Mochi Documentation

Argonne National Laboratory

Feb 25, 2021

Contents

The Mochi project is a collaboration between Argonne National Laboratory, Los Alamos National Laboratory, Carnegie Mellon University, and the HDF Group. The objective of this project is to explore a software defined storage approach for composing storage services that provides new levels of functionality, performance, and reliability for science applications at extreme scale.

This website gathers documentation and tutorials for the main libraries used or developed in the context of the Mochi project.

CHAPTER 1

Getting started

There are multiple ways of getting started with Mochi depending on your goal. In all cases though, we recommend reading about Margo first. Margo is what binds Mercury (RPC/RDMA) and Argobots (threading/tasking) under a runtime that all the other components will use. If you are into C++, you may then want to read about Thallium, which is a C++ library written on top of Margo.

If you want to use some of our components, you can then jump to their corresponding tutorial sections. If you want to _develop_ your own component, your can jump to the Mochi templates and to the Bedrock bootstrapping system.

CHAPTER 2

The core Mochi libraries

Margo is a C library enabling the development of distributed HPC services. It relies on Mercury for RPC/RDMA, and Argobots for threading/tasking, hidding the complexity of these two libraries under a simple programming model.

Thallium is a C++14 library wrapping Margo and enabling the development of the same sort of services using all the power of modern C++. It is the recommended library for C++ developers. Note that Thallium also provides C++ wrappers to Argobots.

Mercury is Mochi's underlying RPC/RDMA library. While it is not necessary to understand how to use Mercury itself when developing with Margo or Thallium (which we recommend), we provide a set of tutorials for those who would need to use it directly rather than through higher level libraries.

Argobots is used for threading/tasking in Mochi. Understanding its underlying programming model may not be necessary at first, for simple Margo or Thallium services, but may become useful to optimize performance or customize the scheduling and placement of threads and tasks in a Mochi service.

Note: The tutorials for each of these libraries are independent of one another. Feel free to start with the most relevant for you.

Important: In all the tutorials, we use the term "server" to denote a process to which one can send RPC requests, and "client" to denote a process that sends such RPC requests. It is important to note however that a server can also send RPC requests to other servers, and even to itself.

CHAPTER 3

Other Mochi libraries/components

ABT-IO is a small library that can be used to offload POSIX I/O operations to dedicated execution stream to better integrate with the core Mochi libraries. ABT-IO depends on Argobots only.

SSG is Mochi's Scalable Service Group library. It provides functionalities to bootstrap a dynamic group of process and manage group membership. This library can be used for fault tolerance and/or to implement elastic services.

CHAPTER 4

Developing and deploying a service

Templates are provided to help users develop their own components using Margo or Thallium. These templates can save a tremendous amount of time and will let you focus on the important part of your service: its RPCs and its API.

Bedrock is a bootstrapping system for composed Mochi services. It provides a simple a unified way of deploying Mochi components in a process and configure them using a JSON file. It also enable querying and changing this configuration at run time.

CHAPTER 5

Contents

5.1 Installing

The recommended way to install the Mochi libraries and dependencies is to use Spack. Spack is a package management tool designed to support multiple versions and configurations of software on a wide variety of platforms and environments.

5.1.1 Installing Spack and the SDS repository

First, you will need to install Spack as explained here. Once Spack is installed and available in your path, clone the following git reporitory and add it as a Spack namespace.

```
git clone https://xgitlab.cels.anl.gov/sds/sds-repo.git
spack repo add sds-repo
```

You can then check that Spack can find Margo (for example) by typping:

```
spack info mochi-margo
```

You should see something like the following.

```
AutotoolsPackage:   mochi-margo

Description:
    A library that provides Argobots bindings to the Mercury RPC
    implementation.

Homepage: https://xgitlab.cels.anl.gov/sds/margo
... (more lines follow) ...
```

5.1.2 Installing the Mochi libraries

Installing Margo is then as simple as typping the following.

```
spack install mochi-margo
```

You will notice that Spack also installs Mercury and Argobots, since these are needed by Margo, as well as other dependencies.

You can install Thallium using `spack install mochi-thallium` (this will install Margo if you didn't install it before, as well as its dependencies).

`spack install mercury` can be used to install Mercury, and `spack install argobots` can be used to install Argobots, should you need to install either independently of Margo or Thallium. `spack install mochi-abt-io` will install ABT-IO. `spack install mochi-ssg` will install SSG.

5.1.3 Loading and using the Mochi libraries

Once installed, you can load Margo using the following command.

```
spack load -r mochi-margo
```

This will load Margo and its dependencies (Mercury, Argobots, etc.). `spack load -r mochi-thallium` will load Thallium and its dependencies (Margo, Mercury, Argobots, etc.). You are now ready to use the Mochi libraries!

5.1.4 Using the Mochi libraries with pkg-config

Once loaded, all the Mochi libraries can be found using `pkg-config`. For examples:

```
$ pkg-config --libs margo
```

5.1.5 Using the Mochi libraries with cmake

Within a cmake project, Thallium and Mercury can be found using:

```
find_package(mercury REQUIRED)
include_directories(${MERCURY_INCLUDE_DIR})
find_package(thallium REQUIRED)
```

To make cmake find Margo, Argobots, or ABT-IO, download this file and place it in a *cmake* folder in your project. In the root CMakeLists.txt file of your project, add `set (CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_CURRENT_SOURCE_DIR}/cmake")` and `include (xpkg-import)`. You can then find Margo, Argobots, and ABT-IO using the following:

```
xpkg_import_module (argobots REQUIRED argobots)
xpkg_import_module (margo REQUIRED margo)
xpkg_import_module (abtio REQUIRED abt-io)
xpkg_import_module (ssg REQUIRED ssg)
```

You can now link targets as follows.

```
# Code using Mercury
add_executable(my_mercury_prog source.c)
target_link_libraries(my_mercury_prog mercury)

# Code using Margo
add_executable(my_margo_prog source.c)
target_link_libraries(my_margo_prog margo)

# Code using Thallium
add_executable(my_thallium_prog source.cpp)
target_link_libraries(my_thallium_prog thallium)

# Code using Argobots
add_executable(my_abt_prog source.c)
target_link_libraries(my_abt_prog abt)

# Code using ABT-IO
add_executable(my_abt_io_prog source.c)
target_link_libraries(my_abt_io_prog abt-io abt)

# Code using SSG
add_executable(my_ssg_prog source.c)
target_link_libraries(my_ssg_prog ssg)
```

5.2 Margo

Margo is a C library that helps develop distributed services based on RPC and RDMA.

Margo provides Argobots-aware wrappers to Mercury functions. It simplifies service development by expressing Mercury operations as conventional blocking functions so that the caller does not need to manage progress loops or callback functions. Internally, Margo suspends callers after issuing a Mercury operation, and automatically resumes them when the operation completes. This allows other concurrent user-level threads to make progress while Mercury operations are in flight without consuming operating system threads. The goal of this design is to combine the performance advantages of Mercury's native event-driven execution model with the progamming simplicity of a multi-threaded execution model.

This section will walk you through a series of tutorials on how to use Margo. We highly recommend to read all the tutorials before diving into the implementation of any Margo-based service, in order to understand how we envision designing such a service. This will also help you greatly in understanding other Margo-based services.

5.2.1 Initializing Margo

In this tutorial, we will see how to initialize Margo.

