



UnifyFS Documentation

Release 0.9.1

Kathryn Mohror, Adam Moody, Oral Sarp, Feiyi Wang, Hyogi Sim,

Sep 13, 2021

1	Overview	1
1.1	High Level Design	2
2	Definitions	3
2.1	Job	3
2.2	Run or Job Step	3
3	Assumptions and Semantics	5
3.1	System Requirements	5
3.2	Application Behavior	5
3.3	Consistency Model	6
3.4	File System Behavior	7
4	Build UnifyFS	9
4.1	Build UnifyFS and its dependencies with Spack	9
4.2	Build dependencies with Spack, build UnifyFS with autotools	10
4.3	Build dependencies with bootstrap and build UnifyFS with autotools	11
4.4	Configure Options	12
5	Integrate the UnifyFS API	15
5.1	Include the UnifyFS header	15
5.2	Mounting	15
5.3	Unmounting	16
6	Link with the UnifyFS library	17
6.1	Static link	17
6.2	Dynamic link	17
7	UnifyFS Configuration	19
7.1	unifyfs.conf	19
7.2	Environment Variables	21
7.3	Command Line Options	21
8	Run UnifyFS	23
8.1	Start UnifyFS	23
8.2	Stop UnifyFS	24
8.3	Resource Manager Job Integration	24

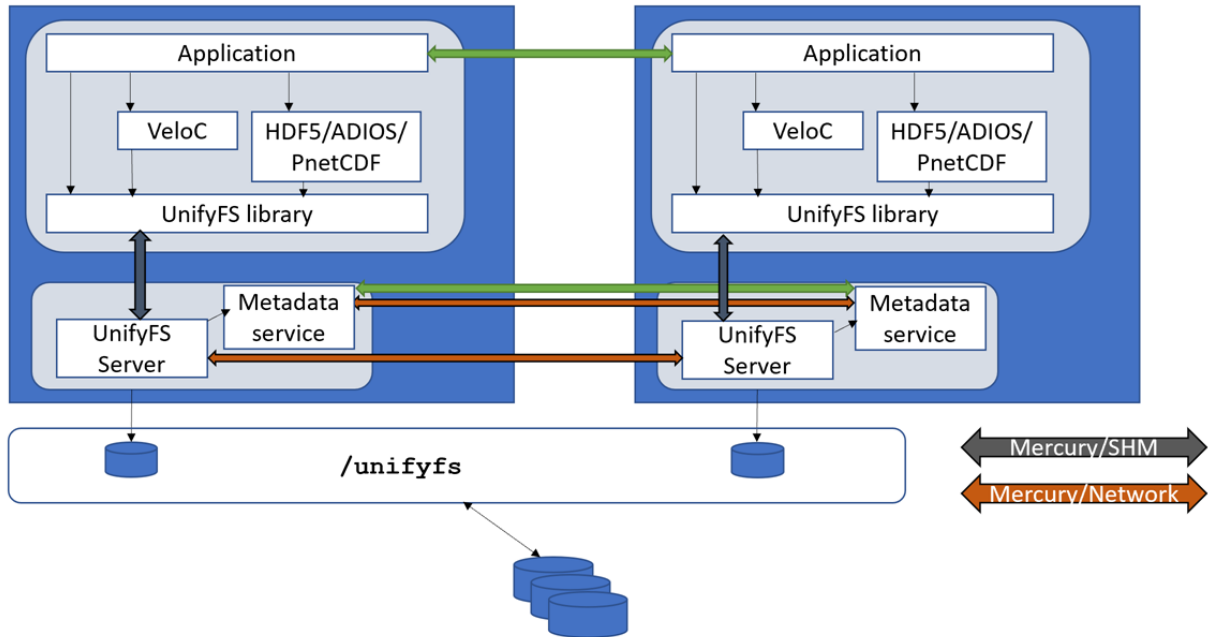
8.4	Transferring Data In and Out of UnifyFS	25
9	Examples	29
9.1	Examples Locations	29
9.2	Running the Examples	30
9.3	Producer-Consumer Workflow	31
10	UnifyFS API for I/O Middleware	33
10.1	Library API Purpose	33
10.2	Library API Concepts	33
10.3	Library API Types	34
10.4	Example Library API Usage	35
11	Ways to Contribute	39
11.1	Getting Started	39
11.2	Reporting Bugs	39
11.3	Suggesting Enhancements	40
11.4	Pull Requests	40
11.5	Testing	40
11.6	Documentation	41
12	Style Guides	43
12.1	Coding Conventions	43
12.2	Commit Message Format	44
13	Testing Guide	45
13.1	Unit Tests	45
13.2	Integration Tests	51
14	Wrapper Guide	61
14.1	unifyfs_check_fns Tool	61
14.2	Building the GOTCHA List	62
14.3	Commands to Build Files	62
15	Adding RPC Functions With Margo Library	63
15.1	Common	63
15.2	Server	63
15.3	Client	64
16	UnifyFS Dependencies	65
16.1	Required	65
16.2	Optional	65
17	Indices and tables	67

Overview

UnifyFS is a user-level file system under active development that supports shared file I/O over distributed storage on HPC systems, e.g., node-local burst buffers. With UnifyFS, applications can write to fast, scalable, node-local burst buffers as easily as they do to the parallel file system. UnifyFS is designed to support common I/O workloads such as checkpoint/restart and other bulk-synchronous I/O workloads typically performed by HPC applications.

Because the UnifyFS file system is implemented at user-level, the file system is visible only to applications linked with the UnifyFS client library. A consequence of this is that traditional file system tools (`ls`, `cd`, etc.) installed by system administrators cannot act on files in a UnifyFS file system because they are not linked against the UnifyFS client library. The lifetime of a UnifyFS file system is the duration of the execution of the UnifyFS server processes, which is typically for the duration of an HPC job. When the servers exit, the UnifyFS file system terminates. Users must copy files that need to be persisted beyond the lifetime of the job from UnifyFS to a permanent file system. UnifyFS provides an API and a utility to perform these copies.

1.1 High Level Design



This section provides a high level design of UnifyFS. UnifyFS presents a shared namespace (e.g., `/unifyfs` as a mount point) to all compute nodes in a job allocation. There are two main components of UnifyFS: the UnifyFS library and the UnifyFS server.

The UnifyFS library is linked into the user application. The library intercepts I/O calls from the application and sends I/O requests for UnifyFS files on to the UnifyFS server. The library forwards I/O requests for all other files on to the system. The UnifyFS library uses ECP **GOTCHA** as its primary mechanism to intercept I/O calls. The user application is linked with the UnifyFS client library and perhaps a high-level I/O library, like HDF5, ADIOS, or PnetCDF.

A UnifyFS server process runs on each compute node in the job allocation. The UnifyFS server handles the I/O requests from the UnifyFS library. The UnifyFS server uses ECP **Mochi** to communicate with user application processes and server processes on other nodes.

This section defines some terms used throughout the document.

2.1 Job

A set of commands that is issued to the resource manager and is allocated a set of nodes for some duration

2.2 Run or Job Step

A single application launch of a group of one or more application processes issued within a job

Assumptions and Semantics

In this section, we provide assumptions we make about the behavior of applications that use UnifyFS and about the file system semantics of UnifyFS.

3.1 System Requirements

The system requirements to run UnifyFS are:

- Compute nodes must be equipped with local storage device(s) that UnifyFS can use for storing file data, e.g., SSD or RAM.
- The system must support the ability for UnifyFS user-level server processes to run concurrently with user application processes on compute nodes.

3.2 Application Behavior

UnifyFS is specifically designed to support the bulk synchronous I/O pattern that is typical in HPC applications, e.g., checkpoint/restart or output dumps. In bulk synchronous I/O, I/O operations occur in separate write and read phases, and files are not read and written simultaneously. For example, files are written during checkpointing (a write phase) and read during recovery/restart (a read phase). Additionally, parallel writes and reads to shared files occur systematically, where processes access computable, regular offsets of files, e.g., in strided or segmented access patterns, with ordering of potential conflicting updates enforced by inter-process communication. This behavior is in contrast to other I/O patterns that may perform random, small writes and reads or overlapping writes without synchronization.

UnifyFS offers the best performance for applications that exhibit the bulk synchronous I/O pattern. While UnifyFS does support deviations from this pattern, the performance might be slower and the user may have to take additional steps to ensure correct execution of the application with UnifyFS. For more information on this topic, refer to the section on [:ref:'commit consistency semantics in UnifyFS <commit_consistency_label>'_.](#)

3.3 Consistency Model

The UnifyFS file system does not support strict POSIX consistency semantics. (Please see [Chen et al., HPDC 2021](#) for more details on different file system consistency semantics models.) Instead, UnifyFS supports two different consistency models: *commit consistency semantics* when a file is actively being modified; and *lamination semantics* when the file is no longer being modified by the application. These two consistency models provide opportunities for UnifyFS to provide better performance for the I/O operations of HPC applications.

3.3.1 Commit Consistency Semantics in UnifyFS

Commit consistency semantics require explicit “commit” operations to be performed before updates to a file are globally visible. We chose commit consistency semantics for UnifyFS because it is sufficient for correct execution of typical HPC applications that adhere to the bulk synchronous I/O pattern, and it enables UnifyFS to provide better performance than with strict POSIX semantics. For example, because we assume that applications using UnifyFS will not execute concurrent modifications to the same file offset, UnifyFS does not have to employ locking to ensure sequential access to file regions. This assumption allows us to cache file modifications locally which greatly improves the write performance of UnifyFS.

The use of synchronization operations are required for applications that exhibit I/O accesses that deviate from the bulk synchronous I/O pattern. There are two types of synchronization that are required for correct execution of parallel I/O on UnifyFS: *local synchronization* and *inter-process synchronization*. Here, *local synchronization* refers to synchronization operations performed locally by a process to ensure that its updates to a file are visible to other processes. For example, a process may update a region of a file and then execute `fflush()` so that a different process can read the updated file contents. *Inter-process synchronization* refers to synchronization operations that are performed to enforce ordering of conflicting I/O operations from multiple processes. These inter-process synchronizations occur outside of normal file I/O operations and typically involve inter-process communication, e.g., with MPI. For example, if two processes need to update the same file region and it is important to the outcome of the program that the updates occur in a particular order, then the program needs to enforce this ordering with an operation like an `MPI_Barrier()` to be sure that the first process has completed its updates before the next process begins its updates.

There are several methods by which applications can adhere to the synchronization requirements of UnifyFS.

- Using MPI-IO. The (MPI-IO) interface requirements are a good match for the consistency model of UnifyFS. Specifically, the MPI-IO interface requires explicit synchronization in order for updates made by processes to be globally visible. If an application utilizes the MPI-IO interface correctly, it will adhere to the requirements of UnifyFS.
- Using (HDF5) and other parallel I/O libraries. Most parallel I/O libraries hide the synchronization requirements of file systems from their users. For example, HDF5 implements the synchronization required by the MPI-IO interface so users of HDF5 do not need to perform any synchronization operations explicitly in their codes.
- With explicit synchronization. If an application does not use a compliant parallel I/O library or if the developer wishes to perform explicit synchronization, local synchronization can be achieved through adding explicit “flush” operations with calls to `fflush()`, `close()`, or `fsync()` in the application source code, or by supplying the `client.write_sync` configuration parameter to UnifyFS on startup, which will cause an implicit “flush” operation after every write (note: use of the `client.write_sync` mode can significantly slow down write performance). In this case, inter-process synchronization is still required for applications that perform conflicting updates to files.

During a write phase, a process can deviate from the bulk synchronous I/O pattern and read any byte in a file, including remote data that has been written by processes executing on remote compute nodes in the job. However, the performance will differ based on which process wrote the data that is being read:

- If the bytes being read were written by the same process that is reading the bytes, UnifyFS offers the fastest performance and no synchronization operations are needed. This kind of access is typical in some I/O libraries, e.g., HDF5, where file metadata may be updated and read by the same process. (Note: to obtain the performance benefit for this case, one must set the `client.local_extents` configuration parameter.)

- If the bytes being read were written by a process executing on the same compute node as the reading process, UnifyFS can offer slightly slower performance than the first case and the application must introduce synchronization operations to ensure that the most recent data is read.
- If the bytes being read were written by a process executing on a different compute node than the reading process, then the performance is slower than the first two cases and the application must introduce synchronization operations to ensure that the most recent data is read.

In summary, reading the local data (which has been written by processes executing on the same compute node) will always be faster than reading remote data.

Note that, as we discuss above, commit semantics also require inter-process synchronization for potentially conflicting write accesses. If an application does not enforce sequential ordering of file modifications during a write phase, e.g., with MPI synchronization, and multiple processes write concurrently to the same file offset or to an overlapping region, the result is undefined and may reflect the result of a mixture of the processes' operations to that offset or region.

3.3.2 Lamination Consistency Semantics in UnifyFS

The other consistency model that UnifyFS employs is called "lamination semantics" which is intended to be applied once a file is done being modified at the end of a write phase of an application. After a file is laminated, it becomes permanently read-only and its data is accessible across all the compute nodes in the job without further synchronization. Once a file is laminated, it cannot be further modified, except for being renamed or deleted.

