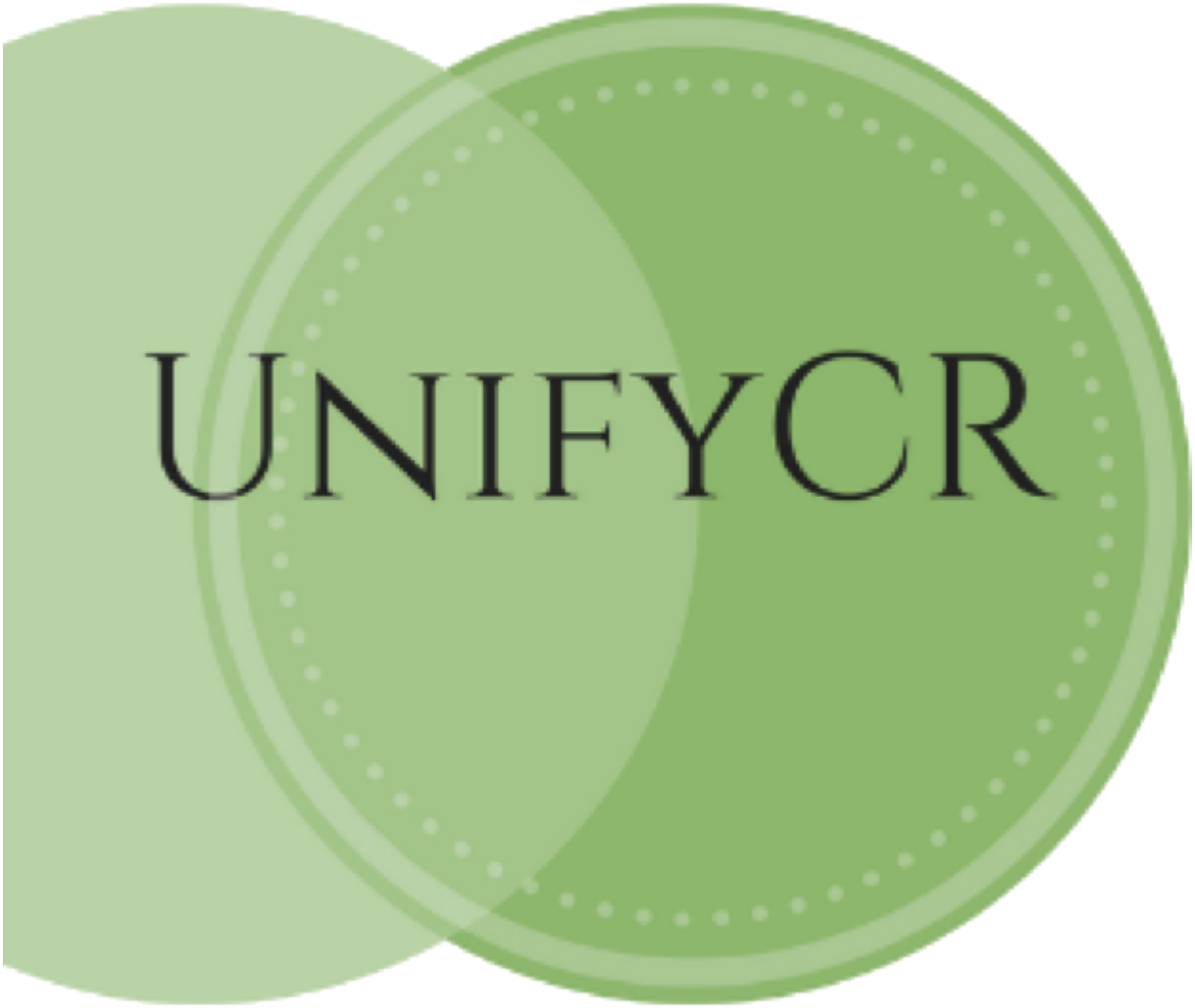



# UnifyFS Documentation

***Release 0.1***

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

**Oct 03, 2019**

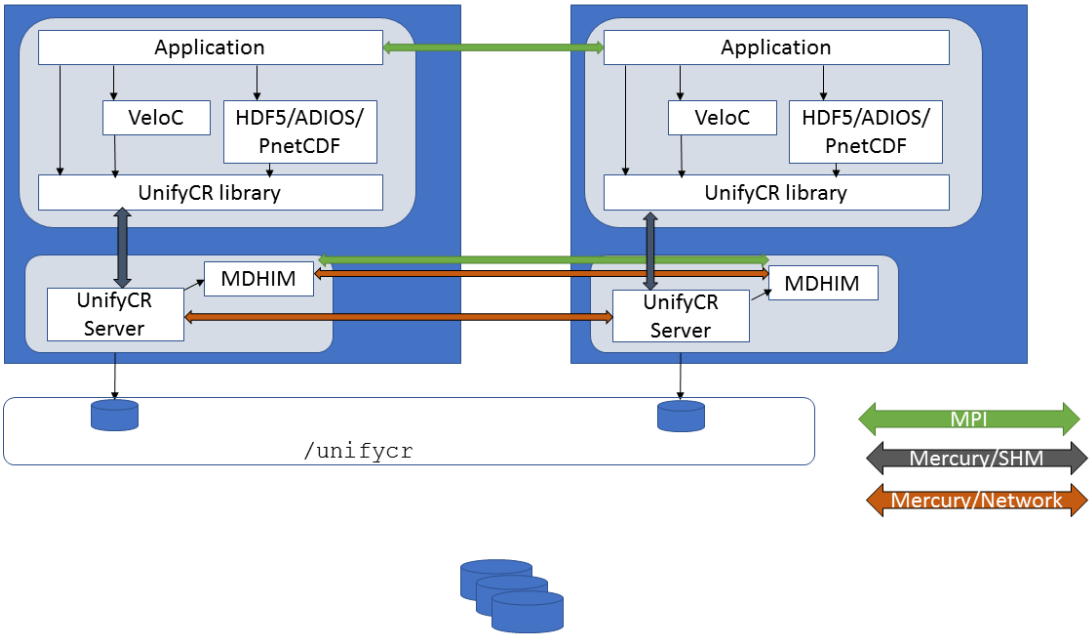
# User Guide

CHAPTER 1

# Overview

UnifyFS is a user level file system currently under active development. An application can use node-local storage as burst buffers for shared files. UnifyFS is designed to support both checkpoint/restart which is the most important I/O workload for HPC and other common I/O workloads as well. With UnifyFS, applications can write to fast, scalable, node-local burst buffers as easily as they do the parallel file system. This section will provide a high level design of UnifyFS. It will describe the UnifyFS library and the UnifyFS daemon.

## 1.1 High Level Design



UnifyFS will present a shared namespace (e.g., /unifyfs as a mount point) to all compute nodes in a users job allocation. There are two main components of UnifyFS: the UnifyFS library and the UnifyFS daemon. The UnifyFS library (also referred to as the UnifyFS client library) is linked into the user application and is responsible for intercepting I/O calls from the user application and then sending the I/O requests on to a UnifyFS server to be handled. The UnifyFS client library uses the ECP GOTCHA software as its primary mechanism for intercepting I/O calls. Each UnifyFS daemon (also referred to as a UnifyFS server daemon) runs as a daemon on a compute node in the users allocation. The UnifyFS server is responsible for handling the I/O requests from the UnifyFS library. On each compute node, there will be user application processes running as well as tool daemon processes. The user application is linked with the UnifyFS client library and a high-level I/O library, e.g. HDF5, ADIOS, or PnetCDF. The UnifyFS server daemon also runs on the compute node and is linked with the MDHIM library which is used for metadata services.

CHAPTER 2

# Definitions

In this section, we provide some useful definitions for terms used in this 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

CHAPTER 3

# Assumptions

In this section, we provide assumptions we make about the behavior of applications that use UnifyFS, and about how UnifyFS currently functions.