Initializing a server

The following code initializes Margo for use as a server, then prints the address at which the server can be contacted.

```
#include <assert.h>
#include <stdio.h>
#include <margo.h>

```

(continues on next page)

(continued from previous page)

```
int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, -1);
    assert(mid);

    hg_addr_t my_address;
    margo_addr_self(mid, &my_address);
    char addr_str[128];
    size_t addr_str_size = 128;
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);

    margo_set_log_level(mid, MARGO_LOG_INFO);
    margo_info(mid, "Server running at address %s", addr_str);

    margo_wait_for_finalize(mid);

    return 0;
}
```

`margo_init` creates a `margo_instance_id` object. It takes four arguments. The first one is *protocol* (here TCP). It is also possible to provide the address and port number to use. The second argument specifies that Margo is initialized as a server. The third argument indicates whether an Argobots execution stream (ES) should be created to run the Mercury progress loop. If this argument is set to 0, the progress loop is going to run in the context of the main ES (this should be the standard scenario, unless you have a good reason for not using the main ES, such as the main ES using MPI primitives that could block the progress loop). A value of 1 will make Margo create an ES to run the Mercury progress loop. The fourth argument is the number of ES to create and use for executing RPC handlers. A value of 0 will make Margo execute RPCs in the ES that called `margo_init`. A value of -1 will make Margo execute the RPCs in the ES running the progress loop. A positive value will make Margo create new ESs to run the RPCs.

`margo_addr_self` is then used to get the address of the server, which is then converted into a string using `margo_addr_to_string`. The address returned by `margo_addr_self` should be freed using `margo_addr_free`.

`margo_wait_for_finalize` blocks the server in the Mercury progress loop until another ES calls `margo_finalize`. In this example, nothing calls `margo_finalize`, so you will need to kill the server manually if you run it.

Note: We use `margo_info` to display information. This function is part of Margo's logging feature, which includes six logging levels: `margo_trace`, `margo_debug`, `margo_info`, `margo_warning`, `margo_error`, and `margo_critical`. These functions take a margo instance as first argument, a string format as second argument, and optional parameters. Note that these functions will automatically add a `\n` at the end of the provided string. The logging level can be set using `margo_set_log_level` (see margo-logging.h).

Initializing a client

The following code initializes Margo for use as a client.

client.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
```

(continues on next page)

(continued from previous page)

```c
#include <margo.h>

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp",MARGO_CLIENT_MODE, 0, 0);
    assert(mid);

    margo_finalize(mid);

    return 0;
}
```

We call `margo_init` with `MARGO_CLIENT_MODE` to indicate that this is a client. Just like servers, clients have to run a Mercury progress loop. This progress loop can be executed in the context of the main ES (by providing 0 as a third argument) or in a separate ES (by providing 1). In general, a value of 0 is sufficient. Putting the Mercury progress loop of a client in a separate ES is useful if the client uses non-blocking RPCs, or if the client is multithreaded.

`margo_finalize` is used to finalize the `margo_instance_id` object.

Extended initialization

A `margo_init_ext` function is provided that enables passing configuration parameters as well as externally-initialized Argobots pools and Mercury class/context. This function is explained in detail in a latter section.

5.2.2 Simple Hello World RPC

The previous tutorial explained how to initialize a server and a client. This this tutoria, we will have the server register and RPC handler and the client send an RPC request to the server.

Server-side RPC handler

We will change the code of our server as follows.

server.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <margo.h>

static const int TOTAL_RPCS = 4;
static int num_rpcs = 0;

static void hello_world(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(hello_world)

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, -1);
    assert(mid);

    hg_addr_t my_address;
    margo_addr_self(mid, &my_address);
    char addr_str[128];
```

(continues on next page)

(continued from previous page)

```
    size_t addr_str_size = 128;
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);

    margo_set_log_level(mid, MARGO_LOG_INFO);
    margo_info(mid, "Server running at address %s", addr_str);

    hg_id_t rpc_id = MARGO_REGISTER(mid, "hello", void, void, hello_world);
    margo_registered_disable_response(mid, rpc_id, HG_TRUE);

    margo_wait_for_finalize(mid);

    return 0;
}

static void hello_world(hg_handle_t h)
{
    hg_return_t ret;

    margo_instance_id mid = margo_hg_handle_get_instance(h);

    margo_info(mid, "Hello World!");
    num_rpcs += 1;

    ret = margo_destroy(h);
    assert(ret == HG_SUCCESS);

    if(num_rpcs == TOTAL_RPCS) {
        margo_finalize(mid);
    }
}
DEFINE_MARGO_RPC_HANDLER(hello_world)
```

What changes is the following declaration of an RPC handler.

```
static void hello_world(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(hello_world)
```

The first line declares the function that will be called upon receiving a "hello" RPC request. This function must take a `hg_handle_t` object as argument and does not return anything.

The second line declares *hello_world* as a RPC handler. `DECLARE_MARGO_RPC_HANDLER` is a macro that generates the code necessary for the RPC handler to be placed in an Argobots user-level thread (ULT).

The two lines that register the RPC handler in the Margo instance are the following.

```
hg_id_t rpc_id = MARGO_REGISTER(mid, "hello", void, void, hello_world);
margo_registered_disable_response(mid, rpc_id, HG_TRUE);
```

`MARGO_REGISTER` is a macro that registers the RPC handler. Its first argument is the Margo instance. The second is the name of the RPC. The third and fourth are the types of the RPC's input and output, respectively. We will cover these in the next tutorial. For now, our hello_world RPC is not going to receive any argument and not return any value. The last parameter is the function we want to use as RPC handler.

The `margo_registered_disable_response` is used to indicate that this RPC handler does not send a response back to the client.

The rest of the program defines the `hello_world` function. From inside an RPC handler, we can access the Margo instance using `margo_hg_handle_get_instance`. This is the prefered method for better code organization, rather than declaring the Margo instance as a global variable.

The RPC handler must call `margo_destroy` on the `hg_handle_t` argument it is being passed, after we are done using it.

In this example, after receiving 4 requests, the RPC handler will call `margo_finalize`, which will make the main ES exit the call to `margo_wait_for_finalize` and terminate.

After the definition of the RPC handler, `DEFINE_MARGO_RPC_HANDLER` must be called for Margo to define additional wrapper functions.

Note: We will see at the end of this tutorial how to avoid using global variables (here `TOTAL_RPCS` and `num_rpcs`).

Calling the RPC from clients

The following code is the corresponding client.

client.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <margo.h>

int main(int argc, char** argv)
{
    if(argc != 2) {
        fprintf(stderr,"Usage: %s <server address>\n", argv[0]);
        exit(0);
    }

    hg_return_t ret;
    margo_instance_id mid = MARGO_INSTANCE_NULL;


    mid = margo_init("tcp",MARGO_CLIENT_MODE, 0, 0);
    assert(mid);

    hg_id_t hello_rpc_id = MARGO_REGISTER(mid, "hello", void, void, NULL);

    margo_registered_disable_response(mid, hello_rpc_id, HG_TRUE);

    hg_addr_t svr_addr;
    ret = margo_addr_lookup(mid, argv[1], &svr_addr);
    assert(ret == HG_SUCCESS);

    hg_handle_t handle;
    ret = margo_create(mid, svr_addr, hello_rpc_id, &handle);
    assert(ret == HG_SUCCESS);

    ret = margo_forward(handle, NULL);
    assert(ret == HG_SUCCESS);

    ret = margo_destroy(handle);
    assert(ret == HG_SUCCESS);
```

(continues on next page)

(continued from previous page)

```
    ret = margo_addr_free(mid, svr_addr);
    assert(ret == HG_SUCCESS);

    margo_finalize(mid);

    return 0;
}
```

This client takes the server's address as argument (copy-past the address printed by the server when calling the client). This string representation of the server's address must be resolved into a `hg_addr_t` object. This is done by `margo_addr_lookup`.

Once resolved, the address can be used in a call to `margo_create` to create a `hg_handle_t` object. The `hg_handle_t` object represents an RPC request ready to be sent to the server.

`margo_forward` effectively sends the request to the server. We pass `NULL` as a second argument because the RPC does not take any input.

Because we have called `margo_registered_disable_response`, Margo knows that the client should not expect a response from the server, hence `margo_forward` will return immediately. We then destroy the handle using `margo_destroy`, free the `hg_addr_t` object using `margo_addr_free`, and finalize Margo.

Note: `MARGO_REGISTER` in clients is being passed `NULL` as last argument, since the actual RPC handler function is located in the server.

Attaching data to RPC handlers

Back to the server, we have used two global variables: `TOTAL_RPCS` and `num_rpcs`. Any good developer knows that global variables are evil and every use of a global variable kills a kitten somewhere. Fortunately, we can modify our program to get rid of global variables.

First we will declare a structure to encapsulate the server's data.