A typical use case for lamination is for checkpoint/restart. An application can laminate checkpoint files after they have been successfully written so that they can be read by any process on any compute node in the job in a restart operation. To laminate a file, an application can simply call `chmod()` to remove all the write bits, after its write phase is completed. When write bits of a file are removed, UnifyFS will laminate the file. A typical checkpoint write operation with UnifyFS will look like:

```
fd = open("checkpoint1.chk", O_WRONLY)
write(fd, <checkpoint data>, <len>)
close(fd)
chmod("checkpoint1.chk", 0444)
```

We plan for future versions of UnifyFS to support different methods for laminating files, such as laminating all files on `close()` or laminating via an explicit API call.

We define the laminated consistency model to enable certain optimizations while supporting the typical requirements of bulk synchronous I/O. Recall that for bulk synchronous I/O patterns, reads and writes typically occur in distinct phases. This means that for the majority of the time, processes do not need to read arbitrary bytes of a file until the write phase is completed, which in practice is when the file is done being modified and closed and can be safely made read-only with lamination. For applications in which processes do not need to access file data modified by other processes before lamination, UnifyFS can optimize write performance by buffering all metadata and file data for processes locally, instead of performing costly exchanges of metadata and file data between compute nodes on every write. Also, since file contents cannot change after lamination, aggressive caching may be used during the read phase with minimal locking.

3.4 File System Behavior

The following summarize the behavior of UnifyFS under our consistency model.

Failure behavior:

- In the event of a compute node failure, all file data from the processes running on the failed compute node will be lost.

- In the event of the failure of a UnifyFS server process, all file data from the processes assigned to that server process (typically on the same compute node) will be lost.
- In the event of application process failures when the UnifyFS server processes remain running, the file data can be retrieved by the local UnifyFS server or a remote UnifyFS server.
- The UnifyFS team plans to improve the reliability of UnifyFS in the event of failures using redundancy scheme implementations available from the [VeloC](#) project as part of a future release.

Behavior before lamination (write phase):

- open/close: A process may open/close a file multiple times.
- write: A process may write to any part of a file. If two processes write to the same location concurrently (i.e., without inter-process synchronization to enforce ordering), the result is undefined.
- read: A process may read bytes it has written. Reading other bytes is invalid without explicit synchronization operations.
- rename: A process may rename a file.
- truncate: A process may truncate a file.
- unlink: A process may delete a file.

Behavior after lamination (read phase):

- open/close: A process may open/close a file multiple times.
- write: All writes are invalid.
- read: A process may read any byte in the file.
- rename: A process may rename a file.
- truncate: Truncation is invalid (considered to be a write operation).
- unlink: A process may delete a file.

The additional behavior of UnifyFS can be summarized as follows.

- UnifyFS exists on node local storage only and is not automatically persisted to stable storage like a parallel file system (PFS). When the data needs to be persisted to an external file system, users can use *unifyfs utility* with its data staging options.
- UnifyFS also can be coupled with [SymphonyFS](#), high level I/O libraries, or a checkpoint library like [SCR](#) or [VeloC](#) to move data to the PFS periodically.
- UnifyFS can be used with checkpointing libraries like [SCR](#) or [VeloC](#), or with I/O libraries like [HDF5](#) to support shared files on burst buffers.
- The UnifyFS file system will be empty at job start. A user job must populate the file system manually or by using *unifyfs utility*.
- UnifyFS creates a shared file system namespace across all compute nodes in a job, even if an application process is not running on all compute nodes.
- UnifyFS survives across multiple application runs within a job.
- UnifyFS transparently intercepts system level I/O calls of applications and I/O libraries.

This section describes how to build UnifyFS and its dependencies. There are three options:

- build both UnifyFS and its dependencies with Spack,
 - build the dependencies with Spack, but build UnifyFS with autotools
 - build the dependencies with a bootstrap script, and build UnifyFS with autotools
-

4.1 Build UnifyFS and its dependencies with Spack

One may install UnifyFS and its dependencies with [Spack](#). If you already have Spack, make sure you have the latest release. If you use a clone of the Spack develop branch, be sure to pull the latest changes.

4.1.1 Install Spack

```
$ git clone https://github.com/spack/spack
$ # create a packages.yaml specific to your machine
$ . spack/share/spack/setup-env.sh
```

Use [Spack's shell support](#) to add Spack to your PATH and enable use of the `spack` command.

4.1.2 Build UnifyFS and its dependencies

```
$ spack install unifyfs
$ spack load unifyfs
```

If the most recent changes on the development branch ('dev') of UnifyFS are desired, then do `spack install unifyfs@develop`.

Include or remove variants with Spack when installing UnifyFS when a custom build is desired. Run `spack info unifyfs` for more info.

Table 1: UnifyFS Build Variants

Variant	Command (spack install <package>)	Default	Description
Auto-mount	<code>unifyfs+auto-mount</code>	True	Enable transparent mounting
HDF5	<code>unifyfs+hdf5</code>	False	Build with parallel HDF5
	<code>unifyfs+hdf5 ^hdf5~mpi</code>	False	Build with serial HDF5
Fortran	<code>unifyfs+fortran</code>	False	Enable Fortran support
PMI	<code>unifyfs+pmi</code>	False	Enable PMI2 support
PMIx	<code>unifyfs+pmix</code>	False	Enable PMIx support
spath	<code>unifyfs+spath</code>	True	Normalize relative paths

Attention: The initial install could take a while as Spack will install build dependencies (autoconf, automake, m4, libtool, and pkg-config) as well as any dependencies of dependencies (cmake, perl, etc.) if you don't already have these dependencies installed through Spack or haven't told Spack where they are locally installed on your system (i.e., through a custom `packages.yaml`). Run `spack spec -I unifyfs` before installing to see what Spack is going to do.

4.2 Build dependencies with Spack, build UnifyFS with autotools

One can install the UnifyFS dependencies with Spack and build UnifyFS with autotools. This is useful if one needs to modify the UnifyFS source code between builds. Take advantage of [Spack Environments](#) to streamline this process.

4.2.1 Build the dependencies

Once Spack is installed on your system (see [above](#)), the UnifyFS dependencies can then be installed.

```
$ spack install gotcha
$ spack install mochi-margo ^libfabric fabrics=rxm,sockets,tcp
$ spack install spath~mpi
```

Tip: You can run `spack install --only=dependencies unifyfs` to install all UnifyFS dependencies without installing UnifyFS.

Keep in mind this will also install all the build dependencies and dependencies of dependencies if you haven't already installed them through Spack or told Spack where they are locally installed on your system via a `packages.yaml`.

4.2.2 Build UnifyFS

Download the latest UnifyFS release from the [Releases](#) page or clone the develop branch from the UnifyFS repository <https://github.com/LLNL/UnifyFS>.

Load the dependencies into your environment and then configure and build UnifyFS from its source code directory.

```
$ spack load gotcha
$ spack load argobots
$ spack load mercury
$ spack load mochi-margo
$ spack load spath
$
$ gotcha_install=$(spack location -i gotcha)
$ spath_install=$(spack location -i spath)
$
$ ./autogen.sh
$ ./configure --prefix=/path/to/install --with-gotcha=${gotcha_install} --with-spath=$
→{spath_install}
$ make
$ make install
```

Alternatively, UnifyFS can be configured using CPPFLAGS and LDFLAGS:

```
$ ./configure --prefix=/path/to/install CPPFLAGS="-I${gotcha_install}/include -I
→{spath_install}/include" LDFLAGS="-L${gotcha_install}/lib64 -L${spath_install}/lib64
```

To see all available build configuration options, run `./configure --help` after `./autogen.sh` has been run.

4.3 Build dependencies with bootstrap and build UnifyFS with auto-tools

Download the latest UnifyFS release from the [Releases](#) page or clone the develop branch from the UnifyFS repository <https://github.com/LLNL/UnifyFS>.

4.3.1 Build the Dependencies

UnifyFS requires MPI, GOTCHA, Margo and OpenSSL. References to these dependencies can be found on our *UnifyFS Dependencies* page.

A `bootstrap.sh` script in the UnifyFS source distribution downloads and installs all dependencies. Simply run the script in the top level directory of the source code.

```
$ ./bootstrap.sh
```

Note: UnifyFS requires automake version 1.15 or newer in order to build.

Before building the UnifyFS dependencies, the `bootstrap.sh` script will check the system's current version of automake and attempt to build the autotools suite if an older version is detected.

4.3.2 Build UnifyFS

After `bootstrap.sh` installs the dependencies, it prints the commands one needs to execute to build UnifyFS. As an example, the commands may look like:

```
$ export PKG_CONFIG_PATH=$INSTALL_DIR/lib/pkgconfig:$INSTALL_DIR/lib64/pkgconfig:$PKG_
→CONFIG_PATH
$ export LD_LIBRARY_PATH=$INSTALL_DIR/lib:$INSTALL_DIR/lib64:$LD_LIBRARY_PATH
$ ./autogen.sh
$ ./configure --prefix=/path/to/install CPPFLAGS=-I/path/to/install/include LDFLAGS="-
→L/path/to/install/lib -L/path/to/install/lib64"
$ make
$ make install
```

Alternatively, UnifyFS can be configured using `--with` options:

```
$ ./configure --prefix=/path/to/install --with-gotcha=$INSTALL_DIR --with-spath=
→$INSTALL_DIR
```

To see all available build configuration options, run `./configure --help` after `./autogen.sh` has been run.

Note: On Cray systems, the detection of MPI compiler wrappers requires passing the following flags to the configure command: `MPICC=cc MPIFC=ftn`

4.4 Configure Options

When building UnifyFS with autotools, a number of options are available to configure its functionality.

4.4.1 Fortran

To use UnifyFS in Fortran applications, pass the `--enable-fortran` option to `configure`. Note that only GCC Fortran (i.e., `gfortran`) is known to work with UnifyFS. There is an open [ifort_issue](#) with the Intel Fortran compiler as well as an [xlf_issue](#) with the IBM Fortran compiler.

Note: UnifyFS requires GOTCHA when Fortran support is enabled

4.4.2 GOTCHA

GOTCHA is the preferred method for I/O interception with UnifyFS, but it is not available on all platforms. If GOTCHA is not available on your target system, you can omit it during UnifyFS configuration by using the `--without-gotcha` configure option. Without GOTCHA, static linker wrapping is required for I/O interception, see [Link with the UnifyFS library](#).

Warning: UnifyFS requires GOTCHA for dynamic I/O interception of MPI-IO functions. If UnifyFS is configured using `--without-gotcha`, support will be lost for MPI-IO (and as a result, HDF5) applications.

4.4.3 HDF5

UnifyFS includes example programs that use HDF5. If HDF5 is not available on your target system, it can be omitted during UnifyFS configuration by using the `--without-hdf5` configure option.

4.4.4 PMI2/PMIx Key-Value Store

When available, UnifyFS uses the distributed key-value store capabilities provided by either PMI2 or PMIx. To enable this support, pass either the `--enable-pmi` or `--enable-pmix` option to configure. Without PMI support, a distributed file system accessible to all servers is required.

4.4.5 SPATH

The spath library can be optionally used to normalize relative paths (e.g., ones containing “.”, “..”, and extra or trailing “/”) and enable the support of using relative paths within an application. To enable, use the `--with-spath` configure option or provide the appropriate `CPPFLAGS` and `LDFLAGS` at configure time.

4.4.6 Transparent Mounting for MPI Applications

MPI applications written in C or C++ may take advantage of the UnifyFS transparent mounting capability. With transparent mounting, calls to `unifyfs_mount()` and `unifyfs_unmount()` are automatically performed during `MPI_Init()` and `MPI_Finalize()`, respectively. Transparent mounting always uses `/unifyfs` as the namespace mountpoint. To enable transparent mounting, use the `--enable-mpi-mount` configure option.

Integrate the UnifyFS API

This section describes how to use the UnifyFS API in an application.

Attention: Fortran Compatibility

`unifyfs_mount` and `unifyfs_unmount` are now usable with GFortran. There is a known `ifort_issue` with the Intel Fortran compiler as well as an `xlf_issue` with the IBM Fortran compiler. Other Fortran compilers are currently unknown.

If using fortran, when *installing UnifyFS* with Spack, include the `+fortran` variant, or configure UnifyFS with the `--enable-fortran` option if building manually.

5.1 Include the UnifyFS header

In C or C++ applications, include `unifyfs.h`. See `writeread.c` for a full example.

Listing 1: C

```
#include <unifyfs.h>
```

In Fortran applications, include `unifyfsf.h`. See `writeread.f90` for a full example.

Listing 2: Fortran

```
include 'unifyfsf.h'
```

5.2 Mounting

UnifyFS implements a file system in user space, which the system has no knowledge about. The UnifyFS library intercepts and handles I/O calls whose path matches a prefix that is defined by the user. Calls corresponding to matching

paths are handled by UnifyFS and all other calls are forwarded to the original I/O routine.

To use UnifyFS, the application must register the path that the UnifyFS library should intercept by making a call to `unifyfs_mount`. This must be done once on each client process, and it must be done before the client process attempts to access any UnifyFS files.

For instance, to use UnifyFS on all path prefixes that begin with `/unifyfs` this would require a:

Listing 3: C

```
unifyfs_mount('/unifyfs', rank, rank_num);
```

Listing 4: Fortran

```
call UNIFYFS_MOUNT('/unifyfs', rank, size, ierr);
```

Here, `/unifyfs` is the path prefix for UnifyFS to intercept. The `rank` parameter specifies the MPI rank of the calling process. The `size` parameter specifies the number of MPI ranks in the user job.