## 3.1 Application Behavior

- Workload supported is globally synchronous checkpointing.
- I/O occurs in write and read phases. Files are not read and written at the same time. There is some (good) amount of time between the two phases. For example, files are written during checkpoint phases and only read during recovery or restart.
- Processes on any node can read any byte in the file (not just local data), but the common case will be processes read only their local bytes.
- Assume general parallel I/O concurrency semantics where processes can write to the same offset concurrently. We assume the outcome of concurrent writes to the same offset or other conflicting concurrent accesses is undefined. For example, if a command in the job renames a file while the parallel application is writing to it, the outcome is undefined. It could be a failure or not, depending on timing.

## 3.2 Consistency Model

One key aspect of UnifyFS is the idea of "laminating" a file. After a file is laminated, it becomes "set in stone" and its data is accessible across all the nodes. Laminated files are permanently read-only, and cannot be modified in any way (but can be deleted). If the application process group fails before a file has been laminated, UnifyFS may delete the file.

The typical use case is to laminate your checkpoint files after they've been written. To laminate a file, first call fsync() to sync all your writes to the server, then call chmod() to remove all the write bits. Removing the write bits does the actual lamination. A typical checkpoint will look like this:

```
fp = fopen("checkpoint1.chk")
write(fp, <checkpoint data>)
fsync(fp)
fclose(fp)
chmod("checkpoint1.chk", 0444)
```

Future versions of UnifyFS may support different laminate semantics, such as laminate on close(), or laminate via an explicit API call.

We define the laminated consistency model to enable certain optimizations while supporting the perceived requirements of application checkpoints. Since remote processes are not permitted to read arbitrary bytes within the file until lamination, global exchange of file data and/or data index information can be buffered locally on each node until the point of lamination. Since file contents cannot change after lamination, aggressive caching may be used during the read-only phase with minimal locking. Since a file may be lost on application failure unless laminated, data redundancy schemes can be delayed until lamination.

Behavior before lamination:

- 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, the value is undefined.
- read: A process may read bytes it has written. Reading other bytes is invalid.
- rename: A process may rename a file.
- truncate: A process may truncate a file.
- unlink: A process may delete a file.

Behavior after lamination:

- 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.

## 3.3 File System Behavior

- The file system exists on node local storage only and is not persisted to stable storage like a parallel file system (PFS). Can be coupled with
- SymphonyFS or high level I/O or checkpoint library (VeloC) to move data to PFS periodically, or data can be moved manually
- Can be used with checkpointing libraries (VeloC) or I/O libraries to support shared files on burst buffers
- File system starts empty at job start. User job must populate the file system.
- Shared file system namespace across all compute nodes in a job, even if an application process is not running on all compute nodes
- Survives application termination and/or relaunch within a job
- Will transparently intercept system level I/O calls of applications and I/O libraries

## 3.4 System Characteristics

- There is some storage available for storing file data on a compute node, e.g. SSD or RAM disk
- We can run user-level daemon processes on compute nodes concurrently with a user application

CHAPTER 4

# Build & I/O Interception

In this section, we describe how to build UnifyFS with I/O interception.

**Note:** The current version of UnifyFS adopts the mdhim key-value store, which strictly requires:

"An MPI distribution that supports MPI_THREAD_MULTIPLE and per-object locking of critical sections (this excludes OpenMPI up to version 3.0.1, the current version as of this writing)"

as specified in the project github.

## 4.1 How to Build UnifyFS

To install all dependencies and set up your build environment, we recommend using the Spack package manager. If you already have Spack, make sure you have the latest release or if using a clone of their develop branch, ensure you have pulled the latest changes.

### 4.1.1 Building with Spack

These instructions assume that you do not already have a module system installed such as LMod, Dotkit, or Environment Modules. If your system already has Dotkit or LMod installed then installing the environment-modules package with Spack is unnecessary (so you can safely skip that step).

If you use Dotkit then replace `spack load` with `spack use`. First, install Spack if you don't already have it:

```
$ git clone https://github.com/spack/spack
$ ./spack/bin/spack install environment-modules
$ . spack/share/spack/setup-env.sh
```

Make use of Spack's shell support to automatically add Spack to your `PATH` and allow the use of the `spack` command.

