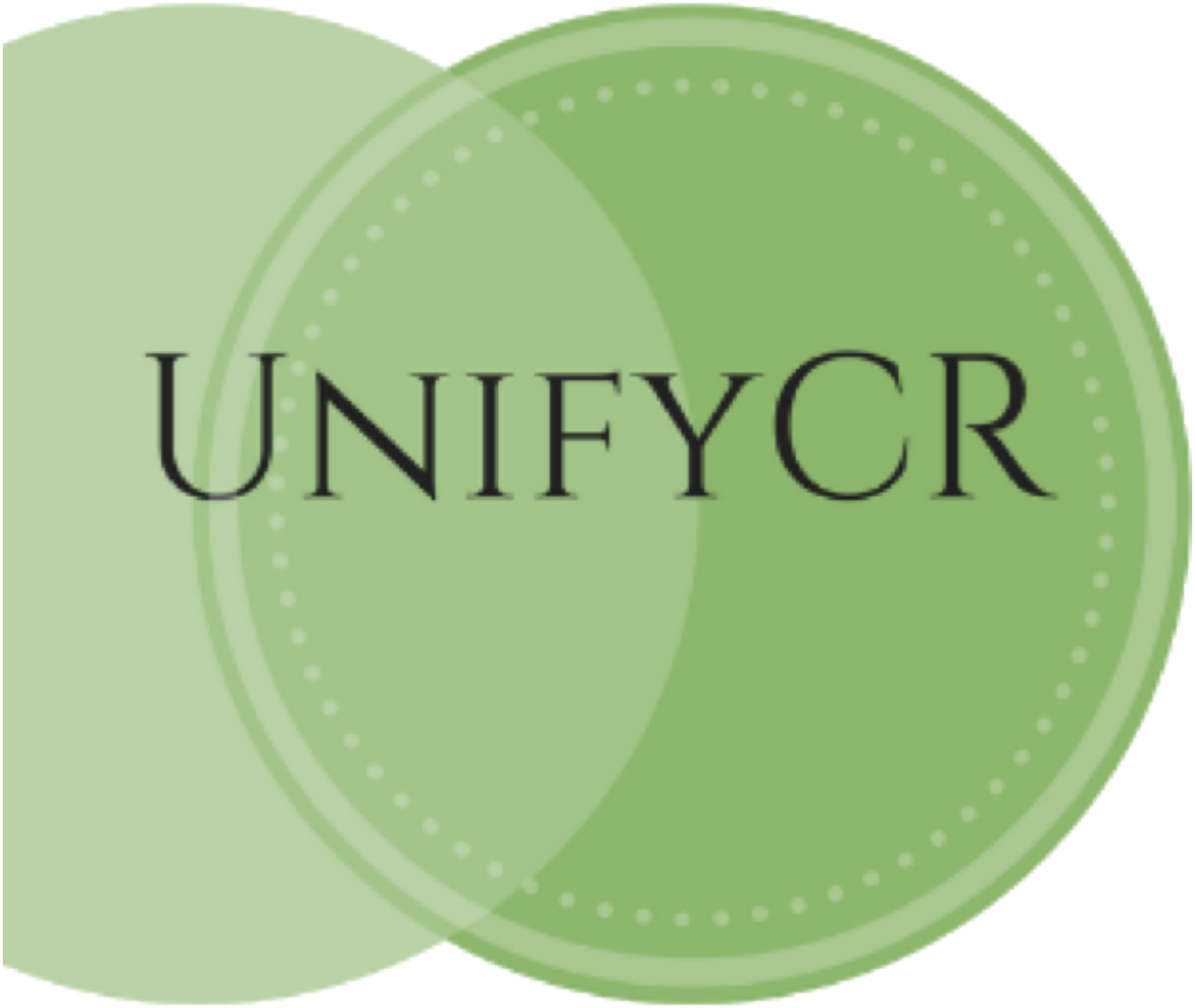


UnifyCR Documentation

Release 0.1

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

May 20, 2019

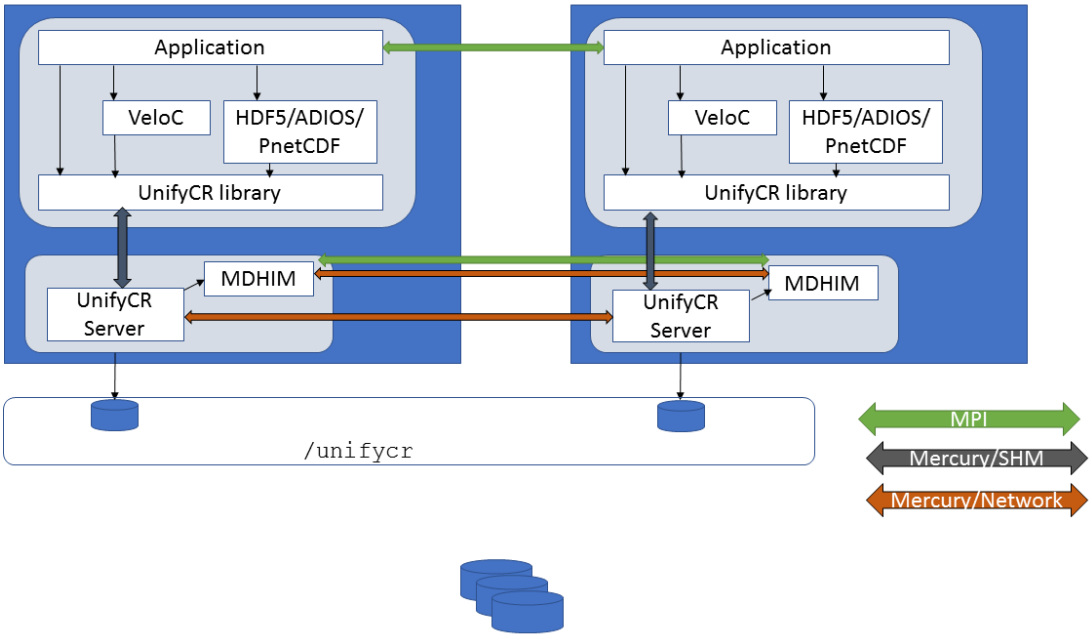
User Guide

CHAPTER 1

Overview

UnifyCR is a user level file system currently under active development. An application can use node-local storage as burst buffers for shared files. UnifyCR 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 UnifyCR, 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 UnifyCR. It will describe the UnifyCR library and the UnifyCR daemon.

1.1 High Level Design



UnifyCR will present a shared namespace (e.g., /unifycr as a mount point) to all compute nodes in a users job allocation. There are two main components of UnifyCR: the UnifyCR library and the UnifyCR daemon. The UnifyCR library (also referred to as the UnifyCR 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 UnifyCR server to be handled. The UnifyCR client library uses the ECP GOTCHA software as its primary mechanism for intercepting I/O calls. Each UnifyCR daemon (also referred to as a UnifyCR server daemon) runs as a daemon on a compute node in the users allocation. The UnifyCR server is responsible for handling the I/O requests from the UnifyCR 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 UnifyCR client library and a high-level I/O library, e.g. HDF5, ADIOS, or PnetCDF. The UnifyCR 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 UnifyCR, and about how UnifyCR 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