5.3 Unmounting

When the application is done using UnifyFS, it should call `unifyfs_unmount`.

Listing 5: C

```
unifyfs_unmount();
```

Listing 6: Fortran

```
call UNIFYFS_UNMOUNT(ierr);
```

Link with the UnifyFS library

This section describes how to link an application with the UnifyFS library. The UnifyFS library contains symbols for the UnifyFS API, like `unifyfs_mount`, as well as wrappers for I/O routines, like `open`, `write`, and `close`. In the examples below, replace `<unifyfs>` with the path to your UnifyFS install.

6.1 Static link

For a static link, UnifyFS utilizes the `--wrap` feature of the `ld` command. One must specify a `--wrap` option for every I/O call that is wrapped, for which there are many. To make this easier, UnifyFS installs a `unifyfs-config` script that one should invoke to specify those flags, e.g.,

```
$ mpicc -o test_write \  
  `  
  <unifyfs>/bin/unifyfs-config --pre-ld-flags` \  
  test_write.c \  
  `  
  <unifyfs>/bin/unifyfs-config --post-ld-flags`
```

6.2 Dynamic link

To intercept I/O calls using `gotcha`, use the following syntax to link an application:

6.2.1 C

```
$ mpicc -o test_write test_write.c \  
  -I<unifyfs>/include -L<unifyfs>/lib -lunifyfs_gotcha \  
  -L<gotcha>/lib64 -lgotcha
```

6.2.2 Fortran

```
$ mpif90 -o test_write test_write.F \  
-I<unifyfs>/include -L<unifyfs>/lib -lunifyfsf -lunifyfs_gotcha
```

UnifyFS Configuration

Here, we explain how users can customize the runtime behavior of UnifyFS. In particular, UnifyFS provides the following ways to configure:

- Configuration file: `$INSTALL_PREFIX/etc/unifyfs/unifyfs.conf`
- Environment variables
- Command line options to `unifyfsd`

All configuration settings have corresponding environment variables, but only certain settings have command line options. When defined via multiple methods, the command line options have the highest priority, followed by environment variables, and finally config file options from `unifyfs.conf`.

The system-wide configuration file is used by default when available. However, users can specify a custom location for the configuration file using the `-f` command-line option to `unifyfsd` (see below). There is a sample `unifyfs.conf` file in the installation directory under `etc/unifyfs/`. This file is also available in the `extras` directory in the source repository.

The unified method for providing configuration control is adapted from [CONFIGURATOR](#). Configuration settings are grouped within named sections, and each setting consists of a key-value pair with one of the following types:

- **BOOL**: `0|1, y|n, Y|N, yes|no, true|false, on|off`
- **FLOAT**: scalars convertible to C double, or compatible expression
- **INT**: scalars convertible to C long, or compatible expression
- **STRING**: quoted character string

7.1 `unifyfs.conf`

`unifyfs.conf` specifies the system-wide configuration options. The file is written in [INI](#) language format, as supported by the `inih` parser.

The config file has several sections, each with a few key-value settings. In this description, we use `section.key` as shorthand for the name of a given section and key.

Table 1: [unifyfs] section - main configuration settings

Key	Type	Description
cleanup	BOOL	cleanup storage on server exit (default: off)
configfile	STRING	path to custom configuration file
consistency	STRING	consistency model [LAMINATED POSIX NONE]
daemonize	BOOL	enable server daemonization (default: off)
mountpoint	STRING	mountpoint path prefix (default: /unifyfs)

Table 2: [client] section - client settings

Key	Type	Description
cwd	STRING	effective starting current working directory
fsync_persist	BOOL	persist data to storage on fsync() (default: on)
local_extents	BOOL	service reads from local data if possible (default: off)
max_files	INT	maximum number of open files per client process (default: 128)
super_magic	BOOL	whether to return UNIFYFS (on) or TMPFS (off) statfs magic (default: on)
write_index_size	INT	maximum size (B) of memory buffer for storing write log metadata
write_sync	BOOL	sync data to server after every write (default: off)

The `cwd` setting is used to emulate the behavior one expects when changing into a working directory before starting a job and then using relative file names within the application. If set, the application changes its working directory to the value specified in `cwd` when `unifyfs_mount()` is called. The value specified in `cwd` must be within the directory space of the UnifyFS mount point.

Enabling the `local_extents` optimization may significantly improve read performance for extents written by the same process. However, it should not be used by applications in which different processes write to the same byte offset within a file, nor should it be used with applications that truncate files.

Table 3: [log] section - logging settings

Key	Type	Description
dir	STRING	path to directory to contain server log file
file	STRING	log file base name (rank will be appended)
on_error	BOOL	increase log verbosity upon encountering an error (default: off)
verbosity	INT	logging verbosity level [0-5] (default: 0)

Table 4: [logio] section - log-based write data storage settings

Key	Type	Description
chunk_size	INT	data chunk size (B) (default: 4 MiB)
shmem_size	INT	maximum size (B) of data in shared memory (default: 256 MiB)
spill_size	INT	maximum size (B) of data in spillover file (default: 4 GiB)
spill_dir	STRING	path to spillover data directory

Table 5: [runstate] section - server runstate settings

Key	Type	Description
dir	STRING	path to directory to contain server-local state

Table 6: [server] section - server settings

Key	Type	Description
hostfile	STRING	path to server hostfile
init_timeout	INT	timeout in seconds to wait for servers to be ready for clients (default: 120)

Table 7: [margo] section - margo server NA settings

Key	Type	Description
tcp	BOOL	Use TCP for server-to-server rpcs (default: on, turn off to enable libfabric RMA)

Table 8: [sharedfs] section - server shared files settings

Key	Type	Description
dir	STRING	path to directory to contain server shared files

7.2 Environment Variables

All environment variables take the form UNIFYFS_SECTION_KEY, except for the [unifyfs] section, which uses UNIFYFS_KEY. For example, the setting `log.verbosity` has a corresponding environment variable named UNIFYFS_LOG_VERBOSITY, while `unifyfs.mountpoint` corresponds to UNIFYFS_MOUNTPOINT.

7.3 Command Line Options

For server command line options, we use `getopt_long()` format. Thus, all command line options have long and short forms. The long form uses `--section-key=value`, while the short form `-<optchar> value`, where the short option character is given in the below table.

Note that for configuration options of type `BOOL`, the value is optional. When not provided, the `true` value is assumed. If the short form option is used, the value must immediately follow the option character (e.g., `-Cyes`).

Table 9: unifyfsd command line options

LongOpt	ShortOpt
<code>--unifyfs-cleanup</code>	<code>-C</code>
<code>--unifyfs-configfile</code>	<code>-f</code>
<code>--unifyfs-consistency</code>	<code>-c</code>
<code>--unifyfs-daemonize</code>	<code>-D</code>
<code>--unifyfs-mountpoint</code>	<code>-m</code>
<code>--log-verbosity</code>	<code>-v</code>
<code>--log-file</code>	<code>-l</code>
<code>--log-dir</code>	<code>-L</code>
<code>--runstate-dir</code>	<code>-R</code>
<code>--server-hostfile</code>	<code>-H</code>
<code>--sharedfs-dir</code>	<code>-S</code>
<code>--server-init_timeout</code>	<code>-t</code>

This section describes the mechanisms to start and stop the UnifyFS server processes within a job allocation.

Overall, the steps to run an application with UnifyFS include:

1. Allocate nodes using the system resource manager (i.e., start a job)
2. Update any desired UnifyFS server configuration settings
3. Start UnifyFS servers on each allocated node using `unifyfs`
4. Run one or more UnifyFS-enabled applications
5. Terminate the UnifyFS servers using `unifyfs`

8.1 Start UnifyFS

First, one must start the UnifyFS server process (`unifyfsd`) on the nodes in the job allocation. UnifyFS provides the `unifyfs` command line utility to simplify this action on systems with supported resource managers. The easiest way to determine if you are using a supported system is to run `unifyfs start` within an interactive job allocation. If no compatible resource management system is detected, the utility reports an error message to that effect.

In `start` mode, the `unifyfs` utility automatically detects the allocated nodes and launches a server on each node. For example, the following script could be used to launch the `unifyfsd` servers with a customized configuration. On systems with resource managers that propagate environment settings to compute nodes, the environment variables override any settings in `/etc/unifyfs/unifyfs.conf`. See *UnifyFS Configuration* for further details on customizing the UnifyFS runtime configuration.

```
1 #!/bin/bash
2
3 # spillover data to node-local ssd storage
4 export UNIFYFS_LOGIO_SPILL_DIR=/mnt/ssd/$USER/data
5
6 # store server logs in job-specific scratch area
7 export UNIFYFS_LOG_DIR=$JOBSCRATCH/logs
```

(continues on next page)

```
unifyfs start --share-dir=/path/to/shared/file/system
```

`unifyfs` provides command-line options to choose the client mountpoint, adjust the consistency model, and control stage-in and stage-out of files. The full usage for `unifyfs` is as follows:

```
[prompt]$ unifyfs --help

Usage: unifyfs <command> [options...]

<command> should be one of the following:
  start      start the UnifyFS server daemons
  terminate  terminate the UnifyFS server daemons

Common options:
  -d, --debug          enable debug output
  -h, --help          print usage

Command options for "start":
  -C, --consistency=<model> [OPTIONAL] consistency model (NONE | LAMINATED | POSIX)
  -e, --exe=<path>         [OPTIONAL] <path> where unifyfsd is installed
  -m, --mount=<path>      [OPTIONAL] mount UnifyFS at <path>
  -s, --script=<path>     [OPTIONAL] <path> to custom launch script
  -t, --timeout=<sec>     [OPTIONAL] wait <sec> until all servers become ready
  -S, --share-dir=<path> [REQUIRED] shared file system <path> for use by servers
  -c, --cleanup          [OPTIONAL] clean up the UnifyFS storage upon server exit
  -i, --stage-in=<manifest> [OPTIONAL] stage in file(s) listed in <manifest> file
  -T, --stage-timeout=<sec> [OPTIONAL] timeout for stage-in operation

Command options for "terminate":
  -o, --stage-out=<manifest> [OPTIONAL] stage out file(s) listed in <manifest> on_
  ↪ termination
  -T, --stage-timeout=<sec> [OPTIONAL] timeout for stage-out operation
  -s, --script=<path>       [OPTIONAL] <path> to custom termination script
  -S, --share-dir=<path>   [REQUIRED for --stage-out] shared file system <path> for_
  ↪ use by servers
```

After UnifyFS servers have been successfully started, you may run your UnifyFS-enabled applications as you normally would (e.g., using `mpirun`). Only applications that explicitly call `unifyfs_mount()` and access files under the specified mountpoint prefix will utilize UnifyFS for their I/O. All other applications will operate unchanged.

8.2 Stop UnifyFS

After all UnifyFS-enabled applications have completed running, use `unifyfs terminate` to terminate the servers. Pass the `--cleanup` option to `unifyfs start` to have the servers remove temporary data locally stored on each node after termination.

8.3 Resource Manager Job Integration

UnifyFS includes optional support for integrating directly with compatible resource managers to automatically start and stop servers at the beginning and end of a job when requested by users. Resource manager integration requires administrator privileges to deploy.

Currently, only IBM's Platform LSF with Cluster System Manager (LSF-CSM) is supported. LSF-CSM is the resource manager on the CORAL2 IBM systems at ORNL and LLNL. The required job prologue and epilogue scripts, along with a README documenting the installation instructions, is available within the source repository at `util/scripts/lsfcsm`.

Support for the SLURM resource manager is under development.

8.4 Transferring Data In and Out of UnifyFS

Data can be transferred in/out of UnifyFS during server startup and termination, or at any point during a job using two stand-alone applications.

8.4.1 Transfer at Server Start/Terminate

The transfer subsystem within UnifyFS can be invoked by providing the `-i|--stage-in` option to `unifyfs start` to transfer files into UnifyFS:

```
$ unifyfs start --stage-in=/path/to/input/manifest/file --share-dir=/path/to/shared/
↳file/system
```

and/or by providing the `-o|--stage-out`, and consequently required `-S|--share-dir`, option to `unifyfs terminate` to transfer files out of UnifyFS:

```
$ unifyfs terminate --stage-out=/path/to/output/manifest/file --share-dir=/path/to/
↳shared/file/system
```

A manifest file needs to be provided to the `start/terminate` commands in order to specify the desired transfers.

Manifest File

UnifyFS's stage functionality requires a manifest file in order to move data.

The manifest file contains one or more file copy requests. Each line in the manifest corresponds to one transfer request, and it contains both the source and destination file paths. Directory copies are currently not supported.

Each line is formatted as: `<source filename> <whitespace> <destination filename>`.

If either of the filenames contain whitespace or special characters, then both filenames should be surrounded by double-quote characters (") (ASCII character 34 decimal). The double-quote and linefeed end-of-line characters are not supported in any filenames used in a `unifyfs` manifest file. Any other characters are allowed, including control characters. If a filename contains any characters that might be misinterpreted, then enclosing the filename in double-quotes is always a safe thing to do.

