Date: 2013-09-26	**High Level Design – POSIX Function Shipping** **FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O** **MILESTONE: 5.1 Initial POSIX Function Shipping Demonstration**

LLNS Subcontract No.	B599860
Subcontractor Name	Intel Federal LLC
Subcontractor Address	2200 Mission College Blvd. Santa Clara, CA 95052

Table of Contents

Revision History

Date	Revision	Author
2013-09-26	1.0	Jerome Soumagne, Quincey Koziol, The HDF Group

Introduction

High performance I/O on exascale systems is not expected to be feasible without exporting the I/O API from I/O nodes onto the compute nodes. One solution to address this problem is to use a method called function shipping, also commonly called remote procedure call (RPC). Making use of this method, I/O calls from the compute nodes are locally encoded and sent through the network to the I/O nodes where they in turn get decoded and executed—with the operation's result being sent back to the issuing node. While the main objective of the FastForward project is to realize I/O through the IOD and DAOS APIs, most of the existing applications that make use of POSIX I/O must be able to run in this environment and be able to remotely store and access data. This document describes the implementation of the framework that allows POSIX I/O calls to be transparently redirected to a remote function-shipping server.

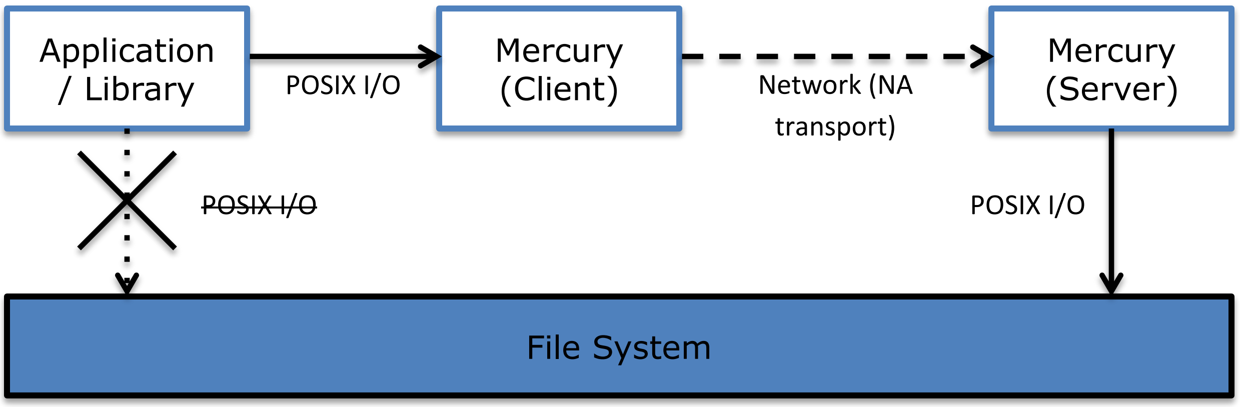
Changes from Solution Architecture

None.

Specification

Overview

The framework is built on top of the existing function shipping framework, referred to as *Mercury*. It is designed so that existing applications or libraries do not require any code modifications to redirect POSIX I/O to a remote mercury server. As shown in the figure below, applications and libraries can be dynamically linked to a mercury client library that intercepts POSIX I/O calls and ships them through Mercury's network abstraction layer to a remote server; the server can in turn execute the POSIX I/O calls on its local file system.



POSIX I/O calls are shipped using transfer mechanisms defined by Mercury: metadata transfer (small data arguments) and bulk data transfer (in case of large data arguments or read/write calls).

POSIX I/O routines
POSIX I/O routines that are supported for this milestone are defined below:

access	fdatasync	mkdir	truncate
chdir	fpathconf	mkfifo	umask
chmod	fstat	mknod	unlink
chown	fsync	open	write
creat	ftruncate	pathconf	
close	getcwd	read	+LFS versions:
dup	lchown	readlink	creat64
dup2	link	rmdir	ftruncate64
fchdir	lockf	stat	lseek64
fchmod	lseek	symlink	open64
fchown	lstat	sync	etc.

Since applications and libraries can make use of large file support (LFS) offered by the Linux kernel on file systems that support it and hence use 64-bit file offsets, support for 64 versions (e.g., open64, etc) of POSIX I/O routines has also been added.

API and Protocol Additions and Changes
Mercury itself has not been subject to any major modification and all the POSIX I/O calls forwarded use the mechanisms and API that have been defined in the previous milestones. Support for POSIX I/O is added by building a new package called *Mercury POSIX*, which defines both a lightweight library and a server, built on top of the existing Mercury API. Following sections describe the internal implementation of Mercury POSIX.