In the first version of UnifyCR, lamination will be explicitly initiated by a UnifyCR API call. In subsequent versions, we will support implicit initiation of file lamination. Here, UnifyCR will determine a file to be laminated based on conditions, e.g., texttt{fsync} or texttt{ioctl} calls, or a time out on texttt{close} operations. As part of the UnifyCR project, we will investigate these implicit lamination conditions to determine the best way to enable lamination of files without explicit UnifyCR API calls being made by the application.

In the first version of UnifyCR, eventually, a process declares the file to be laminated through a UnifyCR API call. After a file has been laminated, the contents of the file cannot be changed. The file becomes permanently read-only. After lamination, any process may freely read any part of the file. If the application process group fails before a file has been laminated, UnifyCR may delete the file. An application can delete a laminated file.

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 UnifyCR with I/O interception.

Note: The current version of UnifyCR 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 UnifyCR

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 UnifyCR:

```
$ spack install unifycr
$ spack load unifycr
```

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

Table 1: UnifyCR Build Variants

Variant	Command	Description
HDF5	`spack install unifycr+hdf5` `spack install unifycr+hdf5 ^hdf5~mpi`	Build with parallel HDF5 Build with serial HDF5
Fortran	`spack install unifycr+fortran`	Build with gfortran
NUMA	`spack install unifycr+numa`	Build with NUMA
pmpi	`spack install unifycr+pmpi`	Transparent mount/unmount
PMIx	`spack install unifycr+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 unifycr` before installing to see what Spack is going to do.

4.1.2 Building with Autotools

Download the latest UnifyCR release from the Releases page.

Building the Dependencies

UnifyCR requires MPI, LevelDB, and GOTCHA(version 0.0.2).