Here is an example of a valid stage-in manifest file:

```
$ [prompt] cat example_stage_in.manifest

/scratch/users/me/input_data/input_1.dat /unifyfs/input/input_1.dat
/home/users/me/configuration/run_12345.conf /unifyfs/config/run_12345.conf
"/home/users/me/file with space.dat" "/unifyfs/file with space.dat"
```

8.4.2 Transfer During Job

Data can also be transferred in/out of UnifyFS using one of two stand-alone applications.

The stand-alone applications can be invoked at any time while the UnifyFS servers are up and responding to requests. This allows for bringing in new input and/or transferring results out to be verified before the job terminates.

UnifyFS Stage Executable

```
$UNIFYFS_INSTALL/libexec/unifyfs-stage
```

The start/terminate transfer API stage functionality can also be used via the stand-alone application `unifyfs-stage`.

This application can be run at any time within a job to transfer new data into or results out of UnifyFS.

A manifest file (see [:ref:'above <manifest_file_label>'](#)_) needs to be provided as an argument to use this approach.

```
[prompt]$ ./unifyfs-stage --help

Usage: unifyfs-stage [OPTION]... <manifest file>

Transfer files between unifyfs volume and external file system.
The <manifest file> should contain list of files to be transferred,
and each line should be formatted as

    /source/file/path /destination/file/path

OR in the case of filenames with spaces or special characters:

    "/source/file/path" "/destination/file/path"

One file per line; Specifying directories is not supported.

Available options:
  -c, --checksum           verify md5 checksum for each transfer
  -h, --help               print this usage
  -m, --mountpoint=<mnt>  use <mnt> as unifyfs mountpoint
                          (default: /unifyfs)
  -p, --parallel           transfer each file in parallel
                          (experimental)
  -s, --share-dir=<path>  directory path for creating status file
  -v, --verbose            print noisy outputs

Without the '-p, --parallel' option, a file is transferred by a single
process. If the '-p, --parallel' option is specified, each file will be
divided by multiple processes and transferred in parallel.
```

Examples:

Listing 1: Serial Transfer

```
$ srun -N 1 -n 1 $UNIFYFS_INSTALL/libexec/unifyfs-stage $MY_MANIFEST_FILE
```

Listing 2: Parallel Transfer

```
$ srun -N 4 -n 8 $UNIFYFS_INSTALL/libexec/unifyfs-stage --parallel $MY_MANIFEST_FILE
```

Transfer Executable

```
$UNIFYFS_INSTALL/libexec/transfer-static
```

Note: The `transfer-gotcha` executable is currently unusable due to an issue that is being tracked.

The transfer API can also be used during the job by invoking the stand-alone `transfer` application. It works similarly to the Unix `cp` command, with source and destination, except being aware that it is copying files between an external file system and internal UnifyFS.

```
[prompt]$ transfer-static --help

Usage: transfer-static [options...] <source path> <destination path>

Available options:
-d, --debug           pause before running test
                      (handy for attaching in debugger)
-h, --help           help message
-m, --mount=<mountpoint> use <mountpoint> for unifyfs
                      (default: /unifyfs)
-p, --parallel       parallel transfer
-r, --rank=<rank>    use <rank> for transfer (default: 0)
```

Examples of using `transfer-static`:

Listing 3: Serial Transfer

```
$ srun -N 1 -n 1 $UNIFYFS_INSTALL/libexec/transfer-static /path/on/parallels/file.
↳dat /unifyfs/file.dat

$ srun -N 1 -n 1 $UNIFYFS_INSTALL/libexec/transfer-static /unifyfs/output.dat /
↳scratch/my_output/output.dat
```

Listing 4: Parallel Transfer

```
$ srun -N 4 -n 8 /path/to/libexec/transfer-static -parallel /path/on/parallels/file.
↳dat /unifyfs/file.dat

$ srun -N 4 -n 8 /path/to/libexec/transfer-static -parallel /unifyfs/output.dat /
↳scratch/my_output/output.dat
```


There are several [examples](#) available on ways to use UnifyFS. These examples build into static, GOTCHA, and pure POSIX (not linked with UnifyFS) versions depending on how they are linked. Several of the example programs are also used in the UnifyFS *integration testing*.

9.1 Examples Locations

The example programs can be found in two locations, where UnifyFS is built and where UnifyFS is installed.

9.1.1 Install Location

Upon installation of UnifyFS, the example programs are installed into the *install/libexec* folder.

Installed with Spack

The Spack installation location of UnifyFS can be found with the command `spack location -i unifyfs`.

To easily navigate to this location and find the examples, do:

```
$ spack cd -i unifyfs
$ cd libexec
```

Installed with Autotools

The autotools installation of UnifyFS will place the example programs in the *libexec/* directory of the path provided to `--prefix=/path/to/install` during the configure step of *building and installing*.

9.1.2 Build Location

Built with Spack

The Spack build location of UnifyFS (on a successful install) only exists when `--keep-stage` is included during installation or if the build fails. This location can be found with the command `spack location unifyfs`.

To navigate to the location of the static and POSIX examples, do:

```
$ spack install --keep-stage unifyfs
$ spack cd unifyfs
$ cd spack-build/examples/src
```

The GOTCHA examples are one directory deeper in `spack-build/examples/src/libs`.

Note: If you installed UnifyFS with any variants, in order to navigate to the build directory you must include these variants in the `spack cd` command. E.g.:

```
spack cd unifyfs+hdf5 ^hdf5~mpi
```

Built with Autotools

The autotools build of UnifyFS will place the static and POSIX example programs in the `examples/src` directory and the GOTCHA example programs in the `examples/src/libs` directory of your build directory.

9.2 Running the Examples

In order to run any of the example programs you first need to start the UnifyFS server daemon on the nodes in the job allocation. To do this, see [Run UnifyFS](#).

Each example takes multiple arguments and so each has its own `--help` option to aid in this process.

```
[prompt]$ ./write-static --help

Usage: write-static [options...]

Available options:
  -a, --appid=<id>           use given application id
                             (default: 0)
  -A, --aio                  use asynchronous I/O instead of read|write
                             (default: off)
  -b, --blocksize=<bytes>   I/O block size
                             (default: 16 MiB)
  -c, --chunksize=<bytes>   I/O chunk size for each operation
                             (default: 1 MiB)
  -d, --debug                for debugging, wait for input (at rank 0) at start
                             (default: off)
  -f, --file=<filename>     target file name (or path) under mountpoint
                             (default: 'testfile')
  -k, --check                check data contents upon read
                             (default: off)
```

(continues on next page)

(continued from previous page)

-L, --listio	use lio_listio instead of read write (default: off)
-m, --mount=<mountpoint>	use <mountpoint> for unifyfs (default: /unifyfs)
-M, --mpio	use MPI-IO instead of POSIX I/O (default: off)
-n, --nblocks=<count>	count of blocks each process will read write (default: 32)
-N, --mapio	use mmap instead of read write (default: off)
-o, --outfile=<filename>	output file name (or path) (default: 'stdout')
-p, --pattern=<pattern> ↪process)	'n1' (N-to-1 shared file) or 'nn' (N-to-N file per_ (default: 'n1')
-P, --prdwr	use pread pwrite instead of read write (default: off)
-S, --stdio	use fread fwrite instead of read write (default: off)
-t, --pre-truncate=<size>	truncate file to size (B) before writing (default: off)
-T, --post-truncate=<size>	truncate file to size (B) after writing (default: off)
-U, --disable-unifyfs	do not use UnifyFS (default: enable UnifyFS)
-v, --verbose	print verbose information (default: off)
-V, --vecio	use readv writev instead of read write (default: off)
-x, --shuffle	read different data than written (default: off)

One form of running this example could be:

```
$ srun -N4 -n4 write-static -m /unifyfs -f myTestFile
```

9.3 Producer-Consumer Workflow

UnifyFS can be used to support producer/consumer workflows where processes in a job perform loosely synchronized communication through files such as in coupled simulation/analytics workflows.

The *write.c* and *read.c* example programs can be used as a basic test in running a producer-consumer workflow with UnifyFS.

Listing 1: All hosts in allocation

```
$ # start unifyfs
$
$ # write on all hosts
$ srun -N4 -n16 write-gotcha -f testfile
$
$ # read on all hosts
$ srun -N4 -n16 read-gotcha -f testfile
$
$ # stop unifyfs
```

Listing 2: Disjoint hosts in allocation

```
$ # start unifyfs
$
$ # write on half of hosts
$ srun -N2 -n8 --exclude=$hostlist_subset1 write-gotcha -f testfile
$
$ # read on other half of hosts
$ srun -N2 -n8 --exclude=$hostlist_subset2 read-gotcha -f testfile
$
$ # stop unifyfs
```

Note: Producer/consumer support with UnifyFS has been tested using POSIX and MPI-IO APIs on x86_64 (MVA-PICH) and Power 9 systems (Spectrum MPI).

These scenarios have been tested using both the same and disjoint sets of hosts as well as using a shared file and a file per process for I/O.

UnifyFS API for I/O Middleware

This section describes the purpose, concepts, and usage of the UnifyFS library API.

10.1 Library API Purpose

The UnifyFS library API provides a direct interface for UnifyFS configuration, namespace management, and batched file I/O and transfer operations. The library is primarily targeted for use by I/O middleware software such as HDF5 and VeloC, but is also useful for user applications needing programmatic control and interactions with UnifyFS.

Note: Use of the library API is *not* required for most applications, as UnifyFS will transparently intercept I/O operations made by the application. See *Examples* for examples of typical application usage.

10.2 Library API Concepts

10.2.1 Namespace (aka Mountpoint)

All UnifyFS clients provide the mountpoint prefix (e.g., “/unifyfs”) that is used to distinguish the UnifyFS namespace from other file systems available to the client application. All absolute file paths that include the mountpoint prefix are treated as belonging to the associated UnifyFS namespace.

Using the library API, an application or I/O middleware system can operate on multiple UnifyFS namespaces concurrently.

10.2.2 File System Handle

All library API methods require a file system handle parameter of type `unifyfs_handle`. Users obtain a valid handle via an API call to `unifyfs_initialize()`, which specifies the mountpoint prefix and configuration settings associated with the handle.

Multiple handles can be acquired by the same client. This permits access to multiple namespaces, or different configured behaviors for the same namespace.

10.2.3 Global File Identifier

A global file identifier (gfid) is a unique integer identifier for a given absolute file path within a UnifyFS namespace. Clients accessing the exact same file path are guaranteed to obtain the same gfid value when creating or opening the file. I/O operations use the gfid to identify the target file.

Note that unlike POSIX file descriptors, a gfid is strictly a unique identifier and has no associated file state such as a current file position pointer. As such, it is valid to obtain the gfid for a file in a single process (e.g., via file creation), and then share the resulting gfid value among other parallel processes via a collective communication mechanism.

10.3 Library API Types

The file system handle type is a pointer to an opaque client structure that records the associated mountpoint and configuration.

Listing 1: File system handle type

```
/* UnifyFS file system handle (opaque pointer) */
typedef struct unifyfs_client* unifyfs_handle;
```

I/O requests take the form of a `unifyfs_io_request` structure that includes the target file gfid, the specific I/O operation (`unifyfs_ioreq_op`) to be applied, and associated operation parameters such as the file offset or user buffer and size. The structure also contains fields used for tracking the status of the request (`unifyfs_ioreq_state`) and operation results (`unifyfs_ioreq_result`).

Listing 2: File I/O request types

```
/* I/O request structure */
typedef struct unifyfs_io_request {
    /* user-specified fields */
    void* user_buf;
    size_t nbytes;
    off_t offset;
    unifyfs_gfid gfid;
    unifyfs_ioreq_op op;

    /* status/result fields */
    unifyfs_ioreq_state state;
    unifyfs_ioreq_result result;
} unifyfs_io_request;

/* enumeration of supported I/O request operations */
typedef enum unifyfs_ioreq_op {
    UNIFYFS_IOREQ_NOP = 0,
    UNIFYFS_IOREQ_OP_READ,
    UNIFYFS_IOREQ_OP_WRITE,
    UNIFYFS_IOREQ_OP_SYNC_DATA,
    UNIFYFS_IOREQ_OP_SYNC_META,
    UNIFYFS_IOREQ_OP_TRUNC,
    UNIFYFS_IOREQ_OP_ZERO,
} unifyfs_ioreq_op;
```

(continues on next page)

(continued from previous page)

```

/* enumeration of I/O request states */
typedef enum unifyfs_ioreq_state {
    UNIFYFS_IOREQ_STATE_INVALID = 0,
    UNIFYFS_IOREQ_STATE_IN_PROGRESS,
    UNIFYFS_IOREQ_STATE_CANCELED,
    UNIFYFS_IOREQ_STATE_COMPLETED
} unifyfs_ioreq_state;

/* structure to hold I/O request result values */
typedef struct unifyfs_ioreq_result {
    int error;
    int rc;
    size_t count;
} unifyfs_ioreq_result;

```

For the `unifyfs_ioreq_result` structure, successful operations will set the `rc` and `count` fields as applicable to the specific operation type. All operational failures are reported by setting the `error` field to a non-zero value corresponding to the operation failure code, which is often a POSIX `errno` value.

