.patch.gz | patch -Np1


$ cd ${PVFS2_SRC}
$ gunzip -c pvfs2-1.4.0_fixevents.patch.gz | patch -Np1
$ gunzip -c pvfs2-1.4.0_mpirun.patch.gz | patch -Np1
$ gunzip -c pvfs2-1.4.0_writelog.patch.gz | patch -Np1
$ gunzip -c pvfs2-1.4.0_callid.patch.gz | patch -Np1
```

[2] Thanks again to Julian Kunkel for developing the mpirun and writelog changes

6.1.2. Installing plain MPICH2/MPE2 for building PVFS2

The first step when installing the environment is to build and install a plain MPICH2 without PVFS2. This is necessary to have the libraries to bind to PVFS2.

```
$ cd ${MPICH2_PLAIN_BUILD}
$ ${MPICH2_SRC}/configure --with-mpe \
$                         --prefix=${MPICH2_PLAIN} \
$                         CFLAGS="-DPVSHD_CALLID"
$ make; make install
```

6.1.3. Installing PVFS2 with MPI and MPE

Second we build PVFS2 with support for MPI and MPE. We use the just created plain MPICH2 installation.

```
$ cd ${PVFS2_BUILD}
$ ${PVFS2_SRC}/configure --with-mpiexec=${MPICH2_PLAIN} \
$                        --with-mpe=${MPICH2_PLAIN} \
$                        --prefix=${PREFIX} \
$                        CFLAGS="-DPVSHD_CALLID"
$ chmod 755 ${PVFS2_SRC}/maint/*.sh
$ make; make install
```

6.1.4. Installing MPICH2 with PVFS2 support for Logging

The third step is to build and install the MPICH2 version with PVFS2 support we will finally use.

```
$ cd ${MPICH2_BUILD}
$ ${MPICH2_SRC}/configure --enable-romio \
$                         --with-file-system=ufs+nfs+pvfs2 \
$                         --prefix=${PREFIX} \
$                         --with-java2=${JAVADIR} \
$                         --with-mpe \
$                         CFLAGS="-DPVSHD_CALLID \
$                                 -I${PREFIX}/include \
$                                 -L${PREFIX}/lib" \
$                         LIBS="-lpvfs2"
$ make; make install
```

If we do have an appropriate Java version in our path we can omit the `--with-java2` option.

6.1.5. Setting up MPD ring

MPD is only working with a secret word set, so we have to set one:

```
$ echo "secretword=pvshd" > ~/.mpd.conf
$ chmod 600 ~/.mpd.conf
```

Moreover it needs to have the file mpd.hosts containing which nodes to use. We have to put in there at least the hostnames of the nodes we want to use as PVFS2 servers, one per line.

For our example we use `master1`, `node01` and `node02`:

```
$ cd ${WORKDIR}
$ echo -e "master1\nnode01\nnode02" > mpd.hosts
```

Now everything is prepared to start the MPD ring:

```
$ ${PREFIX}/bin/mpdboot --totalnum=3 --rsh=rsh --file=mpd.hosts
```

mpdboot starts the MPD ring using the first `totalnum` hosts entries in mpd.hosts. It uses the remote shell to start the remote daemons, so rsh has to work non-interactively.

To test if the MPD ring is running we use mpdtrace:

```
$ ${PREFIX}/bin/mpdtrace
master1
node01
node02
```

If the correct hostnames are listed the MPD ring is ready.

6.1.6. Setting up PVFS2 servers

Finally we setup the PVFS2 environment.

We need to make the following decisions:

- Which port shell the PVFS2 server use? ⇒${PVFS2_PORT}
- Which host shell become metadata server? (At the moment only one is supported.) ⇒${METASERVER}
- Which hosts shell become IO server? (The metadata server can also be IO server.) ⇒${IOSERVERS}

For our examples the following values are used:

```
$ PVFS2_PORT="21435"
$ METASERVER="master1"
$ IOSERVERS="node01,node02"
```

Now we can run pvfs2-genconfig to automatically generate the configuration:

```
$ cd ${WORKDIR}
$ ${PREFIX}/bin/pvfs2-genconfig --protocol tcp \
                                --tcpport ${PVFS2_PORT} \
                                --ioservers ${IOSERVERS} \
                                --metaservers ${METASERVER} \
                                fs.conf \
                                server.conf \
                                --storage /tmp/pvfs2-callid \
                                --logfile /tmp/pvfs2-callid.log
```

The question about verifying the server list can be answered with 'n'. The script generates one fs.conf and one server.conf-<hostname> for each server. These files contain the structure of the PVFS2 environment.

Next the file pvfs2tab has to be created for specifying a virtual mount point:

```
$ echo "tcp://localhost:${PVFS2_PORT}/pvfs2-fs /pvfs2 pvfs2 default,noauto 0 0"\
       > pvfs2tab
```

Before using it we have to create the storage space:

```
$ ${PREFIX}/bin/mpiexec -np 3 \
                        ${PREFIX}/sbin/pvfs2-server -d -f \
                                                    fs.conf \
                                                    server.conf
```

Nearly the same command again but without the `-f` option starts our PVFS2 servers. Please read ahead a few lines before running the command:

```
$ ${PREFIX}/bin/mpiexec -np 3 \
                        ${PREFIX}/sbin/pvfs2-server -d \
                                                    fs.conf \
                                                    server.conf
```

Running the PVFS2 servers in the background does not work smoothly with MPD. So after starting them we need to open a new console. Having xterm and, if using SSH, X forwarding enabled we can use the following trick instead:

```
$ xterm -e bash -c "${PREFIX}/bin/mpiexec -np 3 \
                                  ${PREFIX}/sbin/pvfs2-server -d \
                                                              fs.conf \
                                                              server.conf" &
```

Again we can test if everything is well, this time by using pvfs2-ping:

```
$ ${PREFIX}/bin/pvfs2-ping -m /pvfs2

(1) Parsing tab file...