```
typedef struct {
    int max_rpcs;
    int num_rpcs;
} server_data;
```

We can now initialize our server data as a local variable inside main, and attach it to our *hello* RPC handler, as follows.

```
server_data svr_data = {
        .max_rpcs = 4,
        .num_rpcs = 0
};
...
hg_id_t rpc_id = MARGO_REGISTER(mid, "hello", void, void, hello_world);
margo_registered_disable_response(mid, rpc_id, HG_TRUE);
margo_register_data(mid, rpc_id, &svr_data, NULL);
```

`margo_register_data` is the function to use to attach data to an RPC handler. It takes a Margo instance, the id of the registered RPC, a pointer to the data to register, and a pointer to a function to call to free that pointer. Since here our data is on the stack, we pass `NULL` as the last parameter.

Important: You need to make sure that the data attached to an RPC handler will not disappear before Margo is finalized. A common mistake consists of attaching a pointer to a piece of data that is on the stack within a function that then returns. In our example above, because `main` will block on `margo_wait_for_finalize`, we know `main` will return only after `margo_finalize` has been called.

In the `hello_world` RPC handler, we can now retrieve the attached data as follows.

```
const struct hg_info* info = margo_get_info(h);
server_data* svr_data = (server_data*)margo_registered_data(mid, info->id);
```

We can now replace the use of global variables by accessing the variables inside `svr_data` instead.

Important: If you have initialized Margo with multiple ES to server RPCs (last argument of `margo_init` strictly greater than 1), you will need to protect such attached data with a mutex or a read-write lock. For more information on such locking mechanisms, please refer to the Argobots tutorials.

5.2.3 Sending arguments, returning values

In the previous tutorial we saw how to send a simple RPC. This RPC was not taking any argument and the server was not sending any response to the client. In this tutorial we will see how to add arguments and return values to an RPC. We will build an RPC handler that receives two integers and returns their sum to the client.

Input and output structures

First, we need to define structures encapsulating the input and output data. This is done using Mercury macros as shown in the following code, which we will place in a *types.h* private header file.

types.h (show/hide)

```
#ifndef PARAM_H
#define PARAM_H

#include <mercury.h>
#include <mercury_macros.h>

/* We use the Mercury macros to define the input
 * and output structures along with the serialization
 * functions.
 */
MERCURY_GEN_PROC(sum_in_t,
        ((int32_t)(x))\
        ((int32_t)(y)))

MERCURY_GEN_PROC(sum_out_t, ((int32_t)(ret)))

#endif
```

We include `<mercury.h>` and `<mercury_macros.h>`. The latter contains the necessary macros. We then use `MERCURY_GEN_PROC` to declare our structures. This macro expends not only to the definition of the `sum_in_t` and `sum_out_t` structures, but also to that of serialization functions that Mercury (hence Margo) can use to serialize these structures into a buffer, and deserialize them from a buffer.

Note: Once these types are defined using the macro, you can use them as members of other types.

Note: The `<mercury_proc_string.h>` may also be included. It provides the `hg_string_t` and `hg_const_string_t` types, which are typedefs of `char*` and `const char*` respectively and must be used to represent null-terminated strings.

Important: The structures defined with the `MERCURY_GEN_PROC` cannot contain pointers (apart from the `hg_string_t` and `hg_const_string_t` types). We will see in a future tutorial how to define serialization functions for more complex structures.

Sum server code

The following shows the server code.

server.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <margo.h>
#include "types.h"

typedef struct {
    int max_rpcs;
    int num_rpcs;
} server_data;

static void sum(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(sum)

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, 0);
    assert(mid);

    server_data svr_data = {
        .max_rpcs = 4,
        .num_rpcs = 0
    };

    hg_addr_t my_address;
    margo_addr_self(mid, &my_address);
    char addr_str[128];
    size_t addr_str_size = 128;
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);

    margo_info(mid, "Server running at address %s", addr_str);

    hg_id_t rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, sum);
    margo_register_data(mid, rpc_id, &svr_data, NULL);

    margo_wait_for_finalize(mid);

    return 0;
}
```

(continues on next page)

(continued from previous page)

```
static void sum(hg_handle_t h)
{
    hg_return_t ret;

    sum_in_t in;
    sum_out_t out;

    margo_instance_id mid = margo_hg_handle_get_instance(h);
    margo_set_log_level(mid, MARGO_LOG_INFO);

    const struct hg_info* info = margo_get_info(h);
    server_data* svr_data = (server_data*)margo_registered_data(mid, info->id);

    ret = margo_get_input(h, &in);
    assert(ret == HG_SUCCESS);

    out.ret = in.x + in.y;
    margo_trace(mid, "Computed %d + %d = %d", in.x, in.y, out.ret);

    ret = margo_respond(h, &out);
    assert(ret == HG_SUCCESS);

    ret = margo_free_input(h, &in);
    assert(ret == HG_SUCCESS);

    ret = margo_destroy(h);
    assert(ret == HG_SUCCESS);

    svr_data->num_rpcs += 1;
    if(svr_data->num_rpcs == svr_data->max_rpcs) {
        margo_finalize(mid);
    }
}
DEFINE_MARGO_RPC_HANDLER(sum)
```

This code is very similar to our earlier code (you will notice that we have attached data to the RPC to avoid using global variables, as advised at the end of the previous tutorial).

Now `MARGO_REGISTER` takes the types of the arguments being sent and received, rather than `void`. Notice that we are also not calling `margo_registered_disable_response` anymore since this time the server will send a response to the client.

Two structures `in` and `out` are declared at the beginning of the RPC handler. `in` will be used to deserialize the arguments sent by the client, while `out` will be used to send a response.

`margo_get_input` is used to deserialize the content of the RPC's data into the variable `in`. The `out` variable is then modified with `out.ret = in.x + in.y;` before being sent back to the client using `margo_respond`.

Important: An input deserialized using `margo_get_input` should be freed using `margo_free_input`, even if the structure is on the stack and does not contain pointers.