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 UnifyCR.

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 unifycr` to install all of UnifyCR's dependencies without installing UnifyCR.

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 build UnifyCR:

```
$ 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, you may fetch version 0.0.2 (compatibility with latest release in progress) of GOTCHA

And leveldb (if not already installed on your system): leveldb

To get flatcc flatcc

To download and install Margo and its dependencies (Mercury and Argobots) follow the instructions here: Margo

Important: Margo uses pkg-config to ensure it compiles and links correctly with all of its dependencies' libraries. When building without Spack, you'll need to manually 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.

Then to build UnifyCR:

```
$ 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 `UNIFYCR_NUMA_POLICY = local | interleaved`, or set NUMA nodes explicitly with `UNIFYCR_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. UnifyCR installs a unifycr-config script that returns those flags, e.g.,

```
$ mpicc -o test_write \
      `<unifycr>/bin/unifycr-config --pre-ld-flags` \
      test_write.c \
      `<unifycr>/bin/unifycr-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<unifycr>/include -L<unifycy>/lib -lunifycr_gotcha \
    -L<gotcha>/lib64 -lgotcha
```

Fortran

```
$ mpif90 -o test_write test_write.F \
    -I<unifycr>/include -L<unifycy>/lib -lunifycrf -lunifycr_gotcha
```

CHAPTER 5

Mounting UnifyCR

In this section, we describe how to use the UnifyCR API in an application.

Attention: Fortran Compatibility

`unifycr_mount` and `unifycr_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 UnifyCR* with Spack, include the `+fortran` variant, or configure UnifyCR with the `--enable-fortran` option if building manually.

5.1 Mounting

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

```
#include <unifycr.h>
```

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

```
include 'unifycrf.h'
```

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

Listing 1: C

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

Listing 2: Fortran

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

Where /tmp is the path prefix you want UnifyCR 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 UnifyCR in your application, you should unmount.

Listing 3: C

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

Listing 4: Fortran

```
call UNIFYCR_UNMOUNT(ierr);
```

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

CHAPTER 6

UnifyCR Configuration

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

- System-wide configuration file: `/etc/unifycr/unifycr.conf`
- Environment variables
- Command line options to `unifycrd`

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 `unifycr.conf`.

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 unifycr.conf

`unifycr.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: `[unifycr]` section - main configuration settings

Key	Type	Description
configfile	STRING	path to custom configuration file
consistency	STRING	consistency model [LAMINATED | POSIX | NONE]
daemonize	BOOL	enable server daemonization (default: off)
debug	BOOL	enable debug output (default: off)
mountpoint	STRING	mountpoint path prefix (default: /unifycr)

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 UnifyCR 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: `[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 7: `[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 `UNIFYCR_SECTION_KEY`, except for the `[unifycr]` section, which uses `UNIFYCR_KEY`. For example, the setting `log.verbosity` has a corresponding environment variable named `UNIFYCR_LOG_VERBOSITY`, while `unifycr.mountpoint` corresponds to `UNIFYCR_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 8: `unifycrd` command line options

LongOpt	ShortOpt
–unifycr-configfile	-C
–unifycr-consistency	-c
–unifycr-daemonize	-D
–unifycr-debug	-d
–unifycr-mountpoint	-m
–log-dir	-L
–log-file	-l
–log-verbosity	-v
–runstate-dir	-R

CHAPTER 7

Starting & Stopping in a Job

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

Overall, the steps taken to run an application with UnifyCR include:

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

7.1 Starting UnifyCR

First, we need to start the UnifyCR server daemon (`unifycrd`) on the nodes in the job allocation. UnifyCR provides the `unifycr` 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 `unifycr 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 `unifycr` utility automatically detects the allocated nodes and launches a server on each node. For example, the following script could be used to launch the `unifycrd` 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/unifycr/unifycr.conf`. See *UnifyCR Configuration* for further details on customizing the UnifyCR runtime configuration.

```
#!/bin/bash

# spillover checkpoint data to node-local ssd storage
export UNIFYCR_SPILLOVER_DATA_DIR=/mnt/ssd/$USER/data
export UNIFYCR_SPILLOVER_META_DIR=/mnt/ssd/$USER/meta

# store server logs in job-specific scratch area
```

(continues on next page)

(continued from previous page)

```
    export UNIFYCR_LOG_DIR=$JOBSCRATCH/logs

    unifycr start --mount=/mnt/unifycr
```