(2) Initializing system interface...

(3) Initializing each file system found in tab file: pvfs2tab...

   /pvfs2: Ok

(4) Searching for /pvfs2 in pvfstab...

   [...]

(5) Verifying that all servers are responding...

   meta servers:
   tcp://master1:21435 Ok

   data servers:
   tcp://node01:21435 Ok
   tcp://node02:21435 Ok

(6) Verifying that fsid 1661493072 is acceptable to all servers...

   Ok; all servers understand fs_id 1661493072

(7) Verifying that root handle is owned by one server...

   Root handle: 1048576
   Ok; root handle is owned by exactly one server.

=============================================================

The PVFS filesystem at /pvfs2 appears to be correctly configured.
```

If the output looks somewhat like that, everything went right.

6.1.7. Installing slog2tools

The installation of the slog2tools package should make no problems at all. Just unpack it and run the following commands in the package directory:

```
$ ./configure --with-mpe2=${PREFIX} --with-java2=${JAVADIR}
$ make; make install
```

If there is a system-wide Java installation the script is able to detect it and this option can be omitted. Assure to use a Java version supporting at least API 1.5

6.2. Running a Small Demo Program

We now will run a small program to see the results of our work.

6.2.1. Writing the program

What is does. Each process of our parallel program will first write something to its segment of one big file, wait for all others to complete and then read something that another process has written. To disperse the events, there are a few rest periods between the operations. So we get a clearer visualisation.

The complete code can be found in appendix B on page 80.

6.2.2. Compiling the program

We now compile the program and link it to *libmpe* and *liblmpe* using mpicc:

```
$ ${PREFIX}/bin/mpicc callid-demo.c -o callid-demo -llmpe -lmpe
```

6.2.3. Running the program with event logging

We activate the servers event logging by setting the whole event-mask to one. In that way every event from now on will get logged by the event manager:

```
$ ${PREFIX}/bin/pvfs2-set-eventmask -m /pvfs2 -a 0xFFFF -o 0xFFFF
```

Now we run the program and afterwards stop the PVFS2 logging by resetting the event-mask to zero.

```
$ ${PREFIX}/bin/mpiexec -np 2 ./callid-demo -f pvfs2://pvfs2/test
$ ${PREFIX}/bin/pvfs2-set-eventmask -m /pvfs2 -a 0 -o 0
```

Due to the writelog patch the server writes the trace immediately when the event mask gets reset. For the moment the position for the log is hard coded to /tmp/pvfs2-server.clog2.

Since this is no good place to keep the trace, we move it to our working directory where the trace from the client side is already located:

```
$ mv /tmp/pvfs2-server.clog2 ${WORKDIR}
```

If we want to, we can quit the servers now:

```
$ ps x | grep mpiexec | grep pvfs2-server | cut -d 'p' -f 1 | xargs kill
$ sleep 3;
$ ${PREFIX}/bin/mpdallexit
```

6.2.4. Prepare the created logs for visualization

Since all our tools are working on SLOG-2 format, we first convert the traces:

```
$ cd ${WORKDIR}
$ ${PREFIX}/bin/clog2TOslog2 callid-demo.clog2
$ ${PREFIX}/bin/clog2TOslog2 pvfs2-server.clog2
```

The next step is to merge the logs using MergeSlog2. Unfortunately the auto time adjust function implemented until now is not sufficient for our needs. We have to calculate the time difference between the start points of both logs more accurately. The following commands are one alternative, it sets the beginning of the first *'MPI_File_write' event* to the same time as the beginning of the first *'Flow' event*.

```
$ T1=$(${PREFIX}/bin/slog2print callid-demo.slog2 \
          | egrep "Primitive.*MPI_File_write" \
          | line | sed -e "s/.*TimeBBox(//;s/,.*//" \
      )
$ T2=$(${PREFIX}/bin/slog2print pvfs2-server.slog2 \
          | egrep "Primitive.*Flow" \
          | line | sed -e "s/.*TimeBBox(//;s/,.*//" \
      )
$ ${PREFIX}/bin/MergeSlog2 -t1 $(perl -e "print $T2 - $T1;") \
                           -o merged.slog2 \
                           callid-demo.slog2 \
                           pvfs2-server.slog2
```

Now we are finally at the point to use Slog2ToArrowSlog2:

```
$ ${PREFIX}/bin/Slog2ToArrowSlog2 -o arrows.slog2 merged.slog2
```

6.2.5. Examination of the results

First we will take a look at the output of our new tool slog2stats. Remember that is has to run with the merged SLOG2 trace before the arrows are created to produce the correct output.