Client code

Let's now take a look at the client's code.

client.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <margo.h>
#include "types.h"

int main(int argc, char** argv)
{
    if(argc != 2) {
        fprintf(stderr,"Usage: %s <server address>\n", argv[0]);
        exit(0);
    }

    margo_instance_id mid = margo_init("tcp", MARGO_CLIENT_MODE, 0, 0);
    margo_set_log_level(mid, MARGO_LOG_INFO);

    hg_id_t sum_rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, NULL);

    hg_addr_t svr_addr;
    margo_addr_lookup(mid, argv[1], &svr_addr);

    int i;
    sum_in_t args;
    for(i=0; i<4; i++) {
        args.x = 42+i*2;
        args.y = 42+i*2+1;

        hg_handle_t h;
        margo_create(mid, svr_addr, sum_rpc_id, &h);
        margo_forward(h, &args);

        sum_out_t resp;
        margo_get_output(h, &resp);

        margo_info(mid, "Got response: %d+%d = %d\n", args.x, args.y, resp.ret);

        margo_free_output(h,&resp);
        margo_destroy(h);
    }

    margo_addr_free(mid, svr_addr);

    margo_finalize(mid);

    return 0;
}
```

Again, `MARGO_REGISTER` takes the types of the arguments being sent and received. We initialize the `sum_in_t args;` and `sum_out_t resp;` to hold respectively the arguments of the RPC (what will become the `in` variable on the server side) and the return value (`out` on the server side).

`margo_forward` now takes a pointer to the input argument as second parameter, and `margo_get_output` is used to deserialized the value returned by the server into the `resp` variable.

Important: Just like we called `margo_free_input` on the server because the input had been obtained using `margo_get_input`, we must call `margo_free_output` on the client side because the output has been obtained using `margo_get_output`.

Timeout

It can sometimes be important for the client to be able to timeout if an operation takes too long. This can be done using `margo_forward_timed`, which takes an extra parameter: a timeout (`double`) value in milliseconds. If the server has not responded to the RPC after this timeout expires, `margo_forward_timed` will return `HG_TIMEOUT` and the RPC will be cancelled.

Important: The fact that a call has timed out does not mean that the server hasn't received the RPC or hasn't processed it. It simply means that, should the server send a reponse back, this response will be ignored by the client. Worse: the server will not be aware that the client has cancelled the operation. It is up to the developer to make sure that such a behavior is consistent with the semantics of her protocol.

5.2.4 Transferring data over RDMA

Margo inherits from Mercury the possibility to do RDMA operations. In this tutorial, we will revisit our *sum* example and have the client send a bunch of values to the server by exposing a buffer of memory where these values are located, and have the server pull from this memory.

Input/Output with hg_bulk_t

Let's first take a look at the types.

types.h (show/hide)

```
#ifndef PARAM_H
#define PARAM_H

#include <mercury.h>
#include <mercury_macros.h>

MERCURY_GEN_PROC(sum_in_t,
        ((int32_t)(n))\
        ((hg_bulk_t)(bulk)))

MERCURY_GEN_PROC(sum_out_t, ((int32_t)(ret)))

#endif
```

The `hg_bulk_t` opaque type represents a handle to a region of memory in a process. In addition to this handle, we add a field `n` that will tell us how many values are in the buffer.

Client exposing memory

Starting with the client code for once.

client. (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <margo.h>
#include "types.h"

int main(int argc, char** argv)
{
    if(argc != 2) {
        fprintf(stderr,"Usage: %s <server address>\n", argv[0]);
        exit(0);
    }

    margo_instance_id mid = margo_init("tcp", MARGO_CLIENT_MODE, 0, 0);
    margo_set_log_level(mid, MARGO_LOG_DEBUG);

    hg_id_t sum_rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, NULL);

    hg_addr_t svr_addr;
    margo_addr_lookup(mid, argv[1], &svr_addr);

    int i;
    sum_in_t args;
    for(i=0; i<4; i++) {

        int32_t values[10] = { 1,4,2,5,6,3,5,3,2,5 };
        hg_size_t segment_sizes[1] = { 10*sizeof(int32_t) };
        void* segment_ptrs[1] = { (void*)values };

        hg_bulk_t local_bulk;
        margo_bulk_create(mid, 1, segment_ptrs, segment_sizes, HG_BULK_READ_ONLY, &
↪local_bulk);

        args.n = 10;
        args.bulk = local_bulk;

        hg_handle_t h;
        margo_create(mid, svr_addr, sum_rpc_id, &h);
        margo_forward(h, &args);

        sum_out_t resp;
        margo_get_output(h, &resp);

        margo_debug(mid, "Got response: %d", resp.ret);

        margo_free_output(h,&resp);
        margo_destroy(h);

        margo_bulk_free(local_bulk);
    }

    margo_addr_free(mid, svr_addr);

    margo_finalize(mid);

    return 0;
}
```

We allocate the `values` buffer as an array of 10 integers (this array is on the stack in this example. An array allocated on the heap would work just the same). `margo_bulk_create` is used to create an `hg_bulk_t` handle representing the segment of memory exposed by the client. Its first parameter is the `margo_instance_id`. Then come the number of segments to expose, a `void**` array of addresses pointing to each segment, a `hg_size_t*` array of sizes for each segment, and the mode used to expose the memory region. `HG_BULK_READ_ONLY` indicates that Margo will only read (i.e., the server will only pull) from this segment. `HG_BULK_WRITE_ONLY` indicates that Margo will only write to the segment and `HG_BULK_READWRITE` indicates that both operations may happen.

The bulk handle is freed after being used, using `margo_bulk_free`.

Server pulling from client

Let's now take a look at the server.

server.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <margo.h>
#include "types.h"

static const int TOTAL_RPCS = 16;
static int num_rpcs = 0;

static void sum(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(sum)

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, 0);
    assert(mid);
    margo_set_log_level(mid, MARGO_LOG_INFO);

    hg_addr_t my_address;
    margo_addr_self(mid, &my_address);
    char addr_str[128];
    size_t addr_str_size = 128;
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);

    margo_info(mid, "Server running at address %s\n", addr_str);

    MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, sum);

    margo_wait_for_finalize(mid);

    return 0;
}

static void sum(hg_handle_t h)
{
    hg_return_t ret;
    num_rpcs += 1;

    sum_in_t in;
    sum_out_t out;
    int32_t* values;
    hg_bulk_t local_bulk;
```

(continues on next page)

(continued from previous page)

```
    margo_instance_id mid = margo_hg_handle_get_instance(h);

    const struct hg_info* info = margo_get_info(h);
    hg_addr_t client_addr = info->addr;

    ret = margo_get_input(h, &in);
    assert(ret == HG_SUCCESS);

    values = calloc(in.n, sizeof(*values));
    hg_size_t buf_size = in.n * sizeof(*values);

    ret = margo_bulk_create(mid, 1, (void**)&values, &buf_size,
            HG_BULK_WRITE_ONLY, &local_bulk);
    assert(ret == HG_SUCCESS);

    ret = margo_bulk_transfer(mid, HG_BULK_PULL, client_addr,
            in.bulk, 0, local_bulk, 0, buf_size);
    assert(ret == HG_SUCCESS);

    out.ret = 0;
    int i;
    for(i = 0; i < in.n; i++) {
        out.ret += values[i];
    }

    ret = margo_respond(h, &out);
    assert(ret == HG_SUCCESS);

    ret = margo_bulk_free(local_bulk);
    assert(ret == HG_SUCCESS);

    free(values);

    ret = margo_free_input(h, &in);
    assert(ret == HG_SUCCESS);

    ret = margo_destroy(h);
    assert(ret == HG_SUCCESS);

    if(num_rpcs == TOTAL_RPCS) {
        margo_finalize(mid);
    }
}
DEFINE_MARGO_RPC_HANDLER(sum)
```

Within the RPC handler, after deserializing the RPC's input, we allocate an array of appropriate size:

```
values = calloc(in.n, sizeof(*values));
```

We then expose it the same way as we did on the client side, to get a local bulk handle, using `margo_bulk_create`. This time we specify that this handle will be only written.

`margo_bulk_transfer` is used to do the transfer. Here we pull (`HG_BULK_PULL`) the data from the client's memory to the server's local memory. We provide the client's address (obtained from the *hg_info* structure of the RPC handle), the offset in the client's memory region (here 0) and on the local memory region (0 as well), as well as the size in bytes.

Once the transfer is completed, we perform the sum and return it to the client. We don't forget to use `margo_bulk_free` to free the bulk handle we created (the bulk handle in the `in` structure will be freed by `margo_free_input`, which is why it is so important that this function be called).

5.2.5 Non-blocking RPC

We may sometimes want to send an RPC and carry on some work, checking later whether the RPC has completed. This is done using `margo_iforward` and some other functions that will be described in this tutorial.

Input and output structures

We will take again the example of a *sum* RPC and make the RPC non-blocking. The header bellow is a reminder of what the input and output structures look like.

types.h (show/hide)