Then install UnifyFS:

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

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

Table 1: UnifyFS Build Variants

| Variant | Command | Description |
|---|---|---|
| HDF5 | `spack install unifyfs+hdf5`<br>`spack install unifyfs+hdf5 ^hdf5~mpi` | Build with parallel HDF5<br>Build with serial HDF5 |
| Fortran | `spack install unifyfs+fortran` | Build with gfortran |
| NUMA | `spack install unifyfs+numa` | Build with NUMA |
| pmpi | `spack install unifyfs+pmpi` | Transparent mount/unmount |
| PMI | `spack install unifyfs+pmi` | Enable PMI2 build options |
| PMIx | `spack install unifyfs+pmix` | Enable PMIx build options |

**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). Type `spack spec -I unifyfs` before installing to see what Spack is going to do.

### 4.1.2 Building with Autotools

Download the latest UnifyFS release from the Releases page.

#### Building the Dependencies

UnifyFS requires MPI, LevelDB, GOTCHA(version 0.0.2), FlatCC, and Margo. References to these dependencies can be found *here*.

#### Build the Dependencies with Spack

Once Spack is installed on your system (see *above*), you can install just the dependencies for an easier manual installation of UnifyFS.

If you use Dotkit then replace `spack load` with `spack use`.

```
$ spack install leveldb
$ spack install gotcha@0.0.2
$ spack install flatcc
$ spack install margo
```

**Tip:** You can use `spack install --only=dependencies unifyfs` to install all of UnifyFS's 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.

Then to manually build UnifyFS:

```
$ spack load leveldb
$ spack load gotcha@0.0.2
$ spack load flatcc
$ spack load mercury
$ spack load argobots
$ spack load margo
$
$ ./autogen.sh
$ ./configure --prefix=/path/to/install
$ make
$ make install
```

**Note: Fortran Compatibility**

To build with gfortran compatibility, include the `--enable-fortran` configure option:

`./configure --prefix=/path/to/install/ --enable-fortran`

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.

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

### Build the Dependencies without Spack

For users who cannot use Spack, a bootstrap.sh script has been provided in order to make manual build and installation of dependencies easier. Simply run the script in the top level directory of the source code.

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

References to the UnifyFS dependencies can be found *here*.

After bootstrap.sh is finished building the dependencies, it will print out the commands you need to run to build UnifyFS. The commands look something like this:

```
$ export PKG_CONFIG_PATH=path/to/mercury/lib/pkgconfig:path/to/argobots/lib/
↪pkgconfig:path/to/margo/lib/pkgconfig
$ ./autogen.sh
$ ./configure --prefix=/path/to/install --with-gotcha=/path/to/gotcha --with-leveldb=/
↪path/to/leveldb  --with-flatcc=/path/to/flatcc
$ make
$ make install
```

**Note:** You may need to add the following to your configure line if it is not in your default path on a linux machine:

`--with-numa=$PATH_TO_NUMA`

This is needed to enable NUMA-aware memory allocation on Linux machines. Set the NUMA policy at runtime with `UNIFYFS_NUMA_POLICY = local | interleaved`, or set NUMA nodes explicitly with `UNIFYFS_USE_NUMA_BANK = <node no.>`

## 4.2 I/O Interception

POSIX calls can be intercepted via the methods described below.

### 4.2.1 Statically

Steps for static linking using –wrap:

To intercept I/O calls using a static link, you must add flags to your link line. UnifyFS installs a unifyfs-config script that returns 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`
```

### 4.2.2 Dynamically

Steps for dynamic linking using gotcha:

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

#### C

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

#### Fortran

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

# CHAPTER 5

# Mounting UnifyFS

In this section, we describe 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 Mounting

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

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

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

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

To use the UnifyFS filesystem a user will have to provide a path prefix. All file operations under the path prefix will be intercepted by the UnifyFS filesystem. For instance, to use UnifyFS on all path prefixes that begin with /tmp this would require a:

Listing 1: C

```
unifyfs_mount('/tmp', rank, rank_num, 0);
```

Listing 2: Fortran

```
call UNIFYFS_MOUNT('/tmp', rank, size, 0, ierr);
```

Where /tmp is the path prefix you want UnifyFS to intercept. The rank and rank number is the rank you are currently on, and the number of tasks you have running in your job. Lastly, the zero corresponds to the app id.

## 5.2 Unmounting

When you are finished using UnifyFS in your application, you should unmount.

Listing 3: C