```
$ ${PREFIX}/bin/slog2print merged.slog2 | ${PREFIX}/bin/slog2stats.pl

Statistics:
Found 7 categories with CallID info key.
Counted 144 drawables.
Found 16 different CallID values.
On average there are 4 arrows per CallID having any.
```

Using jumpshot to have a look at the trace with arrows, we get some nicely colored pictures:

```
$ ${PREFIX}/bin/jumpshot arrows.slog2
```

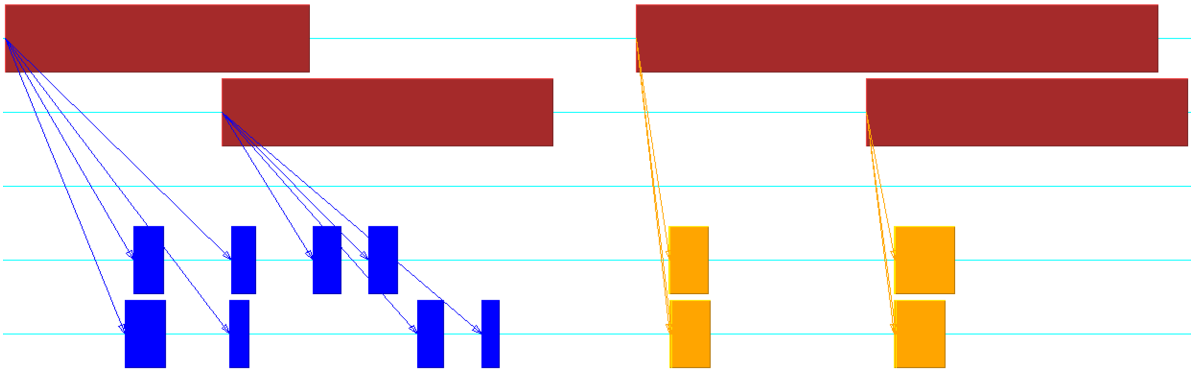
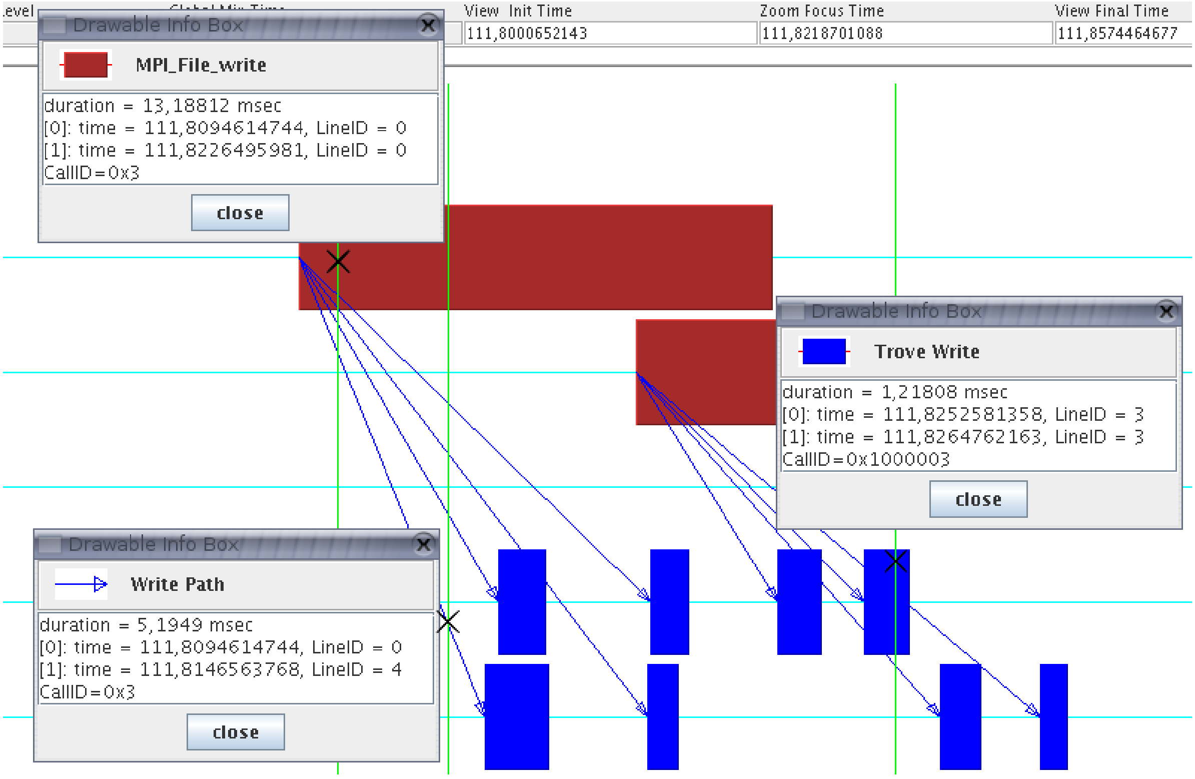


Figure 6.1.: Jumpshot-4 visualisation of arrows.slog2 (complete)

On figure 6.1 the viewport contains the complete interesting part. All objects are hidden[3] except *MPI_File_write*, *MPI_File_read*, *Trove Write*, *Trove Read* and the arrows *Write Path* and *Read Path*. Note the third line, there are no elements shown because this is the time line of master1, our metadata only server.

Figures 6.2 and 6.3 on the next page show the write and read part separately. As we can see the CallID is displayed on all objects making it easy to connect all elements belonging together. Also the separation of process and counter part is identifiable (see figure 5.3 on page 35 in section 5.3.1 "Client Side: From MPE2 to BMI). For example the CallID 0x1000003 is the third ID created in process 1 and 0x7 is the seventh ID created in process 0.

[3] Please refer to jumpshot manual [11] for handling.



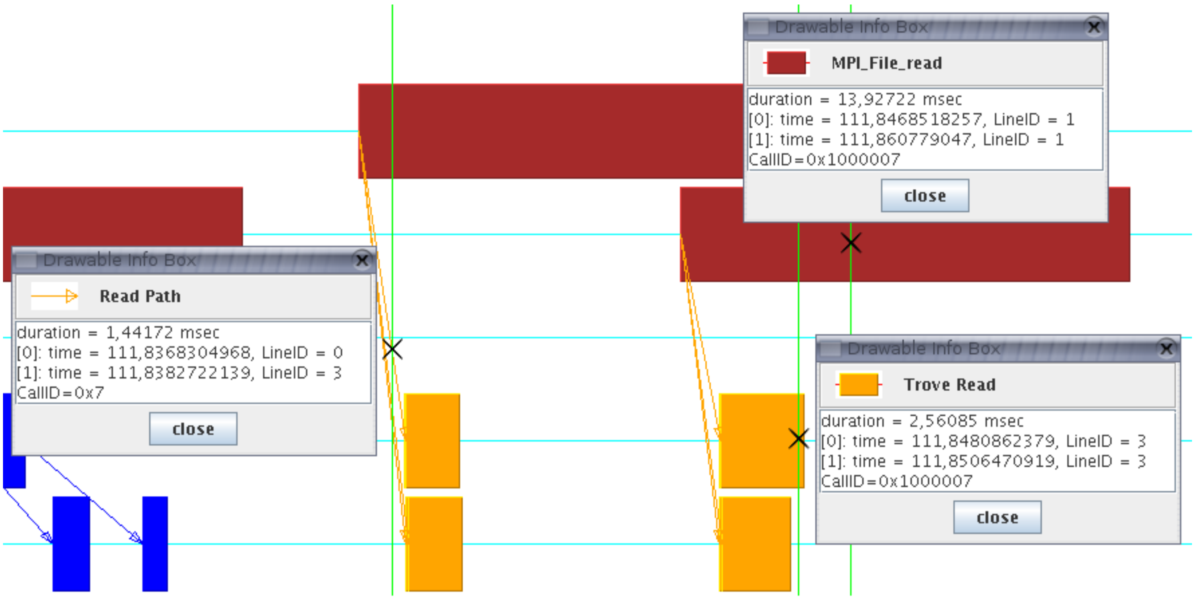
View Init Time: 111,8000652143
Zoom Focus Time: 111,8218701088
View Final Time: 111,8574464677

MPI_File_write
duration = 13,18812 msec
[0]: time = 111,8094614744, LineID = 0
[1]: time = 111,8226495981, LineID = 0
CallID=0x3

Trove Write
duration = 1,21808 msec
[0]: time = 111,8252581358, LineID = 3
[1]: time = 111,8264762163, LineID = 3
CallID=0x1000003

Write Path
duration = 5,1949 msec
[0]: time = 111,8094614744, LineID = 0
[1]: time = 111,8146563768, LineID = 4
CallID=0x3

Figure 6.2.: Jumpshot-4 visualisation of arrows.slog2 (write part)



MPI_File_read
duration = 13,92722 msec
[0]: time = 111,8468518257, LineID = 1
[1]: time = 111,860779047, LineID = 1
CallID=0x1000007

Read Path
duration = 1,44172 msec
[0]: time = 111,8368304968, LineID = 0
[1]: time = 111,8382722139, LineID = 3
CallID=0x7

Trove Read
duration = 2,56085 msec
[0]: time = 111,8480862379, LineID = 3
[1]: time = 111,8506470919, LineID = 3
CallID=0x1000007

Figure 6.3.: Jumpshot-4 visualisation of arrows.slog2 (read part)

6.3. Evaluation

In the example we could see that the goals described in chapter 3 "Specification of the Goals" on page 14 have been achieved.

We were able to make the desired connections and also made them visible in Jumpshot-4. Furthermore we could write a program to get statistical informations about the connections that can be easily enhanced with more functionality.

Nevertheless there is still a lot to be done. The collective call issue (section 4.4 "Limitations of Design" on page 23) is still open and also the reactivation of small I/O requests has to be handled (section 5.6 "Implementation Problems" on page 58). Further some analysis about the scalability issues have to be made (section 5.5 "Limitations of Implementation" on page 57). With these and other working approaches for the future will be discussed in short in chapter 8 "Future Work" on page 73 after the summary and conclutions of our work in the following chapter.

7. Summary and Conclusions

We are working with an environment for running parallel programs with medium to high amounts of I/O. MPICH2 is used for message passing and PVFS2 as parallel filesystem. At the beginning we were able to use MPE2 for creating traces from the calls of MPI-IO functions as well as from the PVFS2 Trove events representing operations on the real disks. However until this point it was not possible to say what MPI-IO function call is responsible for what disk operations. So our intention was to make these connections possible.