```
#ifndef PARAM_H
#define PARAM_H

#include <mercury.h>
#include <mercury_macros.h>

MERCURY_GEN_PROC(sum_in_t,
        ((int32_t)(x))\
        ((int32_t)(y)))

MERCURY_GEN_PROC(sum_out_t, ((int32_t)(ret)))

#endif
```

Non-blocking forward on client

The following code examplifies the use of non-blocking RPC on clients.

client.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <margo.h>
#include "types.h"

int main(int argc, char** argv)
{
    if(argc != 2) {
        fprintf(stderr,"Usage: %s <server address>\n", argv[0]);
        exit(0);
    }

    margo_instance_id mid = margo_init("tcp", MARGO_CLIENT_MODE, 0, 0);
    margo_set_log_level(mid, MARGO_LOG_DEBUG);
    hg_id_t sum_rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, NULL);

    hg_addr_t svr_addr;
    margo_addr_lookup(mid, argv[1], &svr_addr);
```

(continues on next page)

(continued from previous page)

```c
    int i;
    sum_in_t args;
    for(i=0; i<4; i++) {
        args.x = 42+i*2;
        args.y = 42+i*2+1;

        hg_handle_t h;
        margo_create(mid, svr_addr, sum_rpc_id, &h);
        margo_request req;
        margo_iforward(h, &args, &req);

        margo_debug(mid, "Waiting for reply...");

        margo_wait(req);

        sum_out_t resp;
        margo_get_output(h, &resp);

        margo_debug(mid, "Got response: %d+%d = %d", args.x, args.y, resp.ret);

        margo_free_output(h,&resp);
        margo_destroy(h);
    }

    margo_addr_free(mid, svr_addr);

    margo_finalize(mid);

    return 0;
}
```

Instead of using `margo_forward`, we use `margo_iforward`. This function returns immediately after having sent the RPC to the server. It also takes an extra argument of type `margo_request*`. The client will use this request object to check the status of the RPC.

We then use `margo_wait` on the request to block until we have received a response from the server. Alternatively, `margo_test` can be be used to check whether the server has sent a response, without blocking if it hasn't.

Note: It is safe to delete or modify the RPC's input right after the call to `margo_iforward`. `margo_iforward` indeed returns *after* having serialized this input into its send buffer.

Non-blocking response on server

Although generally less useful than non-blocking forwards, non-blocking responses are also available on servers. The `margo_irespond` function can be used for this purpose. It returns as soon as the response has been posted to the Mercury queue.

server.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <margo.h>
#include "types.h"
```

(continues on next page)

(continued from previous page)

```c
static const int TOTAL_RPCS = 16;
static int num_rpcs = 0;

static void sum(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(sum)

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, 0);
    assert(mid);
    margo_set_log_level(mid, MARGO_LOG_INFO);

    hg_addr_t my_address;
    margo_addr_self(mid, &my_address);
    char addr_str[128];
    size_t addr_str_size = 128;
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);
    margo_info(mid, "Server running at address %s", addr_str);

    MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, sum);

    margo_wait_for_finalize(mid);

    return 0;
}

static void sum(hg_handle_t h)
{
    hg_return_t ret;
    num_rpcs += 1;

    sum_in_t in;
    sum_out_t out;

    margo_instance_id mid = margo_hg_handle_get_instance(h);

    ret = margo_get_input(h, &in);
    assert(ret == HG_SUCCESS);

    out.ret = in.x + in.y;
    margo_info(mid, "Computed %d + %d = %d", in.x, in.y, out.ret);

    margo_thread_sleep(mid, 1000);

    margo_request req;

    ret = margo_irespond(h, &out, &req);
    assert(ret == HG_SUCCESS);

    /* ... do other work ... */

    ret = margo_wait(req);
    assert(ret == HG_SUCCESS);

    ret = margo_free_input(h, &in);
```

(continues on next page)

(continued from previous page)

```
    assert(ret == HG_SUCCESS);

    ret = margo_destroy(h);
    assert(ret == HG_SUCCESS);

    if(num_rpcs == TOTAL_RPCS) {
        margo_finalize(mid);
    }
}
DEFINE_MARGO_RPC_HANDLER(sum)
```

Note: `margo_respond` (the blocking version) returns when the response has been sent, but does not guarantees that the client has received it. Its behavior is not very different from `margo_irespond`, which returns as soon as the response has been scheduled for sending. Hence it is unlikely that you ever need `margo_irespond`.

Timeout

Just like there is a `margo_forward_timed`, there is a `margo_iforward_timed`, which takes an additional parameter (before the request pointer) indicating a timeout in millisecond. This timeout applies from the time of the call to `margo_iforward_timed`. Should the server not respond within this time limit, the called to `margo_wait` on the resulting request will return `HG_TIMEOUT`.

5.2.6 Working in terms of providers

In this tutorial, we will learn about providers. This is probably the most important tutorial on Margo since it also describe the design patterns and methodology to use to develop a Margo-based service.

We will take again the *sum* example developped in earlier tutorials, but this time we will give it a proper microservice interface. The result will be composed of two libraries: one for clients, one for servers, as well as their respective headers. We will call this microservice *Alpha*.

Terminology

Although C is not an object-oriented programming language, the best way to understand providers is to think of them an object that can receive RPCs. Rather than targetting a *server*, as we did in previous tutorials, a client's RPC will target a specific *provider* in this server.

Multiple providers belonguing to the same service and living at the same address will expose the same set of RPCs, but each provider will be distinguished from others using its unique *provider id* (an `uint16_t`).

Clients will now use *provider handles* rather than *addresses* to communicate with a particular provider at a given address. A provider handle encapsulates the address of the server as well as the *provider id* of the provider inside this server.

Input and output structures

Our alpha service will use the same input and output structures as in earlier tutorials. We put it here for completeness.

types.h (show/hide)

```
#ifndef PARAM_H
#define PARAM_H

#include <mercury.h>
#include <mercury_macros.h>

MERCURY_GEN_PROC(sum_in_t,
        ((int32_t)(x))\
        ((int32_t)(y)))

MERCURY_GEN_PROC(sum_out_t, ((int32_t)(ret)))

#endif
```

Alpha client interface

First let's start with a header that will be common to the server and the client. This header will simply define the `ALPHA_SUCCESS` and `ALPHA_FAILURE` return codes.

alpha-common.h (show/hide)

```
#ifndef __ALPHA_COMMON_H
#define __ALPHA_COMMON_H

#define ALPHA_SUCCESS  0
#define ALPHA_FAILURE -1

#endif
```

Let's now declare our client interface.

alpha-client.h (show/hide)

```
#ifndef __ALPHA_CLIENT_H
#define __ALPHA_CLIENT_H

#include <margo.h>
#include <alpha-common.h>

#if defined(__cplusplus)
extern "C" {
#endif

typedef struct alpha_client* alpha_client_t;
#define ALPHA_CLIENT_NULL ((alpha_client_t)NULL)

typedef struct alpha_provider_handle *alpha_provider_handle_t;
#define ALPHA_PROVIDER_HANDLE_NULL ((alpha_provider_handle_t)NULL)

/**
 * @brief Creates a ALPHA client.
 *
 * @param[in] mid Margo instance
 * @param[out] client ALPHA client
 *
 * @return ALPHA_SUCCESS or error code defined in alpha-common.h
```

(continues on next page)

(continued from previous page)

```
 */
int alpha_client_init(margo_instance_id mid, alpha_client_t* client);