```
if (rank == 0) {
    unifyfs_unmount();
}
```

Listing 4: Fortran

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

It is only necessary to call unmount once on rank zero.

CHAPTER 6

# UnifyFS Configuration

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

- System-wide configuration file: `/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 config file is installed in /etc by default. However, one can specify a custom location for the unifyfs.conf file with 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

## 6.1 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 |
|---|---|---|
| max_files | INT | maximum number of open files per client process |

Table 3: `[log]` section - logging settings

| Key | Type | Description |
|---|---|---|
| dir | STRING | path to directory to contain server log file |
| file | STRING | server log file base name (rank will be appended) |
| verbosity | INT | server logging verbosity level [0-5] (default: 0) |

Table 4: `[meta]` section - metadata settings

| Key | Type | Description |
|---|---|---|
| db_name | STRING | metadata database file name |
| db_path | STRING | path to directory to contain metadata database |
| range_size | INT | metadata range size (B) (default: 1 MiB) |
| server_ratio | INT | # of UnifyFS servers per metadata server (default: 1) |

Table 5: `[runstate]` section - server runstate settings

| Key | Type | Description |
|---|---|---|
| dir | STRING | path to directory to contain server runstate file |

Table 6: `[server]` section - server settings

| Key | Type | Description |
|---|---|---|
| hostfile | STRING | path to server hostfile |

Table 7: `[sharedfs]` section - server shared files settings

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

Table 8: `[shmem]` section - shared memory segment usage settings

| Key | Type | Description |
|---|---|---|
| chunk_bits | INT | data chunk size (bits), size = 2^bits (default: 24) |
| chunk_mem | INT | segment size (B) for data chunks (default: 256 MiB) |
| recv_size | INT | segment size (B) for receiving data from local server |
| req_size | INT | segment size (B) for sending requests to local server |
| single | BOOL | use one memory region for all clients (default: off) |

Table 9: `[spillover]` section - local data storage spillover settings

| Key | Type | Description |
|---|---|---|
| enabled | BOOL | use local storage for data spillover (default: on) |
| data_dir | STRING | path to spillover data directory |
| meta_dir | STRING | path to spillover metadata directory |
| size | INT | maximum size (B) of spillover data (default: 1 GiB) |

## 6.2 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`.

## 6.3 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.

Table 10: `unifyfsd` command line options

| LongOpt | ShortOpt |
|---|---|
| –unifyfs-cleanup | -C |
| –unifyfs-configfile | -f |
| –unifyfs-consistency | -c |
| –unifyfs-daemonize | -D |
| –unifyfs-mountpoint | -m |
| –log-dir | -L |
| –log-file | -l |
| –log-verbosity | -v |
| –runstate-dir | -R |
| –server-hostfile | -H |
| –sharedfs-dir | -S |

CHAPTER 7

# Starting & Stopping in a Job

In this section, we describe the mechanisms for starting and stopping UnifyFS in a user's job allocation.

Overall, the steps taken 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`

## 7.1 Starting UnifyFS

First, we need to start the UnifyFS server daemon (`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 will report 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 will 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 checkpoint data to node-local ssd storage
4  export UNIFYFS_SPILLOVER_DATA_DIR=/mnt/ssd/$USER/data
5  export UNIFYFS_SPILLOVER_META_DIR=/mnt/ssd/$USER/meta
6
7  # store server logs in job-specific scratch area
```

(continues on next page)

(continued from previous page)

```
    export UNIFYFS_LOG_DIR=$JOBSCRATCH/logs

    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, --cleanup             [OPTIONAL] clean up the UnifyFS storage upon server␣
↪exit
      -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
      -S, --share-dir=<path>    [REQUIRED] shared file system <path> for use by␣
↪servers
      -i, --stage-in=<path>     [OPTIONAL] stage in file(s) at <path>
      -o, --stage-out=<path>    [OPTIONAL] stage out file(s) to <path> on termination

    Command options for "terminate":
      -s, --script=<path>       <path> to custom termination script
```

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 with the specified mountpoint prefix will utilize UnifyFS for their I/O. All other applications will operate unchanged.