To reach this goal we changed the MPE2 logging system to create a unique identification number called CallID for every MPI-IO function call and write it into the client side trace together with the MPI-IO events. We pass this CallID down to the PVFS2 client library by using a global variable. This works since the client part of PVFS2 used with MPI-IO is not yet realized in multiple threads. From the PVFS2 client we send the CallID together with each request to the PVFS2 data servers. Now the data servers know about the MPI-IO call responsible for every single request. We modified the PVFS2 server code to take the CallID with the request handling through all its layers down to Trove and let the event manager write the ID with the Trove events into the server side trace.

After having the same CallID in the client side trace for an MPI-IO function call and in the server side trace for all resulting Trove operations, the basis for connections is created. With the tool MergeSlog2 in the slog2tools package we are able to merge the two traces together getting one trace with MPI-IO function calls and Trove events both with CallID to connect them.

We wrote a statistics tool to get an overview about connections in the merged trace. At the moment it calculates the average number of Trove events triggered by one MPI-IO function call. The tool is prepared to become easily extended with further statistical functions.

To make the connections between MPI-IO function calls and the resulting disk operations really visible, we created a new filter tool that searches the merged trace for all connections and creates an arrow event for each one. These arrows can now be visualized using Jumpshot.

We could definitely show that connecting the MPI-IO function calls and the triggered disk operations is possible and the results are in some simple test cased the expected. If this capability can be really useful for performance analysis and optimisation is still an open question. Perhaps the amount of information in the traces of large programs is so high, especially the number of connections, that further methods have to be developed to really benefit from the new connections.

8. Future Work

8.1. Work in Progress

8.1.1. Avoiding the visualisation problem with more than one client

Within the scope of his diploma thesis [18], Frank Panse is working on some changes of the PVFS2 logging. One of his goals is to deal with the problem occuring when requests from several clients produce overlapping events of the same type in the PVFS2 server trace as mentioned in section 4.4 on page 23. His approach it to only log single events instead of start-stop states and assign an ID to find the matching start and stop events afterwards. Using the CallID for this approach would not be a reasonable solution, since there could be more than one event (esp. Trove events) initiated by the same MPI-IO function call and so having the same CallID.

8.2. Necessary Enhancements

8.2.1. Take care of collective calls

We discussed this also in section 4.4 "Limitations of Design" on page 23. At the moment it is not sure what happens to the CallID when using MPI-IO collective calls.