/**
 * @brief Finalizes a ALPHA client.
 *
 * @param[in] client ALPHA client to finalize
 *
 * @return ALPHA_SUCCESS or error code defined in alpha-common.h
 */
int alpha_client_finalize(alpha_client_t client);

/**
 * @brief Creates a ALPHA provider handle.
 *
 * @param[in] client ALPHA client responsible for the provider handle
 * @param[in] addr Mercury address of the provider
 * @param[in] provider_id id of the provider
 * @param[in] handle provider handle
 *
 * @return ALPHA_SUCCESS or error code defined in alpha-common.h
 */
int alpha_provider_handle_create(
        alpha_client_t client,
        hg_addr_t addr,
        uint16_t provider_id,
        alpha_provider_handle_t* handle);

/**
 * @brief Increments the reference counter of a provider handle.
 *
 * @param handle provider handle
 *
 * @return ALPHA_SUCCESS or error code defined in alpha-common.h
 */
int alpha_provider_handle_ref_incr(
        alpha_provider_handle_t handle);

/**
 * @brief Releases the provider handle. This will decrement the
 * reference counter, and free the provider handle if the reference
 * counter reaches 0.
 *
 * @param[in] handle provider handle to release.
 *
 * @return ALPHA_SUCCESS or error code defined in alpha-common.h
 */
int alpha_provider_handle_release(alpha_provider_handle_t handle);

/**
 * @brief Makes the target ALPHA provider compute the sum of the
 * two numbers and return the result.
 *
 * @param[in] handle provide handle.
 * @param[in] x first number.
 * @param[in] y second number.
 * @param[out] result resulting value.
```

(continues on next page)

(continued from previous page)

```
 *
 * @return ALPHA_SUCCESS or error code defined in alpha-common.h
 */
int alpha_compute_sum(
        alpha_provider_handle_t handle,
        int32_t x,
        int32_t y,
        int32_t* result);

#endif
```

The client interface defines two opaque pointers to structures.

- The `alpha_client_t` handle will be pointing to an object keeping track of registered RPC identifiers for our microservice. An object of this type will be created using `alpha_client_init` and destroyed using `alpha_client_finalize`.
- The `alpha_provider_handle_t` handle will be pointing to a provider handle for a provider of the Alpha service. An object of this type will be created using `alpha_provider_handle_create` function and destroyed using `alpha_provider_handle_release`. This object will have an internal reference count. `alpha_provider_handle_ref_incr` will be used to manually increment this reference count.

The `alpha_compute_sum` function will be in charge of sending a *sum* RPC to the provider designated by the provider handle.

Client implementation

The following code shows the implementation of our client interface.

alpha-client.c (show/hide)

```
#include "types.h"
#include "alpha-client.h"

struct alpha_client {
   margo_instance_id mid;
   hg_id_t           sum_id;
   uint64_t          num_prov_hdl;
};

struct alpha_provider_handle {
    alpha_client_t client;
    hg_addr_t      addr;
    uint16_t       provider_id;
    uint64_t       refcount;
};

int alpha_client_init(margo_instance_id mid, alpha_client_t* client)
{
    int ret = ALPHA_SUCCESS;

    alpha_client_t c = (alpha_client_t)calloc(1, sizeof(*c));
    if(!c) return ALPHA_FAILURE;

    c->mid = mid;
```

(continues on next page)

(continued from previous page)

```
    hg_bool_t flag;
    hg_id_t id;
    margo_registered_name(mid, "alpha_sum", &id, &flag);

    if(flag == HG_TRUE) {
        margo_registered_name(mid, "alpha_sum", &c->sum_id, &flag);
    } else {
        c->sum_id = MARGO_REGISTER(mid, "alpha_sum", sum_in_t, sum_out_t, NULL);
    }

    *client = c;
    return ALPHA_SUCCESS;
}

int alpha_client_finalize(alpha_client_t client)
{
    if(client->num_prov_hdl != 0) {
        margo_warning(client->mid,
            "%d provider handles not released when alpha_client_finalize was called",
            client->num_prov_hdl);
    }
    free(client);
    return ALPHA_SUCCESS;
}

int alpha_provider_handle_create(
        alpha_client_t client,
        hg_addr_t addr,
        uint16_t provider_id,
        alpha_provider_handle_t* handle)
{
    if(client == ALPHA_CLIENT_NULL)
        return ALPHA_FAILURE;

    alpha_provider_handle_t ph =
        (alpha_provider_handle_t)calloc(1, sizeof(*ph));

    if(!ph) return ALPHA_FAILURE;

    hg_return_t ret = margo_addr_dup(client->mid, addr, &(ph->addr));
    if(ret != HG_SUCCESS) {
        free(ph);
        return ALPHA_FAILURE;
    }

    ph->client      = client;
    ph->provider_id = provider_id;
    ph->refcount    = 1;

    client->num_prov_hdl += 1;

    *handle = ph;
    return ALPHA_SUCCESS;
}

int alpha_provider_handle_ref_incr(
        alpha_provider_handle_t handle)
```

(continues on next page)

(continued from previous page)

```c
{
    if(handle == ALPHA_PROVIDER_HANDLE_NULL)
        return ALPHA_FAILURE;
    handle->refcount += 1;
    return ALPHA_SUCCESS;
}

int alpha_provider_handle_release(alpha_provider_handle_t handle)
{
    if(handle == ALPHA_PROVIDER_HANDLE_NULL)
        return ALPHA_FAILURE;
    handle->refcount -= 1;
    if(handle->refcount == 0) {
        margo_addr_free(handle->client->mid, handle->addr);
        handle->client->num_prov_hdl -= 1;
        free(handle);
    }
    return ALPHA_SUCCESS;
}

int alpha_compute_sum(
        alpha_provider_handle_t handle,
        int32_t x,
        int32_t y,
        int32_t* result)
{
    hg_handle_t   h;
    sum_in_t      in;
    sum_out_t   out;
    hg_return_t ret;

    in.x = x;
    in.y = y;

    ret = margo_create(handle->client->mid, handle->addr, handle->client->sum_id, &h);
    if(ret != HG_SUCCESS)
        return ALPHA_FAILURE;

    ret = margo_provider_forward(handle->provider_id, h, &in);
    if(ret != HG_SUCCESS) {
        margo_destroy(h);
        return ALPHA_FAILURE;
    }

    ret = margo_get_output(h, &out);
    if(ret != HG_SUCCESS) {
        margo_destroy(h);
        return ALPHA_FAILURE;
    }

    *result = out.ret;

    margo_free_output(h, &out);
    margo_destroy(h);
    return ALPHA_SUCCESS;
}
```

When initializing the client, `margo_registered_name` is used to check whether the RPC has been defined already. If it has, we use this function to retrieve its id. Otherwise, we use the usual `MARGO_REGISTER` macro.

Notice the use of `margo_provider_forward` in `alpha_compute_sum`, which uses the provider id to send the RPC to a specific provider.

Alpha server interface

Moving on to the server's side, the following code shows how to define the server's interface.

alpha-server.h (show/hide)

```
#ifndef __ALPHA_SERVER_H
#define __ALPHA_SERVER_H

#include <margo.h>
#include <alpha-common.h>

#ifdef __cplusplus
extern "C" {
#endif

#define ALPHA_ABT_POOL_DEFAULT ABT_POOL_NULL

typedef struct alpha_provider* alpha_provider_t;
#define ALPHA_PROVIDER_NULL ((alpha_provider_t)NULL)
#define ALPHA_PROVIDER_IGNORE ((alpha_provider_t*)NULL)