File transfer requests use a `unifyfs_transfer_request` structure that includes the source and destination file paths, transfer mode, and a flag indicating whether parallel file transfer should be used. Similar to I/O requests, the structure also contains fields used for tracking the request status and transfer operation result.

Listing 3: File transfer request types

```

/* File transfer request structure */
typedef struct unifyfs_transfer_request {
    /* user-specified fields */
    const char* src_path;
    const char* dst_path;
    unifyfs_transfer_mode mode;
    int use_parallel;

    /* status/result fields */
    unifyfs_ioreq_state state;
    unifyfs_ioreq_result result;
} unifyfs_transfer_request;

/* enumeration of supported I/O request operations */
typedef enum unifyfs_transfer_mode {
    UNIFYFS_TRANSFER_MODE_INVALID = 0,
    UNIFYFS_TRANSFER_MODE_COPY, // simple copy to destination
    UNIFYFS_TRANSFER_MODE_MOVE // copy, then remove source
} unifyfs_transfer_mode;

```

10.4 Example Library API Usage

To get started using the library API, please add the following to your client source code files that will make calls to API methods. You will also need to modify your client application build process to link with the `libunifyfs_api` library.

Listing 4: Including the API header

```
#include <unifyfs/unifyfs_api.h>
```

The common pattern for using the library API is to initialize a UnifyFS file system handle, perform a number of operations using that handle, and then release the handle. As previously mentioned, the same client process may initialize multiple file system handles and use them concurrently, either to work with multiple namespaces, or to use different configured behaviors with different handles sharing the same namespace.

10.4.1 File System Handle Initialization and Finalization

To initialize a handle to UnifyFS, the client application uses the `unifyfs_initialize()` method as shown below. This method takes the namespace mountpoint prefix and an array of optional configuration parameter settings as input parameters, and initializes the value of the passed file system handle upon success.

In the example below, the `logio.chunk_size` configuration parameter, which controls the size of the log-based I/O data chunks, is set to the value of 32768. See *UnifyFS Configuration* for further options for customizing the behavior of UnifyFS.

Listing 5: UnifyFS handle initialization

```
int n_configs = 1;
unifyfs_cfg_option chk_size = { .opt_name = "logio.chunk_size",
                               .opt_value = "32768" };

const char* unifyfs_prefix = "/my/unifyfs/namespace";
unifyfs_handle fshdl = UNIFYFS_INVALID_HANDLE;
int rc = unifyfs_initialize(unifyfs_prefix, &chk_size, n_configs, &fshdl);
```

Once all UnifyFS operation using the handle have been completed, the client application should call `unifyfs_finalize()` as shown below to release the resources associated with the handle.

Listing 6: UnifyFS handle finalization

```
int rc = unifyfs_finalize(fshdl);
```

10.4.2 File Creation, Use, and Removal

New files should be created by a single client process using `unifyfs_create()` as shown below. Note that if multiple clients attempt to create the same file, only one will succeed.

Note: Currently, the `create_flags` parameter is unused; it is reserved for future use to indicate file-specific UnifyFS behavior.

Listing 7: UnifyFS file creation

```
const char* filename = "/my/unifyfs/namespace/a/new/file";
int create_flags = 0;
unifyfs_gfid gfid = UNIFYFS_INVALID_GFID;
int rc = unifyfs_create(fshdl, create_flags, filename, &gfid);
```

Existing files can be opened by any client process using `unifyfs_open()`.

Listing 8: UnifyFS file use

```
const char* filename = "/my/unifyfs/namespace/an/existing/file";
unifyfs_gfid gfid = UNIFYFS_INVALID_GFID;
int access_flags = O_RDWR;
int rc = unifyfs_open(fshdl, access_flags, filename, &gfid);
```

When no longer required, files can be deleted using `unifyfs_remove()`.

Listing 9: UnifyFS file removal

```
const char* filename = "/my/unifyfs/namespace/an/existing/file";
int rc = unifyfs_remove(fshdl, filename);
```

10.4.3 Batched File I/O

File I/O operations in the library API use a batched request interface similar to POSIX `lio_listio()`. A client application dispatches an array of I/O operation requests, where each request identifies the target file `gfid`, the operation type (e.g., read, write, or truncate), and associated operation parameters. Upon successful dispatch, the operations will be executed by UnifyFS in an asynchronous manner that allows the client to overlap other computation with I/O. The client application must then explicitly wait for completion of the requests in the batch. After an individual request has been completed (or canceled by the client), the request's operation results can be queried.

When dispatching a set of requests that target the same file, there is an order imposed on the types of operations. First, all read operations are processed, followed by writes, then truncations, and finally synchronization operations. Note that this means a read request will not observe any data written in the same batch.

A simple use case for batched I/O is shown below, where the client dispatches a batch of requests including several rank-strided write operations followed by a metadata sync to make those writes visible to other clients, and then immediately waits for completion of the entire batch.

Listing 10: Synchronous Batched I/O

```
/* write and sync file metadata */
size_t n_chks = 10;
size_t chunk_size = 1048576;
size_t block_size = chunk_size * total_ranks;
size_t n_reqs = n_chks + 1;
unifyfs_io_request my_reqs[n_reqs];
for (size_t i = 0; i < n_chks; i++) {
    my_reqs[i].op = UNIFYFS_IOREQ_OP_WRITE;
    my_reqs[i].gfid = gfid;
    my_reqs[i].nbytes = chunk_size;
    my_reqs[i].offset = (off_t)((i * block_size) + (my_rank * chunk_size));
    my_reqs[i].user_buf = my_databuf + (i * chksize);
}
my_reqs[n_chks].op = UNIFYFS_IOREQ_OP_SYNC_META;
my_reqs[n_chks].gfid = gfid;

rc = unifyfs_dispatch_io(fshdl, n_reqs, my_reqs);
if (rc == UNIFYFS_SUCCESS) {
    int waitall = 1;
    rc = unifyfs_wait_io(fshdl, n_reqs, my_reqs, waitall);
    if (rc == UNIFYFS_SUCCESS) {
        for (size_t i = 0; i < n_reqs; i++) {
```

(continues on next page)

(continued from previous page)

```

        assert(my_reqs[i].result.error == 0);
    }
}

```

10.4.4 Batched File Transfers

File transfer operations in the library API also use a batched request interface. A client application dispatches an array of file transfer requests, where each request identifies the source and destination file paths and the transfer mode. Two transfer modes are currently supported:

1. COPY - Copy source file to destination path.
2. MOVE - Copy source file to destination path, then remove source file.

Upon successful dispatch, the transfer operations will be executed by UnifyFS in an asynchronous manner that allows the client to overlap other computation with I/O. The client application must then explicitly wait for completion of the requests in the batch. After an individual request has been completed (or canceled by the client), the request's operation results can be queried.

A simple use case for batched transfer is shown below, where the client dispatches a batch of requests and then immediately waits for completion of the entire batch.

Listing 11: Synchronous Batched File Transfers

```

/* move output files from UnifyFS to parallel file system */
const char* destfs_prefix = "/some/parallel/filesystem/location";
size_t n_files = 3;
unifyfs_transfer_request my_reqs[n_files];
char src_file[PATHLEN_MAX];
char dst_file[PATHLEN_MAX];
for (int i = 0; i < (int)n_files; i++) {
    snprintf(src_file, sizeof(src_file), "%s/file.%d", unifyfs_prefix, i);
    snprintf(dst_file, sizeof(src_file), "%s/file.%d", destfs_prefix, i);
    my_reqs[i].src_path = strdup(src_file);
    my_reqs[i].dst_path = strdup(dst_file);
    my_reqs[i].mode = UNIFYFS_TRANSFER_MODE_MOVE;
    my_reqs[i].use_parallel = 1;
}

rc = unifyfs_dispatch_transfer(fshdl, n_files, my_reqs);
if (rc == UNIFYFS_SUCCESS) {
    int waitall = 1;
    rc = unifyfs_wait_transfer(fshdl, n_files, my_reqs, waitall);
    if (rc == UNIFYFS_SUCCESS) {
        for (int i = 0; i < (int)n_files; i++) {
            assert(my_reqs[i].result.error == 0);
        }
    }
}
}

```

10.4.5 More Examples

Additional examples demonstrating use of the library API can be found in the unit tests (see [api-unit-tests](#)).

First of all, thank you for taking the time to contribute!

By using the following guidelines, you can help us make UnifyFS even better.

11.1 Getting Started

11.1.1 Get UnifyFS

You can build and run UnifyFS by following *these instructions*.

11.1.2 Getting Help

To contact the UnifyFS team, send an email to the [mailing list](#).

11.2 Reporting Bugs

Please contact us via the [mailing list](#) if you are not certain that you are experiencing a bug.

You can open a new issue and search existing issues using the [issue tracker](#).

If you run into an issue, please search the [issue tracker](#) first to ensure the issue hasn't been reported before. Open a new issue only if you haven't found anything similar to your issue.

Important: When opening a new issue, please include the following information at the top of the issue:

- What operating system (with version) you are using
- The UnifyFS version you are using

- Describe the issue you are experiencing
 - Describe how to reproduce the issue
 - Include any warnings or errors
 - Apply any appropriate labels, if necessary
-

When a new issue is opened, it is not uncommon for developers to request additional information.

In general, the more detail you share about a problem the more quickly a developer can resolve it. For example, providing a simple test case is extremely helpful. Be prepared to work with the developers investigating your issue. Your assistance is crucial in providing a quick solution.

11.3 Suggesting Enhancements

Open a new issue in the [issue tracker](#) and describe your proposed feature. Why is this feature needed? What problem does it solve? Be sure to apply the *enhancement* label to your issue.

11.4 Pull Requests

- All pull requests must be based on the current *dev* branch and apply without conflicts.
 - Please attempt to limit pull requests to a single commit which resolves one specific issue.
 - Make sure your commit messages are in the correct format. See the *Commit Message Format* section for more information.
 - When updating a pull request, squash multiple commits by performing a *rebase* (squash).
 - For large pull requests, consider structuring your changes as a stack of logically independent patches which build on each other. This makes large changes easier to review and approve which speeds up the merging process.
 - Try to keep pull requests simple. Simple code with comments is much easier to review and approve.
 - Test cases should be provided when appropriate.
 - If your pull request improves performance, please include some benchmark results.
 - The pull request must pass all regression tests before being accepted.
 - All proposed changes must be approved by a UnifyFS project member.
-

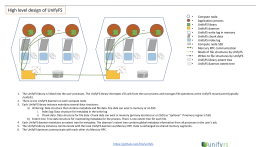
11.5 Testing

All help is appreciated! If you're in a position to run the latest code, consider helping us by reporting any functional problems, performance regressions, or other suspected issues. By running the latest code on a wide range of realistic workloads, configurations, and architectures we're better able to quickly identify and resolve issues.

11.6 Documentation

Here is our current documentation of how the internals of UnifyFS function for several basic operations.

UnifyFS Developer's Documentation



[Download slides.](#)

As UnifyFS is continually improved and updated, it is easy for documentation to become out-of-date. Any contributions to the documentation, no matter how small, is always greatly appreciated. If you are not in a position to update the documentation yourself, please notify us via the [mailing list](#) of anything you notice that needs to be changed.

12.1 Coding Conventions

UnifyFS follows the [Linux kernel coding style](#) except that code is indented using four spaces per level instead of tabs. Please run `make checkstyle` to check your patch for style problems before submitting it for review.

12.1.1 Styling Code

The `astyle` tool can be used to apply much of the required code styling used in the project.

Listing 1: To apply style to the source file `foo.c`:

```
astyle --options=scripts/unifyfs.astyle foo.c
```

The `unifyfs.astyle` file specifies the options used for this project. For a full list of available `astyle` options, see <http://astyle.sourceforge.net/astyle.html>.

12.1.2 Verifying Style Checks

To check that uncommitted changes meet the coding style, use the following command:

```
git diff | ./scripts/checkpatch.sh
```

Tip: This command will only check specific changes and additions to files that are already tracked by git. Run the command `git add -N [<untracked_file>...]` first in order to style check new files as well.

12.2 Commit Message Format

Commit messages for new changes must meet the following guidelines:

- In 50 characters or less, provide a summary of the change as the first line in the commit message.
- A body which provides a description of the change. If necessary, please summarize important information such as why the proposed approach was chosen or a brief description of the bug you are resolving. Each line of the body must be 72 characters or less.

An example commit message for new changes is provided below.

```
Capitalized, short (50 chars or less) summary
```

```
More detailed explanatory text, if necessary. Wrap it to about 72
characters or so. In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body. The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.
```

```
Write your commit message in the imperative: "Fix bug" and not "Fixed bug"
or "Fixes bug." This convention matches up with commit messages generated
by commands like git merge and git revert.
```

```
Further paragraphs come after blank lines.
```

- ```
- Bullet points are okay
- Typically a hyphen or asterisk is used for the bullet, followed by a
 single space, with blank lines in between, but conventions vary here
- Use a hanging indent
```



We can never have enough testing. Any additional tests you can write are always greatly appreciated.

## 13.1 Unit Tests

### 13.1.1 Implementing Tests

The UnifyFS Test Suite uses the [Test Anything Protocol \(TAP\)](#) and the Automake test harness. This test suite has two types of TAP tests (shell scripts and C) to allow for testing multiple aspects of UnifyFS.