These collective calls are MPI-IO calls that are made by multiple processes together to access data in a coordinated way. They are intended to enable the MPI-IO implementation to optimize the low level data access. For example assume that we have five processes that all need data stored in one file at almost the same point in time and these data are all in one contiguous place in the file. Using single calls (e.g. `MPI_File_read`) will result in five single file system access requests (forget about caches at the moment). In contrast using one collective call (e.g. each process calls `MPI_File_read_all`) gives the MPI-IO implementation the chance to do only one file system access request and then distribute the data to the processes.

This absolutely makes sense for environments where the file system with the file is physically connected to only one node in the cluster. But in our case the file system is spread over all nodes. Here requesting one contiguous part of a file will likely result in many disc operations (perhaps as many as single requests) and the coordination of collective calls produces additional network traffic. We see, handling collective calls in this canonical manner does not ensure a global optimization effect so perhaps with PVFS2, meaning with ROMIO using the ADIO for PVFS2, there is another mechanism used to process collective calls, perhaps just a mapping to non-collective calls.

This aspect should be addressed by first investigating the exact handling of collective calls in PVFS2 and if necessary by developing a method to assure the correct and reasonable handling of CallIDs.

8.2.2. Enable small I/O

As described in section 5.6 "Implementation Problems" on page 58 with PVFS2 release 1.4.0 a new request type was introduces. It is called *small I/O* as is intended to speed up very small I/O requests by bypassing the Flow creation in the PVFS2 server. Without our deactivation such a small I/O request is created for each I/O request with data smaller that 16KB (if using TCP) instead. This kind of request becomes handled by another state machine at the PVFS2 server, taking a shorter path through the server layers by not creating a Flow but passing the data directly to or from Trove. Here is not the place to start a deeper discussion of this, we will only give some hints to future developers working on this issue.

Figure 8.1 on the next page illustrate the path a small I/O request takes through the PVFS2 server in a similar diagram as for normal I/O in figure 5.5 on page 40.

Comparing the figures 8.1 on the next page and 5.5 on page 40 we can see again the problem with the void pointers referencing different structures. As far as I know at the moment there is no possibility to recognize the type of the request and thus the path it has taken in Trove.

8.3. Ideas for technical enhancements

8.3.1. Make PVFS2 Code even more flexible

I suggest to add an extra field to the parameter list of some of the functions in the whole PVFS2 system. All those functions should become modified through which someone could want to pass information. This field should be a void pointer or better the pointer to a linked list of key-value pairs. For the approach with the prepared linked list the access functionality should become prepared, too.

The same idea refers to the user pointers. At the moment they are almost useless because no way is given to find out what kind of data the pointer references. Here I suggest not to use a void pointer but a pointer to a user structure. This structure should contain a void pointer to reference the users data object as already done by the user pointer now. In addition it should hold an enumeration field representing the type of the data the void pointer references. With

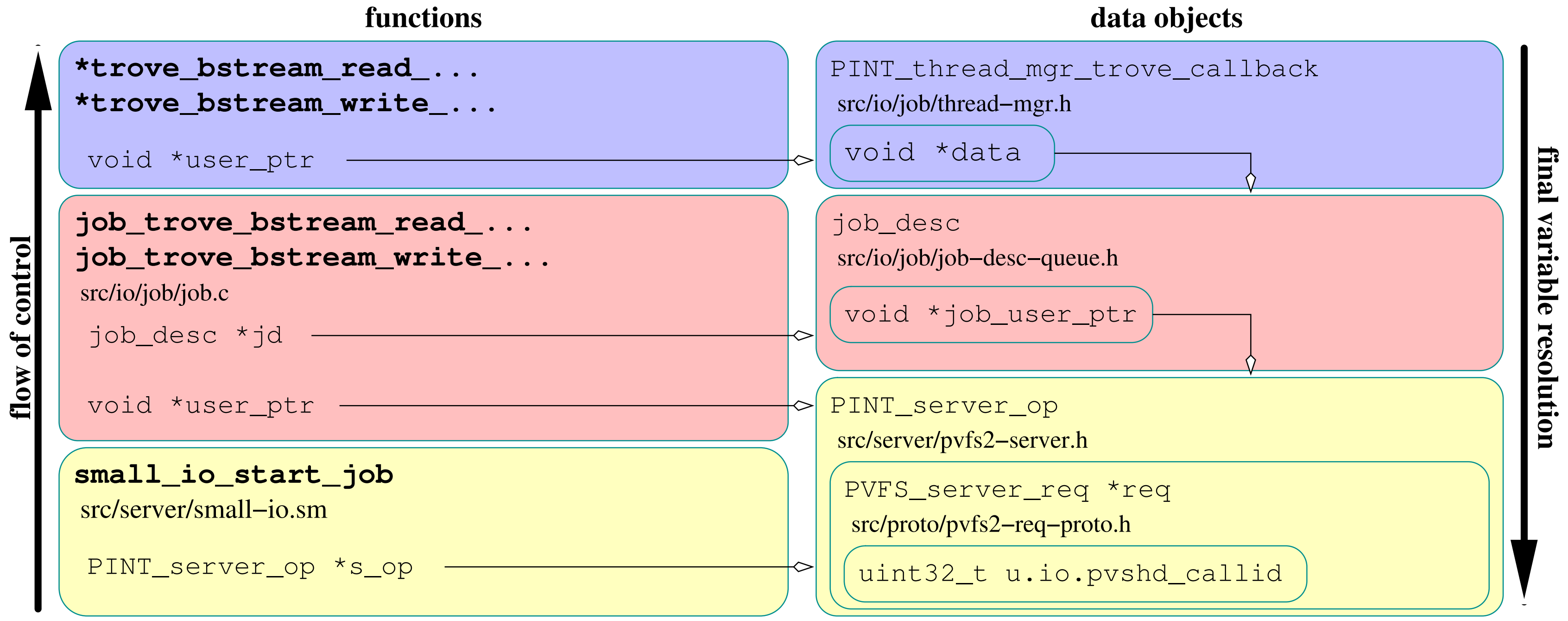


Figure 8.1.: Important functions on a small I/O request's path through a PVFS2 server and accessibility of the CallID.