## 7.2 Stopping UnifyFS

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

CHAPTER 8

# Examples

There are several examples available on ways to use UnifyFS. These examples build into static and GOTCHA versions (pure POSIX versions coming soon) and are also used as a form of *intregraton testing*.

## 8.1 Examples Locations

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

### 8.1.1 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 without Spack

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*.

### 8.1.2 Build Location

#### Built with Spack

The Spack build location of UnifyFS (on a successful install) only exists when `--keep-stage` in 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 without Spack

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.

---

## 8.2 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 *Starting & Stopping in a Job*.

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, --mapio               use mmap instead of read|write
                           (default: off)
 -n, --nblocks=<count>     count of blocks each process will read|write
                           (default: 32)
 -p, --pattern=<pattern>   'n1' (N-to-1 shared file) or 'nn' (N-to-N file per process)
                           (default: 'n1')
 -P, --prdwr               use pread|pwrite instead of read|write
                           (default: off)
 -S, --stdio               use fread|fwrite instead of read|write
                           (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 /myMountPoint -f myTestFile
```

CHAPTER 9

# Ways to Contribute

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

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

## 9.1 Getting Started

### 9.1.1 Get UnifyFS

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

### 9.1.2 Getting Help

To contact the UnifyFS team, send an email to the mailing list.

## 9.2 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.

---

## 9.3 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.

---

## 9.4 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.

---

## 9.5 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.

---

## 9.6 Documentation

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.

CHAPTER 10

# Style Guides

## 10.1 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.

### 10.1.1 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.

### 10.1.2 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.

## 10.2 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
```

CHAPTER 11

# Testing Guide

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

## 11.1 Unit Tests

### 11.1.1 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:

```
#!/bin/sh

test_description="My awesome test cases"

. $(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 test_done
```

### C Program Tests

C programs use the libtap library to implement test cases. 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 is an example libtap test:

```c
#include "t/lib/tap.h"
#include <string.h>

int main(int argc, char *argv[])
{
    int result;

    result = (1 == 1);
    ok(result, "1 equals 1: %d", result);

    todo("Prove this someday");
    result = strcmp("P", "NP");
    ok(result == 0, "P equals NP: %d", result);
    end_todo;

    done_testing();

    return 0;
}
```

## 11.1.2 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. Creata 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

### 11.1.3 Running the Tests

To manually run the UnifyFS test suite, simply run `make check` from your build/t directory. If changes are made to existing files in the test suite, the tests can be run again by simply doing `make clean` followed by `make check`. Individual tests may be run by hand. The test `0001-setup.t` should normally be run first to start the UnifyFS daemon.