`unifycr` 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 `unifycr` is as follows:

```
    [prompt]$ unifycr --help

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

    <command> should be one of the following:
      start       start the unifycr server daemon
      terminate   terminate the unifycr server daemon

    Available options for "start":
      -C, --consistency=<model> consistency model (NONE | LAMINATED | POSIX)
      -e, --exe=<path>          <path> where unifycrd is installed
      -m, --mount=<path>        mount unifycr at <path>
      -s, --script=<path>       <path> to custom launch script
      -i, --stage-in=<path>     stage in file(s) at <path>
      -o, --stage-out=<path>    stage out file(s) to <path> on termination

    Available options for "terminate":
      -c, --cleanup             clean up the unifycr storage on termination
      -s, --script=<path>       <path> to custom termination script
```

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

7.2 Stopping UnifyCR

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

CHAPTER 8

Examples

There are several examples available on ways to use UnifyCR. 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 UnifyCR is built and where UnifyCR is installed.

8.1.1 Install Location

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

Installed with Spack

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

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

```
$ spack cd -i unifycr
$ cd libexec
```

Installed without Spack

The autotools installation of UnifyCR 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 UnifyCR (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 unifycr`.

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

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

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

Note: If you installed UnifyCR 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 unifycr+hdf5 ^hdf5~mpi
```

Built without Spack

The autotools build of UnifyCR 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 UnifyCR 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]$ ./sysio-write-static --help

Usage: sysio-write-static [options...]

Available options:
 -b, --blocksize=<size in bytes>  logical block size for the target file
                                  (default 1048576, 1MB)
 -n, --nblocks=<count>            count of blocks each process will write
                                  (default 128)
 -c, --chunksize=<size in bytes>  I/O chunk size for each write operation
                                  (default 64436, 64KB)
 -d, --debug                      pause before running test
                                  (handy for attaching in debugger)
 -f, --filename=<filename>        target file name under mountpoint
                                  (default: testfile)
 -h, --help                       help message
 -L, --lipsum                     generate contents to verify correctness
 -m, --mount=<mountpoint>         use <mountpoint> for unifycr
                                  (default: /unifycr)
```

(continues on next page)

(continued from previous page)

```
 -P, --pwrite                      use pwrite(2) instead of write(2)
 -p, --pattern=<pattern>           should be 'n1'(n to 1) or 'nn' (n to n)
                                   (default: n1)
 -S, --synchronous                 sync metadata on each write
 -s, --standard                    do not use unifycr but run standard I/O
 -u, --unmount                     unmount the filesystem after test
```

Notice the mountpoint is defaulted to `-mount=/unifycr`. If you chose a different mountpoint during *Starting & Stopping in a Job*, the `-m` option for the example will need to be provided to match.

One form of running this example could be:

```
$ srun -N4 -n4 sysio-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 UnifyCR even better.

9.1 Getting Started

9.1.1 Get UnifyCR

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

9.1.2 Getting Help

To contact the UnifyCR 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 UnifyCR 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 UnifyCR 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 UnifyCR 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

UnifyCR 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/unifycr.astyle foo.c
```

The unifycr.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

11.1 Implementing Tests

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

11.1.1 Shell Script Tests

Test cases in shell scripts are implemented with sharness, which is included in the UnifyCR source distribution. See the file sharness.sh for all available test interfaces. UnifyCR-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

test_expect_success "Verify some critical invariant" '
    test 1 -eq 1
'

test_expect_failure "Prove this someday" '
```

(continues on next page)

(continued from previous page)

```
    test "P" == "NP"
'

test_done
```

11.1.2 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-unifycr-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:

```
#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;
}
```

11.2 Adding Tests

The UnifyCR 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.

11.2.1 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-unifycr-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 unifycr 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* unifycr_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)

11.2.2 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* unifycr_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

11.3 Running the Tests

To manually run the UnifyCR 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 UnifyCR daemon.

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

1. Upon your installation with Spack

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

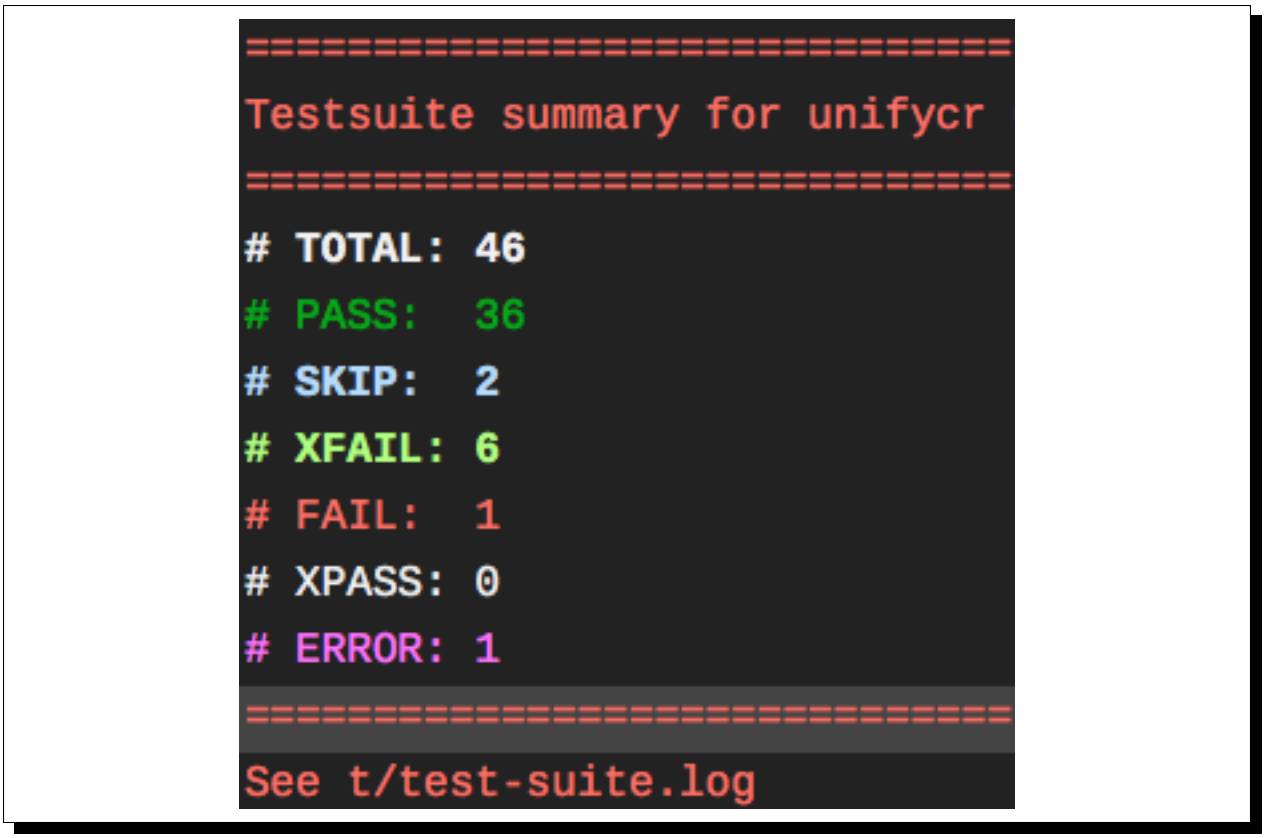
2. Manually from Spack's build directory

```
spack install --keep-stage unifycr
spack cd unifycr
cd spack-build/t
make check
```

The tests in https://github.com/LLNL/UnifyCR/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.

11.3.1 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.

11.3.2 Running the Examples

The UnifyCR examples are also 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.

CHAPTER 12

Wrapper Guide

Warning: This document is out-of-date as the process for generating *unifycr_list.txt* has bugs which causes the generation of *gotcha_map_unifycr_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 UnifyCR.

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 UnifyCR. 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
- unifycr_list.txt - list of wrappers in UnifyCR
- unifycr_unsupported_list.txt - list of wrappers in UnifyCR that are implemented, but not supported

12.1 unifycr_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 UnifyCR. If an application uses an I/O routine that is not supported, it likely cannot use UnifyCR. If the tool does not report unsupported wrappers, the app may work with UnifyCR but it is not guaranteed to work.

```
unifycr_check_fns <executable>
```

12.2 Building the GOTCHA List

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

```
python unifycr_translate.py unifycr_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 unifycr_list.txt

The unifycr_list.txt file specifies the set of wrappers in UnifyCR. Most but not all such wrappers are supported. The command to build unifycr list:

12.3.5 unifycr_unsupported_list.txt

The unifycr_unsupported_list.txt file specifies wrappers that are in UnifyCR, 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 *unifycr_mount_rpc* as an example RPC function to follow throughout.