this solution it would be easy to add additional user types and the used layer has the chance to know who the user is.

8.3.2. Better inclusion of the changes

This is related to the point above. At the moment all my changes are more like a functional hack than a really good enhancement of the existing code. I already described the reasons for that at the corresponding places in this document. If these reasons are eliminated a better sort if implementation of most of the new functionality should be made.

A. List of MPI-IO Functions

MPI_File_open
open a file
MPI_File_close
close a file
MPI_File_delete
delete a file
MPI_File_set_size
resize a file
MPI_File_preallocate
preallocate storage space for a file
MPI_File_get_size
get the actual size of a file
MPI_File_get_group
get the group of the communicator used to open a file
MPI_File_get_amode
get the access mode of a file

MPI_File_set_info
set new values for the hints of a file
MPI_File_get_info
get the hints of a file

MPI_File_set_view
change the process's view of the data in a file
MPI_File_get_view
get the process's view of the data in a file

`MPI_File_read_at`
: read data from a file using an explicit offset

`MPI_File_read_at_all`
: *collective* read data from a file using an explicit offset

`MPI_File_write_at`
: write data to a file using an explicit offset

`MPI_File_write_at_all`
: *collective* write data to a file using an explicit offset

`MPI_File_iread_at`
: read data from a file *non-blocking* using an explicit offset

`MPI_File_iwrite_at`
: write data to a file *non-blocking* using an explicit offset

`MPI_File_read`
: read data from a file using an individual file pointer

`MPI_File_read_all`
: *collective* read data from a file using an individual file pointer

`MPI_File_write`
: write data to a file using an individual file pointer

`MPI_File_write_all`
: *collective* write data to a file using an individual file pointer

`MPI_File_iread`
: read data from a file *non-blocking* using an individual file pointer

`MPI_File_iwrite`
: write data to a file *non-blocking* using an individual file pointer

`MPI_File_seek`
: set the individual file pointer to a new position using an offset

`MPI_File_get_position`
: get the current position of the individual file pointer

`MPI_File_get_byte_offset`
: convert a view-relative offset into an absolute byte position

`MPI_File_read_shared`
: read data from a file using the shared file pointer

`MPI_File_write_shared`
: write data to a file using the shared file pointer

`MPI_File_iread_shared`
: read data from a file *non-blocking* using the shared file pointer

`MPI_File_iwrite_shared`
: write data to a file *non-blocking* using the shared file pointer

`MPI_File_read_ordered`
: *collective* read data from a file using the shared file pointer

`MPI_File_write_ordered`
: *collective* write data to a file using the shared file pointer

`MPI_File_seek_shared`
: set the shared file pointer to a new position using an offset

`MPI_File_get_position_shared`
: get the current position of the shared file pointer

MPI_File_read_at_all_begin
begin a split collective read for a file using an explicit offset

MPI_File_read_at_all_end
finish a split collective read for a file using an explicit offset

MPI_File_write_at_all_begin
begin a split collective write for a file using an explicit offset

MPI_File_write_at_all_end
finish a split collective write for a file using an explicit offset

MPI_File_read_all_begin
begin a split collective read for a file using an individual file pointer

MPI_File_read_all_end
finish a split collective read for a file using an individual file pointer

MPI_File_write_all_begin
begin a split collective write for a file using an individual file pointer