**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
```

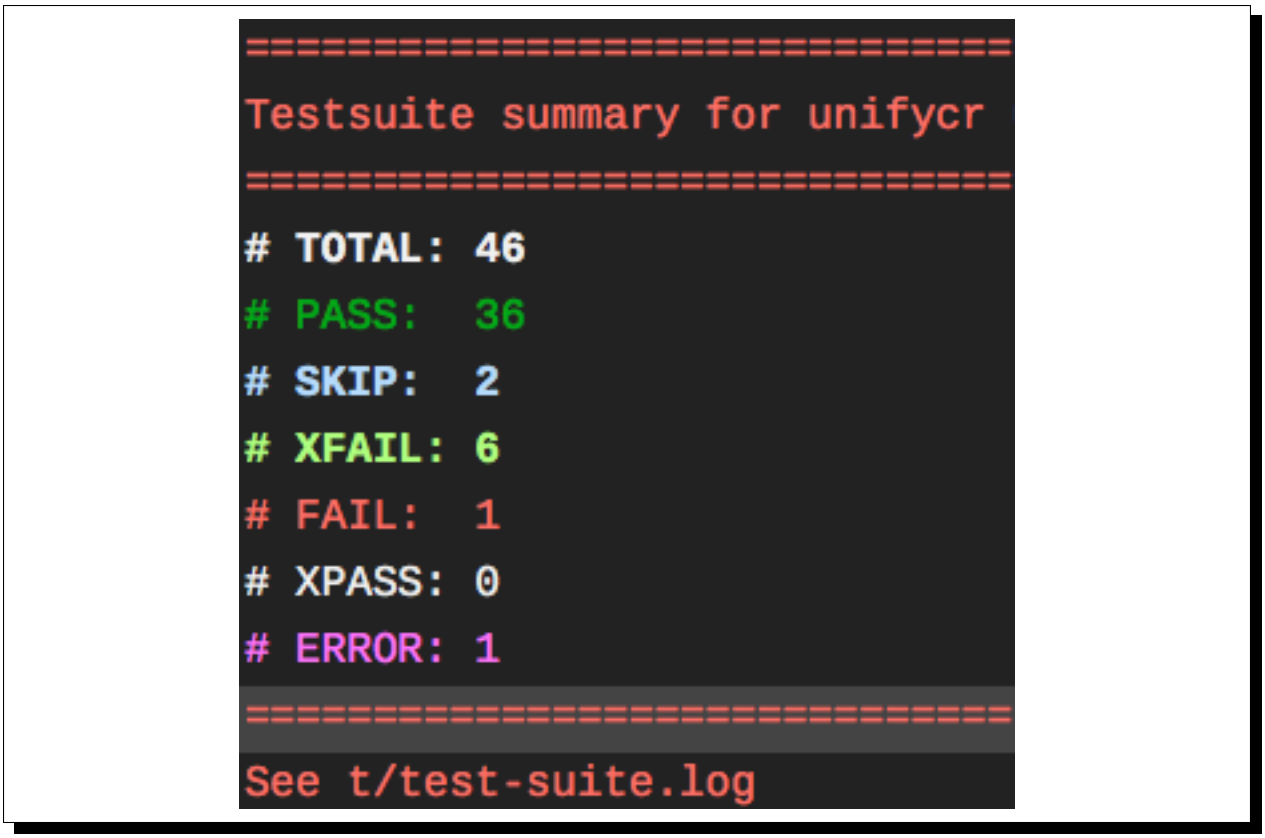
2. Manually from Spack's build directory

```
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**




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 continuation integration tools such as Bamboo or GitLab.

---

## 11.2 Integration Tests

The UnifyFS examples are being used as integration tests with continuation integration tools such as Bamboo or GitLab.

To run any of these examples manually, refer to the *Examples* documentation.

---

### 11.2.1 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 primarly 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.

---

**Important:** 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.

Also make sure all *dependencies* are installed and loaded.

---

By default, the integration tests will use the number of processes-per-node as there are nodes allocated for the job (i.e., if 4 nodes were allocated, then 4 processes will be run per node). This can be changed by setting the *$CI_NPROCS* environment variable.

---

**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, testing functions and variables need to be set up first, and the UnifyFS server needs to be started. To do this, first source the *t/ci/001-setup.sh* script followed by *002-start-server.sh*. Then source each desired test script after that preceded by `$CI_DIR/`. When finished, source the *990-stop-server.sh* script last to stop the server and clean up.

```
$ . full/path/to/001-setup.sh
$ . $CI_DIR/002-start-server.sh
$ . $CI_DIR/100-writeread-tests.sh
$ . $CI_DIR/990-stop-server.sh
```

### Configuration Variables

Along with the already provided *UnifyFS Configuration* options/environment variables, there are available environment variables used by the integration testing suite that can be set in order to change the default behavior. They are listed below in the order they are set up.

#### CI_PROJDIR

USAGE: `CI_PROJDIR=/base/location/to/search/for/UnifyFS/source/files`

During setup, the integration tests will search for the `unifyfsd` executable and installed example scripts if the UnifyFS install directory is not provided by the user with the `UNIFYFS_INSTALL` envar. `CI_PROJDIR` is the base location where this search will start and defaults to `CI_PROJDIR=$HOME`.

#### 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 a UnifyFS installation. Set this envar to prevent the integration tests from searching for a UnifyFS install directory automatically.

#### CI_NPROCS

USAGE: `CI_NPROCS=<number-of-process-per-node>`

The number of processes to use per node inside a job allocation. This defaults to the number of processes per node as there are nodes in the allocation (i.e., if 4 nodes were allocated, then 4 processes will be run per node). This should be adjusted if fewer processes are desired on multiple nodes, multiple processes are desired on a single node, or a large number of nodes have been allocated.