#### Shell Script Tests

Test cases in shell scripts are implemented with `sharness`, which is included in the UnifyFS source distribution. See the file `sharness.sh` for all available test interfaces. UnifyFS-specific `sharness` code is implemented in scripts in the directory `sharness.d`. Scripts in `sharness.d` are primarily used to set environment variables and define convenience functions. All scripts in `sharness.d` are automatically included when your script sources `sharness.sh`.

The most common way to implement a test case with `sharness` is to use the `test_expect_success()` function. Your script must first set a test description and source the `sharness` library. After all tests are defined, your script should call `test_done()` to print a summary of the test run.

Test cases that demonstrate known breakage should use the `sharness` function `test_expect_failure()` to alert developers about the problem without causing the overall test suite to fail. Failing test cases should be tracked with [github issues](#).

Here is an example of a `sharness` test:

```
1 #!/bin/sh
2
3 test_description="My awesome test cases"
4
5 . $(dirname $0)/sharness.sh
```

(continues on next page)

(continued from previous page)

```

6
7 test_expect_success "Verify some critical invariant" '
8 test 1 -eq 1
9
10
11 test_expect_failure "Prove this someday" '
12 test "P" == "NP"
13
14
15 # Various tests available to use inside test_expect_success/failure
16 test_expect_success "Show various available tests" '
17 test_path_is_dir /somedir
18 test_must_fail test_dir_is_empty /somedir
19 test_path_is_file /somedir/somefile
20
21
22 # Use test_set_prereq/test_have_prereq to conditionally skip tests
23 [[-n $(which h5cc 2>/dev/null)]] && test_set_prereq HAVE_HDF5
24 if test_have_prereq HAVE_HDF5; then
25 # run HDF5 tests
26 fi
27
28 # Can also check for prereq in individual test
29 test_expect_success HAVE_HDF5 "Run HDF5 test" '
30 # Run HDF5 test
31
32
33 test_done

```

## C Program Tests

C programs use the `libtap` library to implement test cases. All available testing functions are viewable in the `libtap README`. Convenience functions common to test cases written in C are implemented in the library `lib/testutil.c`. If your C program needs to use environment variables set by `sharness`, it can be wrapped in a shell script that first sources `sharness.d/00-test-env.sh` and `sharness.d/01-unifyfs-settings.sh`. Your wrapper shouldn't normally source `sharness.sh` itself because the TAP output from `sharness` might conflict with that from `libtap`.

The most common way to implement a test with `libtap` is to use the `ok()` function. TODO test cases that demonstrate known breakage are surrounded by the `libtap` library calls `todo()` and `end_todo()`.

Here are some examples of `libtap` tests:

```

1 #include "t/lib/tap.h"
2 #include "t/lib/testutil.h"
3 #include <string.h>
4
5 int main(int argc, char *argv[])
6 {
7 int result;
8
9 result = (1 == 1);
10 ok(result, "1 equals 1: %d", result);
11
12 /* Or put a function call directly in test */
13 ok(somefunc() == 42, "somefunc() returns 42");

```

(continues on next page)

(continued from previous page)

```

14 ok(somefunc() == -1, "somefunc() should fail");
15
16 /* Use pass/fail for more complex code paths */
17 int x = somefunc();
18 if (x > 0) {
19 pass("somefunc() returned a valid value");
20 } else {
21 fail("somefunc() returned an invalid value");
22 }
23
24 /* Use is/isnt for string comparisions */
25 char buf[64] = {0};
26 ok(fread(buf, 12, 1, fd) == 1, "read 12 bytes into buf");
27 is(buf, "hello world", "buf is \"hello world\"");
28
29 /* Use cmp_mem to test first n bytes of memory */
30 char* a = "foo";
31 char* b = "bar";
32 cmp_mem(a, b, 3);
33
34 /* Use like/unlike to string match to a POSIX regex */
35 like("stranger", "^s.(r).*\\1$", "matches the regex");
36
37 /* Use dies_ok/lives_ok to test whether code causes an exit */
38 dies_ok({int x = 0/0;}, "divide by zero crashes");
39
40 /* Use todo for failing tests to be notified when they start passing */
41 todo("Prove this someday");
42 result = strcmp("P", "NP");
43 ok(result == 0, "P equals NP: %d", result);
44 end_todo;
45
46 /* Use skip/end_skip when a feature isn't implemented yet, or to
47 conditionally skip when a resource isn't available */
48 skip(TRUE, 2, "Reason for skipping tests");
49 ok(1);
50 ok(2);
51 end_skip;
52
53 #ifdef HAVE_SOME_FEATURE
54 ok(somefunc());
55 ok(someotherfunc());
56 #else
57 skip(TRUE, 2, "Don't have SOME_FEATURE");
58 end_skip;
59 #endif
60
61 done_testing();
62 }

```

**Tip:** Including the file and line number, as well as any useful variable values, in each test output can be very helpful when a test fails or needs to be debugged.

```

ok(somefunc() == 42, "%s:%d somefunc() returns 42", __FILE__,
__LINE__);

```

Also note that `errno` is only set when an error occurs and is never set back to 0 implicitly by the system. When testing for a failure and using `errno` as part of the test, setting `errno = 0` before the test will ensure a previous test error will not affect the current test. In the following example, we also assign `errno` to another variable `err` for use in constructing the test message. This is needed because the `ok()` macro may use system calls that set `errno`.

```
int err, rc;
errno = 0;
rc = systemcall();
err = errno;
ok(rc == -1 && err == ENOTTY,
 "%s:%d systemcall() should fail (errno=%d): %s",
 __FILE__, __LINE__, err, strerror(err));
```

---

### 13.1.2 Adding Tests

The UnifyFS Test Suite uses the [Test Anything Protocol \(TAP\)](#) and the Automake test harness. By convention, test scripts and programs that output TAP are named with a “.t” extension.

To add a new test case to the test harness, follow the existing examples in [t/Makefile.am](#). In short, add your test program to the list of tests in the `TESTS` variable. If it is a shell script, also add it to `check_SCRIPTS` so that it gets included in the source distribution tarball.

#### Test Suites

If multiple tests fit within the same category (i.e., tests for `creat` and `mkdir` both fall under tests for `sysio`) then create a test suite to run those tests. This makes it so less duplication of files and code is needed in order to create additional tests.

To create a new test suite, look at how it is currently done for the `sysio_suite` in [t/Makefile.am](#) and [t/sys/sysio\\_suite.c](#):

If you’re testing C code, you’ll need to use environment variables set by `sharness`.

- Create a shell script, `<####-suite-name>.t` (the `####` indicates the order in which they should be run by the tap-driver), that wraps your suite and sources [sharness.d/00-test-env.sh](#) and [sharness.d/01-unifyfs-settings.sh](#)
- Add this file to [t/Makefile.am](#) in the `TESTS` and `check_SCRIPTS` variables and add the name of the file (but with a .t extension) this script runs to the `libexec_PROGRAMS` variable

You can then create the test suite file and any tests to be run in this suite.

- Create a `<test_suite_name>.c` file (i.e., `sysio_suite.c`) that will contain the main function and `mpi` job that drives your suite
  - Mount `unifyfs` from this file
  - Call testing functions that contain the test cases (created in other files) in the order desired for testing, passing the mount point to those functions
- Create a `<test_suite_name>.h` file that declares the names of all the test functions to be run by this suite and `include` this in the `<test_suite_name>.c` file
- Create `<test_name>.c` files (i.e., `open.c`) that contains the testing function (i.e., `open_test(char* unifyfs_root)`) that houses the variables and `libtap` tests needed to test that individual function

- Add the function name to the `<test_suite_name>.h` file
- Call the function from the `<test_suite_name>.c` file

The source files and flags for the test suite are then added to the bottom of `t/Makefile.am`.

- Add the `<test_suite_name>.c` and `<test_suite_name>.h` files to the `<test_suite>_SOURCES` variable
- Add additional `<test_name>.c` files to the `<test_suite>_SOURCES` variable as they are created
- Add the associated flags for the test suite (if the suite is for testing wrappers, add a suite and flags for both a gotcha and a static build)

## Test Cases

For testing C code, test cases are written using the `libtap` library. See the *C Program Tests* section above on how to write these tests.

To add new test cases to any existing suite of tests:

1. Simply add the desired tests (order matters) to the appropriate `<test_name>.c` file

If the test cases needing to be written don't already have a file they belong in (i.e., testing a wrapper that doesn't have any tests yet):

1. Create a `<function_name>.c` file with a function called `<function_name>_test(char* unifyfs_root)` that contains the desired `libtap` test cases
2. Add the `<function_name>_test` to the corresponding `<test_suite_name>.h` file
3. Add the `<function_name>.c` file to the bottom of `t/Makefile.am` under the appropriate `<test_suite>_SOURCES` variable(s)
4. The `<function_name>_test` function can now be called from the `<test_suite_name>.c` file

### 13.1.3 Running the Tests

To manually run the UnifyFS unit test suite, simply run `make check` in a single-node allocation from inside the `t/` directory of wherever you built UnifyFS. E.g., if you built in a separate `build/` directory, then do:

```
$ cd build/t
$ make check
```

If on a system where jobs are launched on a separate compute node, then use your systems local MPI job launch command to run the unit tests:

```
$ cd build/t
$ srun -N1 -n1 make check
```

If changes are made to existing files in the test suite, the tests can be run again by simply doing `make clean` followed by another `make check`.

Individual tests may be run by hand. The test `0001-setup.t` should normally be run first to start the UnifyFS daemon. E.g., to run just the `0100-sysio-gotcha.t` tests, do:

```
$ make check TESTS='0001-setup.t 0100-sysio-gotcha.t 9010-stop-unifyfsd.t 9999-
↳cleanup.t'
```

**Note:** If you are using Spack to install UnifyFS then there are two ways to manually run these tests:

1. Upon your installation with Spack

```
spack install -v --test=root unifyfs
```

2. Manually from Spack's build directory

```
spack install --keep-stage unifyfs
```

```
spack cd unifyfs
```

```
cd spack-build/t
```

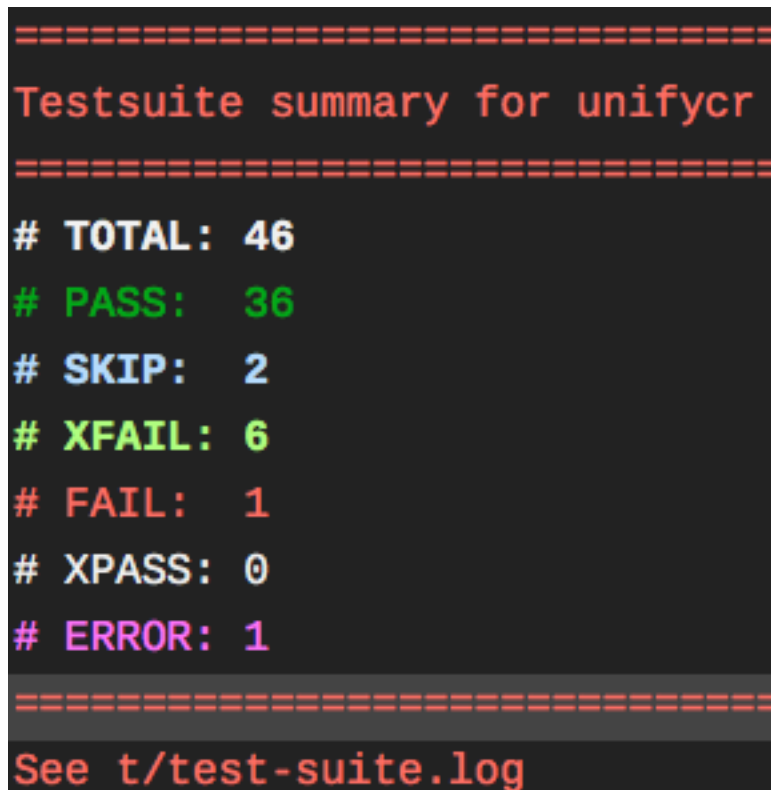
```
make check
```

---

The tests in <https://github.com/LLNL/UnifyFS/tree/dev/t> are run automatically by Travis CI along with the *style checks* when a pull request is created or updated. All pull requests must pass these tests before they will be accepted.

## Interpreting the Results

### TAP Output



```
=====
Testsuite summary for unifycr
=====
TOTAL: 46
PASS: 36
SKIP: 2
XFAIL: 6
FAIL: 1
XPASS: 0
ERROR: 1
=====
See t/test-suite.log
```

After a test runs, its result is printed out consisting of its status followed by its description and potentially a TODO/SKIP message. Once all the tests have completed (either from being run manually or by Travis CI), the overall results are printed out, as shown in the image on the right.

There are six possibilities for the status of each test: PASS, FAIL, XFAIL, XPASS, SKIP, and ERROR.

**PASS** The test had the desired result.

**FAIL** The test did not have the desired result. These must be fixed before any code changes can be accepted.

If a **FAIL** occurred after code had been added/changed then most likely a bug was introduced that caused the test to fail. Some tests may fail as a result of earlier tests failing. Fix bugs that are causing earlier tests to fail first as, once they start passing, subsequent tests are likely to start passing again as well.