13.1 Server

1. Write rpc handler function for the server. This is the function that will be invoked on the client, but executed on the server. Most RPC functions executed on the server are implemented in *server/src/unifycr_cmd_handler.c*, such as *unifycr_mount_rpc(hg_handle_t handle)*. After writing the handler function, use a Margo macro to define the RPC handler function below your implementation:

    DEFINE_MARGO_RPC_HANDLER(unifycr_mount_rpc

That goes at the bottom of your RPC handler. You can look at currently implemented RPC functions in *server/src/unifycr_cmd_handler.c* for reference on what this should look like. Generally, the implementations of these fucntions pass in a handle, then use a *margo_get_input* call to retrieve the input struct parameters passed in by the client. Then the RPC handler function operates on those input struct parameters in some way. After finishing, it replies with a *margo_respond*, where the RPC handle and output struct are passed back to the client.

2. Register your implementation of the RPC handler on the server with margo in *server/src/unifycr_init.c*. .. code-block:: C

    MARGO_REGISTER(unifycr_server_rpc_context->mid, "unifycr_mount_rpc", unifycr_mount_in_t, unifycr_mount_out_t, unifycr_mount_rpc);

The last two parameters to *MARGO_REGISTER* are input and output structs that are defined on the client, but also need to be registered with the server. The input struct is passed in by the client to the RPC handler function, and then forwarded to the server. When the RPC handler function finishes, the output struct will be passed back to the client, which is generally a return code.

13.2 Client

1. Define the input and output structs for your RPC handler in *common/src/unifycr_clientcalls_rpc.h*. This uses the MERCURY_GEN_PROC macro: .. code-block:: C

MERCURY_GEN_PROC(unifycr_mount_out_t, ((hg_size_t)(max_recs_per_slice)) ((int32_t)(ret)))

MERCURY_GEN_PROC(unifycr_mount_in_t, ((int32_t)(app_id)) ((int32_t)(local_rank_idx)) ((int32_t)(dbg_rank)) ((int32_t)(num_procs_per_node)) ((hg_const_string_t)(client_addr_str)) ((hg_size_t)(req_buf_sz)) ((hg_size_t)(recv_buf_sz)) ((hg_size_t)(superblock_sz)) ((hg_size_t)(meta_offset)) ((hg_size_t)(meta_size)) ((hg_size_t)(fmeta_offset)) ((hg_size_t)(fmeta_size)) ((hg_size_t)(data_offset)) ((hg_size_t)(data_size)) ((hg_const_string_t)(external_spill_dir)))

The input struct will be all of the parameters the client sends to the RPC handler function on the server, so they need to be defined in the client invocation function.

Note: Passing some types can be an issue. You can look through the mercury documentation on what types are supported here: **'<https://mercury-hpc.github.io/documentation/'_** (look under Predefined Types). If your type is not a predefined type you will most likely have to write the code to serialize/deserialize the input/output structs. Phil said he has starter code for this, since much of the code is similar.

---

2. Register the RPC handler with the client in cilent/src/unifycr.c. The pointer to the margo id for the RPC handler will be stored in an rpc_context that will be used when the client invokes the RPC handler. .. code-block:: C

(*unifycr_rpc_context)->unifycr_mount_rpc_id = MARGO_REGISTER((*unifycr_rpc_context)->mid, "unifycr_mount_ unifycr_mount_in_t, unifycr_mount_out_t, NULL);

When the client calls *MARGO_REGISTER* the last parameter is *NULL*, this is the RPC handler function that is not defined on the client. When the server calls this function the last parameter would be the name of the RPC handler function instead of *NULL*.

3. Add mercury id for the name of the RPC handler to the ClientRpcContext in *common/src/unifycr_client.h*. .. code-block:: C

typedef struct ClientRpcContext { margo_instance_id mid; hg_context_t* hg_context; hg_class_t* hg_class; hg_addr_t svr_addr; hg_id_t unifycr_mount_rpc_id;

4. Define an invocation prototype for the client invocation function: .. code-block:: C

int32_t unifycr_client_mount_rpc_invoke(unifycr_client_rpc_context_t** unifycr_rpc_context);

5. Invoke the RPC function on the Client: .. code-block:: C

unifycr_client_mount_rpc_invoke(&unifycr_rpc_context);

The implementation for these invocation functions have generally been implemented in *client/src/unifycr_client.c* or the common directory. The connection to the server is established with a margo_create call (where the address of the server is passed), then the RPC is actually forwarded to the server with a *margo_forward* call, where the RPC handle and input struct are passed. You can get the RPC output with a margo_get_output call that passes back a return code. There are many more example invocation functions in *client/src/unifycr_client.c*.

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

Indices and tables

- genindex
- modindex
- search