/**
 * @brief Creates a new ALPHA provider. If ALPHA_PROVIDER_IGNORE
 * is passed as last argument, the provider will be automatically
 * destroyed when calling :code:`margo_finalize`.
 *
 * @param[in] mid Margo instance
 * @param[in] provider_id provider id
 * @param[in] pool Argobots pool
 * @param[out] provider provider handle
 *
 * @return ALPHA_SUCCESS or error code defined in alpha-common.h
 */
int alpha_provider_register(
        margo_instance_id mid,
        uint16_t provider_id,
        ABT_pool pool,
        alpha_provider_t* provider);

/**
 * @brief Destroys the Alpha provider and deregisters its RPC.
 *
 * @param[in] provider Alpha provider
 *
 * @return ALPHA_SUCCESS or error code defined in alpha-common.h
 */
int alpha_provider_destroy(
        alpha_provider_t provider);

#ifdef __cplusplus
```

(continues on next page)

(continued from previous page)

```
}
#endif

#endif
```

This interface contains the definition of an opaque pointer type, `alpha_provider_t`, which will be used to hide the implementation of our Alpha provider. Our interface contains the `alpha_provider_register` function, which creates an Alpha provider and registers its RPCs, and the `alpha_provider_destroy` function, which destroys it and deregisters the corresponding RPCs. The former also allows users to pass `ALPHA_PROVIDER_IGNORE` as last argument, when we don't expect to do anything with the provider after registration.

This interface would also be the place where to put other functions that configure or modify the Alpha provider once created.

Note: The `alpha_provider_register` function also takes an Argobots pool as argument. We will discuss this in a following tutorial.

Server implementation

The following code shows the implementation of the interface we just defined.

alpha-server.c (show/hide)

```
#include "alpha-server.h"
#include "types.h"

struct alpha_provider {
    margo_instance_id mid;
    hg_id_t sum_id;
    /* other provider-specific data */
};

static void alpha_finalize_provider(void* p);

DECLARE_MARGO_RPC_HANDLER(alpha_sum_ult);
static void alpha_sum_ult(hg_handle_t h);
/* add other RPC declarations here */

int alpha_provider_register(
        margo_instance_id mid,
        uint16_t provider_id,
        ABT_pool pool,
        alpha_provider_t* provider)
{
    alpha_provider_t p;
    hg_id_t id;
    hg_bool_t flag;

    flag = margo_is_listening(mid);
    if(flag == HG_FALSE) {
        margo_error(mid, "alpha_provider_register(): margo instance is not a server");
        return ALPHA_FAILURE;
    }

```

(continues on next page)

(continued from previous page)

```
    margo_provider_registered_name(mid, "alpha_sum", provider_id, &id, &flag);
    if(flag == HG_TRUE) {
        margo_error(mid, "alpha_provider_register(): a provider with the same␣
↪provider id (%d) already exists", provider_id);
        return ALPHA_FAILURE;
    }

    p = (alpha_provider_t)calloc(1, sizeof(*p));
    if(p == NULL) {
        margo_error(mid, "alpha_provider_register(): failed to allocate memory for␣
↪provider");
        return ALPHA_FAILURE;
    }

    p->mid = mid;

    id = MARGO_REGISTER_PROVIDER(mid, "alpha_sum",
            sum_in_t, sum_out_t,
            alpha_sum_ult, provider_id, pool);
    margo_register_data(mid, id, (void*)p, NULL);
    p->sum_id = id;
    /* add other RPC registration here */

    margo_provider_push_finalize_callback(mid, p, &alpha_finalize_provider, p);

    if(provider)
        *provider = p;
    return ALPHA_SUCCESS;
}

static void alpha_finalize_provider(void* p)
{
    alpha_provider_t provider = (alpha_provider_t)p;
    margo_deregister(provider->mid, provider->sum_id);
    /* deregister other RPC ids ... */
    free(provider);
}

int alpha_provider_destroy(
        alpha_provider_t provider)
{
    /* pop the finalize callback */
    margo_provider_pop_finalize_callback(provider->mid, provider);
    /* call the callback */
    alpha_finalize_provider(provider);

    return ALPHA_SUCCESS;
}


static void alpha_sum_ult(hg_handle_t h)
{
    hg_return_t ret;
    sum_in_t     in;
    sum_out_t   out;

    margo_instance_id mid = margo_hg_handle_get_instance(h);
```

(continues on next page)

(continued from previous page)

```
    const struct hg_info* info = margo_get_info(h);
    alpha_provider_t provider = (alpha_provider_t)margo_registered_data(mid, info->
↪id);

    ret = margo_get_input(h, &in);

    out.ret = in.x + in.y;
    margo_trace(mid, "Computed %d + %d = %d", in.x, in.y, out.ret);

    ret = margo_respond(h, &out);
    ret = margo_free_input(h, &in);
    margo_destroy(h);
}
DEFINE_MARGO_RPC_HANDLER(alpha_sum_ult)
```

We start by defining the `alpha_provider` structure. It may contain the RPC ids as well as any data you may need as context for your RPCs.

The `alpha_provider_register` function starts by checking that the Margo instance is in server mode by using `margo_is_listening`. It then checks that there isn't already an alpha provider with the same id. It does so by using `margo_provider_registered_name` to check whether the *sum* RPC has already been registered with the same provider id.

We then use `MARGO_REGISTER_PROVIDER` instead of `MARGO_REGISTER`. This macro takes a provider id and an Argobots pool in addition to the parameters of `MARGO_REGISTER`.

Finally, we call `margo_provider_push_finalize_callback` to setup a callback that Margo should call when calling `margo_finalize`. This callback will deregister the RPCs and free the provider.

The `alpha_provider_destroy` function is pretty simple but important to understand. In most cases the user will create a provider and leave it running until something calls `margo_finalize`, at which point the provider's finalization callback will be called. If the user wants to destroy the provider before Margo is finalized, it is important to tell Margo not to call the provider's finalization callback when `margo_finalize`. Hence, we use `margo_provider_pop_finalize_callback`. This function takes a Margo instance, and an owner for the callback (here the provider). If the provider registered multiple callbacks using `margo_provider_push_finalize_callback`, `margo_provider_pop_finalize_callback` will pop the last one pushed, and should therefore be called as many time as needed to pop all the finalization callbacks corresponding to the provider.

Warning: Finalization callbacks are called after the Mercury progress loop is terminated. Hence, you cannot send RPCs from them. If you need a finalization callback to be called before the progress loop is terminated, use `margo_push_prefinalize_callback` or `margo_provider_push_prefinalize_callback`.

Using the Alpha client

The previous codes can be compiled into two libraries, *libalpha-client.{a,so}* and *libalpha-server.{a,so}*. The former will be used by client codes to use the Alpha microservice as follows.

client.c (show/hide)

```
#include <stdio.h>
#include <margo.h>
#include <alpha-client.h>
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char** argv)
{
    if(argc != 3) {
        fprintf(stderr,"Usage: %s <server address> <provider id>\n", argv[0]);
        exit(0);
    }

    const char* svr_addr_str = argv[1];
    uint16_t    provider_id  = atoi(argv[2]);

    margo_instance_id mid = margo_init("tcp", MARGO_CLIENT_MODE, 0, 0);
    margo_set_log_level(mid, MARGO_LOG_INFO);

    hg_addr_t svr_addr;
    margo_addr_lookup(mid, svr_addr_str, &svr_addr);

    alpha_client_t alpha_clt;
    alpha_provider_handle_t alpha_ph;

    alpha_client_init(mid, &alpha_clt);

    alpha_provider_handle_create(alpha_clt, svr_addr, provider_id, &alpha_ph);

    int32_t result;
    alpha_compute_sum(alpha_ph, 45, 23, &result);

    alpha_provider_handle_release(alpha_ph);

    alpha_client_finalize(alpha_clt);

    margo_addr_free(mid, svr_addr);

    margo_finalize(mid);

    return 0;
}
```

Notice how simple such an interface is for end users.

Using the Alpha server

A server can be written that spins up an Alpha providervas follows.

server.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <margo.h>
#include <alpha-server.h>

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, 0);
    assert(mid);
    margo_set_log_level(mid, MARGO_LOG_INFO);
```

(continues on next page)

(continued from previous page)

```
    hg_addr_t my_address;
    margo_addr_self(mid, &my_address);
    char addr_str[128];
    size_t addr_str_size = 128;
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);
    margo_info(mid, "Server running at address %s, with provider id 42", addr_str);

    alpha_provider_register(mid, 42, ALPHA_ABT_POOL_DEFAULT, ALPHA_PROVIDER_IGNORE);

    margo_wait_for_finalize(mid);

    return 0;
}
```

A typical Mochi service will consist of a composition of multiple providers spin up in the same program.