**XFAIL** The test was expected to fail, and it did fail.

An **XFAIL** is created by surrounding a test with `todo()` and `end_todo`. These are tests that have identified a bug that was already in the code, but the cause of the bug hasn't been found/resolved yet. An optional message can be passed to the `todo("message")` call which will be printed after the test has run. Use this to explain how the test should behave or any thoughts on why it might be failing. An **XFAIL** is not meant to be used to make a failing test start "passing" if a bug was introduced by code changes.

**XPASS** A test passed that was expected to fail. These must be fixed before any code changes can be accepted.

The relationship of an **XPASS** to an **XFAIL** is the same as that of a **FAIL** to a **PASS**. An **XPASS** will typically occur when a bug causing an **XFAIL** has been fixed and the test has started passing. If this is the case, remove the surrounding `todo()` and `end_todo` from the failing test.

**SKIP** The test was skipped.

Tests are skipped because what they are testing hasn't been implemented yet, or they apply to a feature/variant that wasn't included in the build (i.e., HDF5). A **SKIP** is created by surrounding the test(s) with `skip(test, n, message)` and `end_skip` where the `test` is what determines if these tests should be skipped and `n` is the number of subsequent tests to skip. Remove these if it is no longer desired for those tests to be skipped.

**ERROR** A test or test suite exited with a non-zero status.

When a test fails, the containing test suite will exit with a non-zero status, causing an **ERROR**. Fixing any test failures should resolve the **ERROR**.

## Running the Examples

To run any of these examples manually, refer to the [Examples](#) documentation.

The UnifyFS [examples](#) are also being used as integration tests with continuous integration tools such as [Bamboo](#) or [GitLab CI](#).

---

## 13.2 Integration Tests

The UnifyFS [examples](#) are being used as integration tests with continuation integration tools such as [Bamboo](#) or [GitLab CI](#).

To run any of these examples manually, refer to the [Examples](#) documentation.

---

### 13.2.1 Configuration Variables

Along with the already provided [UnifyFS Configuration](#) options/environment variables, there are environment variables used by the integration testing suite that can also be set in order to change the default behavior.

### Key Variables

These environment variables can be set prior to sourcing the `t/ci/001-setup.sh` script and will affect how the overall integration suite operates.

#### UNIFYFS\_INSTALL

USAGE: UNIFYFS\_INSTALL=/path/to/dir/containing/UnifyFS/bin/directory

The full path to the directory containing the `bin/` and `libexec/` directories for your UnifyFS installation. Set this envvar to prevent the integration tests from searching for a UnifyFS installation automatically. Where the automatic search starts can be altered by setting the `$BASE_SEARCH_DIR` variable.

#### UNIFYFS\_CI\_NPROCS

USAGE: UNIFYFS\_CI\_NPROCS=<number-of-process-per-node>

The number of processes to use per node inside a job allocation. This defaults to 1 process per node. This can be adjusted if more processes are desired on multiple nodes or multiple processes are desired on a single node.

#### UNIFYFS\_CI\_TEMP\_DIR

USAGE: UNIFYFS\_CI\_TEMP\_DIR=/path/for/temporary/files/created/by/UnifyFS

Can be used as a shortcut to set `UNIFYFS_RUNSTATE_DIR` and `UNIFYFS_META_DB_PATH` to the same path. This envvar defaults to `UNIFYFS_CI_TEMP_DIR=${TMPDIR}/unifyfs.${USER}.${JOB_ID}`.

#### UNIFYFS\_CI\_LOG\_CLEANUP

USAGE: UNIFYFS\_CI\_LOG\_CLEANUP=yes|YES|no|NO

In the event `$UNIFYFS_LOG_DIR` has **not** been set, the logs will be put in `$SHARNESS_TRASH_DIRECTORY`, as set up by `sharness.sh`, and cleaned up automatically after the tests have run. The logs will be in a `<system-name>_<jobid>/` subdirectory. Should any tests fail, `sharness` does not clean up the trash directory for debugging purposes. Setting `UNIFYFS_CI_LOG_CLEANUP=no|NO` will move the `<system-name>_<jobid>/` logs directory to `$UNIFYFS_CI_DIR` (the directory containing the integration testing scripts) to allow them to persist even when all tests pass. This envvar defaults to `yes`.

---

**Note:** Setting `$UNIFYFS_LOG_DIR` will put all created logs in the designated path and will not clean them up.

---

#### UNIFYFS\_CI\_HOST\_CLEANUP

USAGE: UNIFYFS\_CI\_HOST\_CLEANUP=yes|YES|no|NO

After all tests have run, the nodes on which the tests were ran will automatically be cleaned up. This cleanup includes ensuring `unifyfsd` has stopped and deleting any files created by UnifyFS or its dependencies. Set `UNIFYFS_CI_HOST_CLEANUP=no|NO` to skip cleaning up. This envvar defaults to `yes`.

---

**Note:** `PDSH` is required for cleanup and cleaning up is simply skipped if not found.

---



## UNIFYFS\_CI\_CLEANUP

USAGE: UNIFYFS\_CI\_CLEANUP=yes|YES|no|NO

Setting this to no|NO sets both `$CI_LOG_CLEANUP` and `$UNIFYFS_CI_HOST_CLEANUP` to no|NO.

## UNIFYFS\_CI\_TEST\_POSIX

USAGE: UNIFYFS\_CI\_TEST\_POSIX=yes|YES|no|NO

Determines whether any `<example-name>-posix` tests should be run since they require a real mountpoint to exist.

This envvar defaults to no. Setting this to yes will run the posix version of tests along with the regular tests. When `$UNIFYFS_MOUNTPOINT` is set to a existing directory, this option is set to no. This is to allow running the tests a first time with a fake mountpoint while the posix tests use an existing mountpoint. Then the regular tests can be run again using an existing mountpoint and the posix tests won't be run twice.

An example of testing a posix example can be see *below*.

---

**Note:** The posix mountpoint envvar, `UNIFYFS_CI_POSIX_MP`, is set to be located inside `$SHARNESS_TRASH_DIRECTORY` automatically and cleaned up afterwards. However, this envvar can be set before running the integration tests as well. If setting this, ensure that it is a shared file system that all allocated nodes can see.

---

## Additional Variables

After sourcing the `t/ci/001-setup.sh` script there will be additional variables available that may be useful when writing/adding additional tests.

## Directory Structure

File structure here is assuming UnifyFS was cloned to `$HOME`.

**UNIFYFS\_CI\_DIR** Directory containing the CI testing scripts. `$HOME/UnifyFS/t/ci/`

**SHARNESS\_DIR** Directory containing the base sharness scripts. `$HOME/UnifyFS/t/`

**UNIFYFS\_SOURCE\_DIR** Directory containing the UnifyFS source code. `$HOME/UnifyFS/`

**BASE\_SEARCH\_DIR** Parent directory containing the UnifyFS source code. Starting place to auto search for UnifyFS install when `$UNIFYFS_INSTALL` isn't provided. `$HOME/`

## Executable Locations

**UNIFYFS\_BIN** Directory containing `unifyfs` and `unifyfsd`. `$UNIFYFS_INSTALL/bin`

**UNIFYFS\_EXAMPLES** Directory containing the compiled `examples`. `$UNIFYFS_INSTALL/libexec`

## Resource Managers

**JOB\_RUN\_COMMAND** The base MPI job launch command established according to the detected resource manager, number of allocated nodes, and `$UNIFYFS_CI_NPROCS`.

The LSF variables below will also affect the default version of this command when using that resource manager.

**JOB\_RUN\_ONCE\_PER\_NODE** MPI job launch command to only run a single process on each allocated node established according to the detected resource manager.

**JOB\_ID** The ID assigned to the current CI job as established by the detected resource manager.

## LSF

Additional variables used by the LSF resource manager to determine how jobs are launched with `$JOB_RUN_COMMAND`. These can also be set prior to sourcing the `t/ci/001-setup.sh` script and will affect how the integration tests run.

**UNIFYFS\_CI\_NCORES** Number of cores-per-resource-set to use. Defaults to 20.

**UNIFYFS\_CI\_NRS\_PER\_NODE** Number of resource-sets-per-node to use. Defaults to 1.

**UNIFYFS\_CI\_NRES\_SETS** Total number of resource sets to use. Defaults to  $(\text{number\_of\_nodes}) * (\text{UNIFYFS\_CI\_NRS\_PER\_NODE})$ .

## Misc

**KB**  $2^{10}$ .

**MB**  $2^{20}$ .

**GB**  $2^{30}$ .

---

## 13.2.2 Running the Tests

**Attention:** UnifyFS's integration test suite requires MPI and currently only supports `srun` and `jsrun` MPI launch commands. Changes are coming to support `mpirun`.

UnifyFS's integration tests are primarily set up to be run all as one suite. However, they can be run individually if desired.

The testing scripts in `t/ci` depend on `sharness`, which is set up in the containing `t/` directory. These tests will not function properly if moved or if they cannot find the `sharness` files.

Whether running all tests or individual tests, first make sure you have either interactively allocated nodes or are submitting a batch job to run them.

Make sure all *dependencies* are installed and loaded.

---

**Note:** In order to run the the integration tests from a `Spack` installation of UnifyFS, you'll need to tell `Spack` to use a different location for staging builds in order to have the source files available from inside an allocation.

Open your `Spack` config file

---

```
spack config edit config
```

and provide a path that is visible during job allocations:

```
config:
 build_stage:
 - /visible/path/from/all/allocated/nodes
 # or build directly inside Spack's install directory
 - $spack/var/spack/stage
```

Then make sure to include the `--keep-stage` option when installing:

```
spack install --keep-stage unifyfs
```

## Running All Tests

To run all of the tests, simply run `./RUN_CI_TESTS.sh`.

```
$./RUN_CI_TESTS.sh
```

or

```
$ prove -v RUN_CI_TESTS.sh
```

## Running Individual Tests

In order to run individual tests, the testing functions and variables need to be set up first and then the UnifyFS server needs to be started.

First source the `t/ci/001-setup.sh` script whereafter `sharness` will change directories to the `$SHARNESS_TRASH_DIRECTORY`. To account for this, source `002-start-server.sh` and each desired test script after that prefixed with `$UNIFYFS_CI_DIR/`. When finished, source the `990-stop-server.sh` script last to stop the server and clean up.

```
$. ./001-setup.sh
$. $UNIFYFS_CI_DIR/002-start-server.sh
$. $UNIFYFS_CI_DIR/100-writeread-tests.sh
$. $UNIFYFS_CI_DIR/990-stop-server.sh
```

The various CI test suites can be run multiple times with different behaviors. These behaviors are continually being extended. The `-h|--help` option for each script can show what alternate behaviors are currently implemented along with additional information for that particular suite.

```
[prompt]$./100-writeread-tests.sh --help
Usage: 100-writeread-tests.sh [options...]

options:
 -h, --help print help message
 -M, --mpio use MPI-IO instead of POSIX I/O
 -x, --shuffle read different data than written
```

### 13.2.3 Adding New Tests

In order to add additional tests for different workflows, create a script after the fashion of `t/ci/100-writeread-tests.sh` where the prefixed number indicates the desired order for running the tests. Then source that script in `t/ci/RUN_CI_TESTS.sh` in the desired order. The different test suite scripts themselves can also be edited to add/change the number, types, and various behaviors each suite will execute.

Just like the helpers functions found in `sharness.d`, there are continuous integration helper functions (see *below* for more details) available in `t/ci/ci-functions.sh`. These exist to help make adding new tests as simple as possible.

One particularly useful function is `unify_run_test()`. Currently, this function is set up to work for the *write*, *read*, *writeread*, and *checkpoint-restart* examples. This function sets up the MPI job run command and default options as well as any default arguments wanted by all examples. See *below* for details.

#### Testing Helper Functions

There are helper functions available in `t/ci/ci-functions.sh` that can make running and testing the examples much easier. These may get adjusted over time to accommodate other examples, or additional functions may need to be written. Some of the main helper functions that might be useful for running examples are:

##### `unify_run_test()`

USAGE: `unify_run_test app_name "app_args" [output_variable_name]`

Given a example application name and application args, this function runs the example with the appropriate MPI runner and args. This function is meant to make running the *cr*, *write*, *read*, and *writeread* examples as easy as possible.

The `build_test_command()` function is called by this function which automatically sets any options that are always wanted (`-vkfo` as well as `-U` and the appropriate `-m` if *posix* test or not). The `stderr` output file is also created (based on the filename that is autogenerated) and the appropriate option is set for the MPI job run command.

Args that can be passed in are (`[-pncbx][-A|-M|-N|-P|-S|-V]`). All other args (see *Running the Examples*) are set automatically, including the outfile and filename (which are generated based on the input `$app_name` and `$app_args`).

The third parameter is an optional “pass-by-reference” parameter that can contain the variable name for the resulting output to be stored in, allowing this function to be used in one of two ways:

Listing 1: Using command substitution

```
app_output=$(unify_run_test $app_name "$app_args")
```

or

Listing 2: Using a “pass-by-reference” variable

```
unifyfs_run_test $app_name "$app_args" app_output
```

This function returns the return code of the executed example as well as the output produced by running the example.

---

**Note:** If `unify_run_test()` is simply called with only two arguments and without using command substitution, the resulting output will be sent to the standard output.