POSIX I/O Support
A large set of POSIX I/O routines is composed of relatively similar argument types, and therefore the source code that is used to ship these routines and execute them onto the remote server follows the same pattern. To improve maintainability and easily add support for POSIX I/O calls, we make use of the Mercury Boost preprocessor macros that have been

defined in the previous milestone to build a new set of macros on top of them. The two main macros that are internally used in this library to build on top of the Mercury interface are the following:

```
/* Non-bulk routines */
MERCURY_POSIX_GEN_STUB(
 func_name, /* function name */
 ret_type,  /* return type */
 in_types,  /* sequence of input types */
 out_types  /* sequence of output types */
)


/* Bulk routines */
MERCURY_POSIX_GEN_BULK_STUB(
 func_name, /* function name */
      ret_type,  /* return type */
      in_types,  /* sequence of input types */
      out_types, /* sequence of output types */
      bulk_read  /* 1/0 if reading/writing bulk data */
)
```

Calling these macros generates: an input structure that can contain input parameters; an encoding/decoding processor for the input parameters; an output structure that can contain output parameters; an encoding/decoding processor for the output parameters (and return value); a client stub routine that follows the POSIX I/O routine prototype and forwards the call to a predefined server; a server stub routine that executes the POSIX I/O call when the server receives the corresponding request.

For instance, adding support for the lseek routine which has the following prototype:

```
off_t lseek(int fildes, off_t offset, int whence);
```

Can be done by calling:

```
MERCURY_POSIX_GEN_STUB(
 lseek,
 hg_off_t,
 (hg_int32_t)(hg_off_t)(hg_int32_t),
)
```

where hg_off_t is an internally defined type that depends on the LFS support option and maps to off_t or off64_t. Note that the order of the types must match the order defined in the actual prototype for the function to be correctly called on the server.

The same happens for calls that make use of bulk data. In the case of the write call:

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Adding support for it can be done by calling the following macro:

```
MERCURY_POSIX_GEN_BULK_STUB(
 write,
 hg_ssize_t,
 (int32_t), ,
 MERCURY_GEN_TRUE
)
```

Bulk arguments (`void *buf` and `size_t nbyte`) are automatically included within this macro, as they are internally replaced by an `hg_bulk_t` bulk handle. When the client and server stubs are generated, `void *` and `size_t` parameters are automatically appended to the list of parameters. This may of course not completely fit to the actual prototype in some cases but a direct mapping of function parameters can be easily be done by calling an intermediate routine or by defining another macro. It is worth noting that, in general, automatic generation is limited for bulk data calls, in the sense that it may not be possible to automatically generate everything from a macro, in which case client and server stubs need to be manually written (although input/output structure as well as encoding/decoding processor can still be automatically generated). Fortunately this only applies to a very limited number of calls.

Integration with HDF5 and other libraries

As already mentioned, once the Mercury POSIX library is installed on a system, libraries and existing tools that are dynamically linked can forward POSIX I/O to a remote server without any code modification. However, environment variables need to be set in order to provide the client with the required connection information:

- MERCURY_NA_PLUGIN: Underlying network transport method used to forward calls to a remote server. (e.g., "bmi")
- MERCURY_PORT_NAME: Port name information (IP/port) specific to the network transport chosen – used to establish a connection with a remote server. (e.g., "tcp://72.36.68.242:22222")

Additionally, the `LD_PRELOAD` environment variable must be set to the location of the Mercury POSIX shared library. This allows the POSIX routines defined in the Mercury POSIX library to be called instead of the ones that are defined in the standard library.

For instance, I/O of applications that use HDF5 and the sec2 driver (which uses POSIX I/O) can be forwarded to a remote server without any modification of the original library:

```
client$ h5dump -H coord.h5
HDF5 "coord.h5" {
GROUP "/" {
   DATASET "multiple_ends_dset" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 4, 5,
3, 4, 2, 3, 6, 2 ) / ( 4, 5, 3, 4,
2, 3, 6, 2 ) }
   }
   DATASET
"multiple_ends_dset_chunked" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 4, 5,
3, 4, 2, 3, 6, 2 ) / ( 4, 5, 3, 4,
2, 3, 6, 2 ) }
   }
```

```
server$ mercury_posix_server bmi
Waiting for client...
Thu, 19 Sep 13 17:31:00 CDT:
Executing open64
Thu, 19 Sep 13 17:31:00 CDT:
Executing __fxstat64
Thu, 19 Sep 13 17:31:00 CDT:
Executing lseek64
Thu, 19 Sep 13 17:31:00 CDT:
Executing hg_posix_read
Thu, 19 Sep 13 17:31:00 CDT:
Executing lseek64
Thu, 19 Sep 13 17:31:00 CDT:
Executing hg_posix_read
Thu, 19 Sep 13 17:31:00 CDT:
Executing hg_posix_read
```

Open Issues

The library defined provides a simple way of forwarding POSIX I/O to a remote server and supports a large number of POSIX I/O calls already. It is, however, not guaranteed that all the existing applications can use it at this time, especially if they make use of a POSIX call that is not supported yet (which is also one of the reasons why the library has been designed so that it can be easily extended). For instance the Lustre POSIX test suite makes use of `fdopen` to get a file stream from an exisiting file descriptor, this is currently an issue as file streams are currently not supported.

Risks & Unknowns

Libraries and applications that make use of static linking need to explicitly include and link to the Mercury POSIX library. Libraries and applications that make use of dynamic linking can have I/Os dynamically redirected.