#### CI_LOG_CLEANUP

USAGE: `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, the trash directory will not be cleaned up for debugging purposes. Setting `CI_LOG_CLEANUP=no|NO` will move the *<system-name>_<jobid>/* logs directory to `$CI_DIR` (the directory containing the integration tests) to allow them to persist even when all tests pass. This envar defauls to `yes`.

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

#### CI_HOST_CLEANUP

USAGE: `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 `CI_HOST_CLEANUP=no|NO` to skip cleaning up. This envar defaults to `yes`.

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

#### CI_CLEANUP

USAGE: `CI_CLEANUP=yes|YES|no|NO`

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

#### CI_TEMP_DIR

USAGE: `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 envar defaults to `CI_TEMP_DIR=${TMPDIR}/unifyfs.${USER}.${JOB_ID}`.

#### CI_STORAGE_DIR

USAGE: `CI_STORAGE_DIR=/path/for/storage/files/`

Can be used as a shortcut to set `UNIFYFS_SPILLOVER_DATA_DIR` and `UNIFYFS_SPILLOVER_META_DIR` to the same path. This envar defaults to `CI_STORAGE_DIR=${TMPDIR}/unifyfs.${USER}.${JOB_ID}`.

#### CI_TEST_POSIX

USAGE: `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 envar defaults to `yes`. However, when `$UNIFYFS_MOUNTPOINT` is set to a real directory, this envar is switched to `no`. The idea behind this is that the tests can be run a first time with a fake mountpoint (which will also run the posix tests), and then the tests can be run again with a real mountpoint and the posix tests wont be run twice. This behavior can be overridden by setting `CI_TEST_POSIX=yes|YES` before running the integration tests when `$UNIFYFS_MOUNTPOINT` is set to an existing directory.

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

---

**Note:** The the posix mountpoint envar, `CI_POSIX_MP`, is set up inside `$SHARNESS_TRASH_DIRECTORY` automatically and cleaned up afterwards. However, this envar 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.

---

### 11.2.2 Adding New Tests

In order to add additional tests, 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.

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 arguments as well as any default arguments wanted by all examples. See *below* for details.

#### Example 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 (-vkf 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|-P|-S|-V]). All other args (see *Running the Examples*) are set automatically, including the filename (which is 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=writeread
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 for an example so that if it shows up in the `$UNIFYFS_MOUNTPOINT` (when using an existing mountpoint), it can be tracked to its originating test for debugging. Error files are created with this filename and a `.err` suffix and placed in the logs directory for debugging.

Also allows testers 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 cp/stat tests that potentially need the filename from a previous test.

Note that the filename created by `unify_run_test()` will have a `.app` suffix.

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
- cp/stat tests where an already existing filename from a prior test is needed

For example:

```
basetest=writeread
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 $CI_POSIX_MP $filename
'
```

### 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

### 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.

# CHAPTER 12

# Wrapper Guide

**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

## 12.1 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>
```

## 12.2 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
```

---

## 12.3 Commands to Build Files

### 12.3.1 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
```

### 12.3.2 gnulibc_list.txt

The gnulibc_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/I_002fO-Overview.html#I_002fO-Overview.

### 12.3.3 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.

### 12.3.4 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:

### 12.3.5 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.

CHAPTER 13

# 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.

## 13.1 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_clientcalls_rpc.h*, while server-server RPC structs are defined in *common/src/unifyfs_servercalls_rpc.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.

## 13.2 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*. The RPC handler input and output parameters structs are defined in *common/src/unifyfs_clientcalls_rpc.h*.

All the RPC handler functions follow the same protoype, 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:

## 13.3 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).

# CHAPTER 14

# UnifyFS Dependencies

- GOTCHA version 0.0.2 (compatibility with latest release in progress)
- leveldb version 1.22
- flatcc version 0.5.3
- Margo version 0.4.3 and its dependencies:
  - Argobots version 1.0rc1
  - Mercury version 1.0.1
    - bmi

**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.

CHAPTER 15

# Indices and tables

- genindex
- modindex
- search