MPI_File_write_all_end
finish a split collective write for a file using an individual file pointer

MPI_File_read_ordered_begin
begin a split collective read for a file using the shared file pointer

MPI_File_read_ordered_end
finish a split collective read for a file using the shared file pointer

MPI_File_write_ordered_begin
begin a split collective write for a file using the shared file pointer

MPI_File_write_ordered_end
finish a split collective write for a file using the shared file pointer

MPI_File_get_type_extent
get the extent of datatype in a file

MPI_Register_datarep
registers a user-defined data representation

MPI_File_set_atomicity
collective set consistency semantics for a collectively opened file

MPI_File_get_atomicity
get the current consistency semantics for collectively opened file

MPI_File_sync
collective synchronize the file on the disk

B. Code for callid-demo (callid-demo.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <mpi.h>
#include <getopt.h>
#include <sys/select.h>


int main(int argc, char **argv)
{
  MPI_Status status;
  int namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int myrank;
  int nranks;

  char opt_file[256] = "callid.demo"; // file to use
  MPI_File fh;
  int64_t posptr = 0;

  char *buffer;
  int readdata;

  int bufsize = 1*1024*1024;    // size of buffer in MPI_CHAR
  int blocksize = bufsize;      // blocksize for each rank in MPI_CHAR

  struct timespec rel_wait, abs_wait;

  // startup MPI and determine the rank of this process
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nranks);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Get_processor_name(processor_name, &namelen);

  printf("Process␣%d␣of␣%d␣is␣on␣%s\n", myrank, nranks, processor_name);

  // parse arguments
  int opt;

  while ((opt = getopt(argc, argv, "b:f:h")) != EOF)
  {
    switch (opt)
    {
```

```
    case ’f’: // filename
      strncpy(opt_file, optarg, 255);
      break;
    case ’h’:
      if (myrank == 0)
            printf("Usage:␣callid-demo␣[-f␣<FILENAME>]␣[-h]\n");
      exit(0);
    default:
      break;
  }
}

// setup wait times
rel_wait.tv_sec = 0;
rel_wait.tv_nsec = myrank*(10000000);

abs_wait.tv_sec = 0;
abs_wait.tv_nsec = 2000000;

// setup buffer
buffer = malloc((size_t) bufsize*sizeof(MPI_CHAR));

// open opt_file for writing
MPI_File_open(MPI_COMM_WORLD, opt_file,
           MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &fh);

// set position pointer to appropriate position for this rank
posptr = myrank*blocksize;
MPI_File_seek(fh, posptr, MPI_SEEK_SET);

// synchronize
MPI_Barrier(MPI_COMM_WORLD);

// wait for rank*10 ms not to produce overlapping troves in PVFS2 server
pselect( 0, NULL, NULL, NULL, &rel_wait, NULL );

// write data to file system
MPI_File_write(fh, buffer, blocksize, MPI_CHAR, &status);

// close file
MPI_File_close(&fh);

// synchronize
MPI_Barrier(MPI_COMM_WORLD);

// wait 2 ms before proceeding
pselect( 0, NULL, NULL, NULL, &abs_wait, NULL );

// open opt_file for reading
MPI_File_open(MPI_COMM_WORLD, opt_file,
           MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);

// set position pointer to where rank+1 has written (rotating)
```

```
  posptr = (( myrank + 1 ) % nranks ) * blocksize;
  MPI_File_seek(fh, posptr, MPI_SEEK_SET);

  // synchronize
  MPI_Barrier(MPI_COMM_WORLD);

  // wait for rank*10 ms not to produce overlapping troves in PVFS2 server
  pselect( 0, NULL, NULL, NULL, &rel_wait, NULL );

  // read data from file
  MPI_File_read(fh, buffer, blocksize, MPI_CHAR, &status);

  // close opt_file
  MPI_File_close(&fh);

  // synchronize
  MPI_Barrier(MPI_COMM_WORLD);

  free(buffer);
  MPI_Finalize();
  return(0);
}
```