---

The results can then be tested with `sharness`:

```

basetest=write-read
runmode=static

app_name=${basetest}-${runmode}
app_args="-p n1 -n32 -c $((16 * $KB)) -b $MB

unify_run_test $app_name "$app_args" app_output
rc=$?
line_count=$(echo "$app_output" | wc -l)

test_expect_success "$app_name $app_args: (line_count=$line_count, rc=$rc)" '
 test $rc = 0 &&
 test $line_count = 8
'

```

### get\_filename()

USAGE: get\_filename app\_name app\_args [app\_suffix]

Builds and returns the filename with the provided suffix based on the input app\_name and app\_args.

The filename in \$UNIFYFS\_MOUNTPOINT will be given a .app suffix.

This allows tests to get what the filename will be in advance if called from a test suite. This can be used for posix tests to ensure the file showed up in the mount point, as well as for read, cp, stat tests that potentially need the filename from a previous test prior to running.

Error logs and outfiles are also created with this filename, with a .err or .out suffix respectively, and placed in the logs directory.

Returns a string with the spaces removed and hyphens replaced by underscores.

```

get_filename write-static "-p n1 -n 32 -c 1024 -b 1048576" ".app"
write-static_pn1_n32_c1KB_b1MB.app

```

Some uses cases may be:

- posix tests where the file existence is checked for after a test was run
- read, cp, or stat tests where an already existing filename from a prior test might be needed

For example:

```

basetest=write-read
runmode=posix

app_name=${basetest}-${runmode}
app_args="-p nn -n32 -c $((16 * $KB)) -b $MB

unify_run_test $app_name "$app_args" app_output
rc=$?
line_count=$(echo "$app_output" | wc -l)
filename=$(get_filename $app_name "$app_args" ".app")

test_expect_success POSIX "$app_name $app_args: (line_count=$line_count, rc=$rc)" '
 test $rc = 0 &&
 test $line_count = 8 &&
 test_path_has_file_per_process $UNIFYFS_CI_POSIX_MP $filename
'

```

## Additional Functions

There are other convenience functions used by that may be helpful in writing/adding tests are also found in `t/ci/ci-functions.sh`:

**find\_executable()** USAGE: `find_executable abs_path *file_name|*path/file_name [prune_path]`

Locate the desired executable file when provided an absolute path of where to start searching, the name of the file with an optional preceding path, and an optional `prune_path`, or path to omit from the search.

Returns the path of the first executable found with the given name and optional prefix.

**elapsed\_time()** USAGE: `elapsed_time start_time_in_seconds end_time_in_seconds`

Calculates the elapsed time between two given times.

Returns the elapsed time formatted as HH:MM:SS.

**format\_bytes()** USAGE: `format_bytes int`

Returns the input bytes formatted as KB, MB, or GB (1024 becomes 1KB).

## Sharness Helper Functions

There are also additional sharness functions for testing the examples available when `t/ci/ci-functions.sh` is sourced. These are to be used with `sharness` for testing the results of running the examples with or without using the *Example Helper Functions*.

### **process\_is\_running()**

USAGE: `process_is_running process_name seconds_before_giving_up`

Checks if a process with the given name is running on every host, retrying up to a given number of seconds before giving up. This function overrides the `process_is_running()` function used by the UnifyFS unit tests. The primary difference being that this function checks for the process on every host.

Expects two arguments:

- \$1 - Name of a process to check for
- \$2 - Number of seconds to wait before giving up

```
test_expect_success "unifyfsd is running" '
 process_is_running unifyfsd 5
'
```

### **process\_is\_not\_running()**

USAGE: `process_is_not_running process_name seconds_before_giving_up`

Checks if a process with the given name is not running on every host, retrying up to a given number of seconds before giving up. This function overrides the `process_is_not_running()` function used by the UnifyFS unit tests. The primary difference being that this function checks that the process is not running on every host.

Expects two arguments:

- \$1 - Name of a process to check for

- \$2 - Number of seconds to wait before giving up

```
test_expect_success "unifyfsd is not running" '
 process_is_not_running unifyfsd 5
'
```

### test\_path\_is\_dir()

USAGE: test\_path\_is\_dir dir\_name [optional]

Checks that a directory with the given name exists and is accessible from each host. Does NOT need to be a shared directory. This function overrides the `test_path_is_dir()` function in `sharness.sh`, the primary difference being that this function checks for the dir on every host in the allocation.

Takes once argument with an optional second:

- \$1 - Path of the directory to check for
- \$2 - Can be given to provide a more precise diagnosis

```
test_expect_success "$dir_name is an existing directory" '
 test_path_is_dir $dir_name
'
```

### test\_path\_is\_shared\_dir()

USAGE: test\_path\_is\_shared\_dir dir\_name [optional]

Check if same directory (actual directory, not just name) exists and is accessible from each host.

Takes once argument with an optional second:

- \$1 - Path of the directory to check for
- \$2 - Can be given to provide a more precise diagnosis

```
test_expect_success "$dir_name is a shared directory" '
 test_path_is_shared_dir $dir_name
'
```

### test\_path\_has\_file\_per\_process()

USAGE: test\_path\_has\_file\_per\_process dir\_path file\_name [optional]

Check if the provided directory path contains a file-per-process of the provided file name. Assumes the directory is a shared directory.

Takes two arguments with an optional third:

- \$1 - Path of the shared directory to check for the files
- \$2 - File name without the appended process number
- \$3 - Can be given to provided a more precise diagnosis

```
test_expect_success "$dir_name has file-per-process of $file_name" '
 test_path_has_file_per_process $dir_name $file_name
'
```

There are other helper functions available as well, most of which are being used by the test suite itself. Details on these functions can be found in their comments in [t/ci/ci-functions.sh](#).



**Warning:** This document is out-of-date as the process for generating *unifyfs\_list.txt* has bugs which causes the generation of *gotcha\_map\_unifyfs\_list.h* to have bugs as well. More information on this can be found in [issue #172](#).

An updated guide and scripts needs to be created for writing and adding new wrappers to UnifyFS.

The files in `client/check_fns/` folder help manage the set of wrappers that are implemented. In particular, they are used to enable a tool that detects I/O routines used by an application that are not yet supported in UnifyFS. They are also used to generate the code required for GOTCHA.

- `fakechroot_list.txt` - lists I/O routines from fakechroot
- `gnulibc_list.txt` - I/O routines from libc
- `cstdio_list.txt` - I/O routines from stdio
- `posix_list.txt` - I/O routines in POSIX
- `unifyfs_list.txt` - list of wrappers in UnifyFS
- `unifyfs_unsupported_list.txt` - list of wrappers in UnifyFS that are implemented, but not supported

---

## 14.1 unifyfs\_check\_fns Tool

This tool identifies the set of I/O calls used in an application by running `nm` on the executable. It reports any I/O routines used by the app, which are not supported by UnifyFS. If an application uses an I/O routine that is not supported, it likely cannot use UnifyFS. If the tool does not report unsupported wrappers, the app may work with UnifyFS but it is not guaranteed to work.

```
unifyfs_check_fns <executable>
```

## 14.2 Building the GOTCHA List

The `gotcha_map_unifyfs_list.h` file contains the code necessary to wrap I/O functions with GOTCHA. This is generated from the `unifyfs_list.txt` file by running the following command:

```
python unifyfs_translate.py unifyfs_list
```

---

## 14.3 Commands to Build Files

### 14.3.1 fakechroot\_list.txt

The `fakechroot_list.txt` file lists I/O routines implemented in fakechroot. This list was generated using the following commands:

```
git clone https://github.com/fakechroot/fakechroot.git fakechroot.git
cd fakechroot.git/src
ls *.c > fakechroot_list.txt
```

### 14.3.2 glibc\_list.txt

The `glibc_list.txt` file lists I/O routines available in libc. This list was written by hand using information from [http://www.gnu.org/software/libc/manual/html\\_node/L\\_002fO-Overview.html#L\\_002fO-Overview](http://www.gnu.org/software/libc/manual/html_node/L_002fO-Overview.html#L_002fO-Overview).

### 14.3.3 cstdio\_list.txt

The `cstdio_list.txt` file lists I/O routines available in libstdio. This list was written by hand using information from <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf>.

### 14.3.4 unifyfs\_list.txt

The `unifyfs_list.txt` file specifies the set of wrappers in UnifyFS. Most but not all such wrappers are supported. The command to build unifyfs list:

### 14.3.5 unifyfs\_unsupported\_list.txt

The `unifyfs_unsupported_list.txt` file specifies wrappers that are in UnifyFS, but are known to not actually be supported. This list is written by hand.

---

## Adding RPC Functions With Margo Library

---

In this section, we describe how to add an RPC function using the Margo library API.

---

**Note:** This uses the *unifyfs\_mount\_rpc* as an example RPC function to follow throughout.

---

### 15.1 Common

1. Define structs for the input and output parameters of your RPC handler.

The struct definition macro *MERCURY\_GEN\_PROC()* is used to define both input and output parameters. For client-server RPCs, the definitions should be placed in *common/src/unifyfs\_client\_rpcs.h*, while server-server RPC structs are defined in *common/src/unifyfs\_server\_rpcs.h*.

The input parameters struct should contain all values the client needs to pass to the server handler function. The output parameters struct should contain all values the server needs to pass back to the client upon completion of the handler function. The following shows the input and output structs for *unifyfs\_mount\_rpc*.

Passing some types can be an issue. Refer to the Mercury documentation for supported types: '<https://mercury-hpc.github.io/documentation/>' (look under Predefined Types). If your type is not predefined, you will need to either convert it to a supported type or write code to serialize/deserialize the input/output parameters. Phil Carns said he has starter code for this, since much of the code is similar.

### 15.2 Server

1. Implement the RPC handler function for the server.

This is the function that will be invoked on the client and executed on the server. Client-server RPC handler functions are implemented in *server/src/unifyfs\_cmd\_handler.c*, while server-server RPC handlers go in *server/src/unifyfs\_service\_manager.c*.

All the RPC handler functions follow the same prototype, which is passed a Mercury handle as the only argument. The handler function should use *margo\_get\_input()* to retrieve the input parameters struct provided by the client. After the RPC handler finishes its intended action, it replies using *margo\_respond()*, which takes the handle and output parameters struct as arguments. Finally, the handler function should release the input struct using *margo\_free\_input()*, and the handle using *margo\_destroy()*. See the existing RPC handler functions for more info.

After implementing the handler function, place the Margo RPC handler definition macro immediately following the function, for example:

```
DEFINE_MARGO_RPC_HANDLER(unifyfs_mount_rpc
```

2. Register the server RPC handler with margo.

In *server/src/margo\_server.c*, update the client-server RPC registration function *register\_client\_server\_rpcs()* to include a registration macro for the new RPC handler, for example:

## 15.3 Client

1. Add a Mercury id for the RPC handler to the client RPC context.

In *client/src/margo\_client.h*, update the *ClientRpcIds* structure to add a new *hg\_id\_t* variable to hold the RPC handler id.

2. Register the RPC handler with Margo.

In *client/src/margo\_client.c*, update *register\_client\_rpcs()* to register the RPC handler and store its Mercury id in the newly defined *ClientRpcIds* variable. .. code-block:: C 

```
client_rpc_context->rpcs.mount_id = MARGO_REGISTER(client_rpc_context->mid, "unifyfs_mount_rpc",
```

```
unifyfs_mount_in_t, unifyfs_mount_out_t, NULL);
```

When the client calls *MARGO\_REGISTER()* the last parameter is *NULL*. This is the RPC handler function that is only defined on the server.

3. Define and implement an invocation function that will execute the RPC.

The declaration should be placed in *client/src/margo\_client.h*, and the definition should go in *client/src/margo\_client.c*. .. code-block:: C 

```
int invoke_client_mount_rpc();
```

A handle for the RPC is obtained using *margo\_create()*, which takes the server address and the id of the RPC as parameters. The RPC is actually initiated using *margo\_forward()*, where the RPC handle and input struct are supplied. Use *margo\_get\_output()* to obtain the returned output parameters struct, and release it with *margo\_free\_output()*. Finally, *margo\_destroy()* is used to release the RPC handle. See the existing invocation functions for more info.

The general workflow for creating new RPC functions is the same if you want to invoke an RPC on the server, and execute it on the client. One difference is that you will have to pass *NULL* to the last parameter of *MARGO\_REGISTER()* on the server, and on the client the last parameter to *MARGO\_REGISTER()* will be the name of the RPC handler function. To execute RPCs on the client it needs to be started in Margo as a *SERVER*, and the server needs to know the address of the client where the RPC will be executed. The client has already been configured to do those two things, so the only change going forward is how *MARGO\_REGISTER()* is called depending on where the RPC is being executed (client or server).

### 16.1 Required

- Automake version 1.15 or later
- GOTCHA version 1.0.3
- Margo version 0.9.1 and its dependencies:
  - Argobots version 1.0.1
  - Mercury version 2.0.0
    - \* libfabric and/or bmi
  - JSON-C
- OpenSSL

---

**Important:** Margo uses pkg-config to ensure it compiles and links correctly with all of its dependencies' libraries. When building manually, you'll need to set the `PKG_CONFIG_PATH` environment variable and include in that variable the paths for the `.pc` files for Mercury, Argobots, and Margo separated by colons.

---

### 16.2 Optional

- `spath` for normalizing relative paths



# CHAPTER 17

---

## Indices and tables

---

- `genindex`
- `modindex`
- `search`