Listing B.1: Small demo program used in example (callid-demo.c)

Listings

List of Figures

List of Abbreviations

ADIO	Abstract Device for Input/Output
API	Application Programming Interface
BMI	Buffered Message Interface
CallID	Call Identification
I/O	Input/Output
IDE	Integrated Development Environment
LAN	Local Area Network
MPE	MPI Parallel Environment
MPI	Message Passing Interface
MPI-IO	Message Passing Interface - Input/Output
MPICH2	MPI Chameleon 2
PVFS2	Second Parallel Virtual Filesystem
PVSHD	Parallele und Verteilte Systeme, Heidelberg
ROMIO	Who nows? No Abbreviation!
SDK	Standard Development Kit
VFS	Virtual Filesystem Switch
acache	address cache
ncache	name cache

References

[1] PVFS2 Development Team. *Parallel Virtual File System, Version 2*. Internet.
http://www.pvfs.org/pvfs2/index.html

[2] Message Passing Interface Forum (MPIF). *MPI: A Message-Passing Interface Standard, Version 1.1*. Tech. Rep., University of Tennessee, Knoxville, June 1995.
http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[3] Message Passing Interface Forum (MPIF). *MPI-2: Extensions to the Message-Passing Interface*. Tech. Rep., University of Tennessee, Knoxville, 1997.
http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html

[4] MPICH2 Development Team. *MPICH2 Home Page*. Internet.
http://www-unix.mcs.anl.gov/mpi/mpich2/index.htm

[5] William Gropp, Ewing Lusk et al.. *MPICH2 User's Guide - Version 1.0.3*. Argonne National Laboratory, November 2005.
http://www-unix.mcs.anl.gov/mpi/mpich2/downloads/mpich2-doc-user.pdf

[6] William Gropp, Ewing Lusk et al.. *MPICH2 Installer's Guide - Version 1.0.3*. Argonne National Laboratory, November 2005.
http://www-unix.mcs.anl.gov/mpi/mpich2/downloads/mpich2-doc-install.pdf

[7] ROMIO Development Team. *ROMIO: A High-Performance, Portable MPI-IO Implementation*. Internet.
http://www-unix.mcs.anl.gov/romio/index.html

[8] Rajeev Thakur, Ewing Lusk and William Gropp. *Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation*. Technical Memorandum 234, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439, May 2004. ANL/MCS-TM-234.
http://www.mcs.anl.gov/~thakur/papers/users-guide.ps

[9] Laboratory for Advanced Numerical Software (LANS). *Performance Visualization for Parallel Programs*. Internet. This is the website for MPE2, SLOG-2 and Jumpshot-4.
http://www-unix.mcs.anl.gov/perfvis/index.htm

[10] Anthony Chan, William Gropp and Ewing Lusk. *Scalable Log Files for Parallel Program Trace Data*. Internet, 2000.
ftp://ftp.mcs.anl.gov/pub/mpi/slog2/slog2-draft.pdf

[11] Anthony Chan, David Ashton, Rusty Lusk and William Gropp. *Jumpshot-4 Users Guide*, December 2004.
ftp://ftp.mcs.anl.gov/pub/mpi/slog2/js4-usersguide.pdf

[12] PVFS2 Development Team. *Parallel Virtual File System, Version 2*. Argonne National Laboratory, September 2003.
http://www.pvfs.org/pvfs2/pvfs2-guide.html

[13] Rob Lathan, Neill Miller, Robert Ross and Phil Carns. *A Next-Generation Parallel File System for Linux Clusters*. Linux World Magazine, (pp. 56–59), January 2004. An introduction to the second Parallel Virtual File System.
http://www.pvfs.org/pvfs2/files/linuxworld-JAN2004-PVFS2.ps

[14] Julian Kunkel, Thomas Ludwig and Hipolito Vasquez. *Weit verteilt - Dateisystem für parallele Systeme: PVFS, Version 2*. iX - Magazin für professionelle Informationstechnik, 6/04: pp. 110–113, 2004.

[15] Julian Martin Kunkel. *Performance Analysis of the PVFS2 Persistency Layer*. Bachelor's thesis, Rubrecht-Karls-Universität Heidelberg, Institute of Computer Science, Research Group Parallel and Distributed Systems, February 2006.

[16] Withanage Don Samantha Dulip. *Performance Visualization for the PVFS2 Environment*. Bachelor's thesis, Rubrecht-Karls-Universität Heidelberg, Institute of Computer Science, Research Group Parallel and Distributed Systems, November 2005.

[17] Free Software Foundation, Inc. *sed - manual page for sed version 4.1.2*, 2004.

[18] Frank Panse. *Extended Tracing Capabilities and Optimization of the PVFS2 Event Logging Management*. Diploma thesis, Rubrecht-Karls-Universität Heidelberg, Institute of Computer Science, Research Group Parallel and Distributed Systems, 2006. This thesis is not finished yet.