Tip: To avoid conflicts with other microservices, it is recommended to prefix the name of the RPCs with the name of the service, as we did here with "alpha_sum".

Note: Providers declaring RPCs with distinct names (i.e. providers from distinct microservices) can have the same provider ids. The provider id is here to distinguish providers of the same type within a given server.

Timeout

The `margo_provider_forward_timed` and `margo_provider_iforward_timed` can be used when sending RPCs (in a blocking or non-blocking manner) to specify a timeout in milliseconds after which the call (or result of `margo_wait`) will be `HG_TIMEOUT`.

5.2.7 Using Argobots pools with Margo RPCs

In the previous tutorial, we saw that the `alpha_provider_register` function is taking an `ABT_pool` argument that is passed down to `MARGO_REGISTER_PROVIDER`.

Argobots pools are a good way to assign resources (typically cores) to particular providers. In the following example, we rewrite the server code in such a way that the Alpha provider gets its own execution stream.

server.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <margo.h>
#include <alpha-server.h>

static void finalize_xstream_cb(void* data);

int main(int argc, char** argv)
{
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, 0);
    assert(mid);
```

(continues on next page)

(continued from previous page)

```
    margo_set_log_level(mid, MARGO_LOG_INFO);

    hg_addr_t my_address;
    margo_addr_self(mid, &my_address);
    char addr_str[128];
    size_t addr_str_size = 128;
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);
    margo_info(mid, "Server running at address %s, with provider id 42", addr_str);

    ABT_pool pool;
    ABT_pool_create_basic(
            ABT_POOL_FIFO,
            ABT_POOL_ACCESS_SPSC,
            ABT_TRUE,
            &pool);

    ABT_xstream xstream;
    ABT_xstream_create_basic(
            ABT_SCHED_DEFAULT,
            1,
            &pool,
            ABT_SCHED_CONFIG_NULL,
            &xstream);

    alpha_provider_register(mid, 42, pool, ALPHA_PROVIDER_IGNORE);

    margo_push_finalize_callback(mid, finalize_xstream_cb, (void*)xstream);

    margo_wait_for_finalize(mid);

    return 0;
}

static void finalize_xstream_cb(void* data) {
    ABT_xstream xstream = (ABT_xstream)data;
    ABT_xstream_join(xstream);
    ABT_xstream_free(&xstream);
}
```

After initializing Margo (which initializes Argobots), we create an Argobots pool and an execution stream that will execute work (ULTs and tasklets) from this pool. We use `ABT_POOL_ACCESS_MPSC` as access type to indicate that there will be multiple producers of work units (in particular, the ES running the Mercury progress loop) and a single consumer of work units (the ES we are about to create).

`ABT_xstream_create_basic` is then used to create the ES. Because Margo is initializing and finalizing Argobots, we need a way to destroy this ES *before* Margo finalizes Argobots. Hence with use `margo_push_finalize_callback` to add a callback that will be called upon finalizing Margo. This callback joins the ES and destroys it.

We pass the newly created pool to the `alpha_provider_register` function, which will make the Alpha provider use this pool to execute its RPC handlers.

For more elaborate assignments of resources to providers, please see the Argobots section of this documentation.

5.2.8 Serializing complex data structures

Let's come back to serializing/deserializing data structures. In previous tutorials, we have always used structures that can be defined using Mercury's `MERCURY_GEN_PROC` macro. If the structure contains pointers, things get more complicated.

Let's assume that we have a type `int_list_t` that represents a pointer to a linked list of integers.

```
typedef struct int_list {
   int32_t          value;
   struct int_list* next;
} *int_list_t;
```

We will need to define a function `hg_return_t hg_proc_int_list_t(hg_proc_t proc, void *data)`. More generally for any custom type `X` that we want to send or receive, and that hasn't been created using the Mercury macro, we need a function of the form `hg_return_t hg_proc_X(hg_proc_t proc, void *data)`.

This function, in our case will be as follows.

types.h (show/hide)

```
#ifndef __TYPES_H
#define __TYPES_H

#include <mercury.h>

typedef struct int_list {
    int32_t          value;
    struct int_list* next;
} *int_list_t;

static inline hg_return_t hg_proc_int_list_t(hg_proc_t proc, void* data)
{
    hg_return_t ret;
    int_list_t* list = (int_list_t*)data;

    hg_size_t length = 0;
    int_list_t tmp    = NULL;
    int_list_t prev   = NULL;

    switch(hg_proc_get_op(proc)) {

        case HG_ENCODE:
            tmp = *list;
            // find out the length of the list
            while(tmp != NULL) {
                tmp = tmp->next;
                length += 1;
            }
            // write the length
            ret = hg_proc_hg_size_t(proc, &length);
            if(ret != HG_SUCCESS)
                break;
            // write the list
            tmp = *list;
            while(tmp != NULL) {
                ret = hg_proc_int32_t(proc, &tmp->value);
```

(continues on next page)

(continued from previous page)

```
                if(ret != HG_SUCCESS)
                    break;
                tmp = tmp->next;
            }
            break;

        case HG_DECODE:
            // find out the length of the list
            ret = hg_proc_hg_size_t(proc, &length);
            if(ret != HG_SUCCESS)
                break;
            // loop and create list elements
            *list = NULL;
            while(length > 0) {
                tmp = (int_list_t)calloc(1, sizeof(*tmp));
                if(*list == NULL) {
                    *list = tmp;
                }
                if(prev != NULL) {
                    prev->next = tmp;
                }
                ret = hg_proc_int32_t(proc, &tmp->value);
                if(ret != HG_SUCCESS)
                    break;
                prev = tmp;
                length -= 1;
            }
            break;

        case HG_FREE:
            tmp = *list;
            while(tmp != NULL) {
                prev = tmp;
                tmp  = prev->next;
                free(prev);
            }
            ret = HG_SUCCESS;
    }
    return ret;
}

#endif
```

Any proc function must have three part, separated by a switch. The `HG_ENCODE` part is used when the `proc` handle is serializing an existing object into a buffer. The `HG_DECODE` part is used when the `proc` handle is creating an new object from the content of its buffer. The `HG_FREE` part is used when freeing the object, e.g. when calling `margo_free_input` or `margo_free_output`.

Note that here the type we are processing is `int_list_t`, so the `void* data` argument is actually a pointer to an `int_list_t`, which is itself a pointer to a structure.

We use the `hg_proc_int32_t` and `hg_proc_hg_size_t` functions to serialize/deserialize `int32_t` and `hg_size_t` respectively. Most basic datatypes have such a function defined in Mercury. To serialize/deserialize raw memory, you can use `hg_proc_raw(hg_proc_t proc, void* data, hg_size_t size)`, which will copy *size* bytes of the content of the memory pointed by *data*.

5.2.9 Using Margo's JSON configuration

In addition to `margo_init`, Margo provides a second initialization function: `margo_init_ext`. This function takes the address of the process (or the protocol to use), the mode (`MARGO_CLIENT_MODE` or `MARGO_SERVER_MODE`), as well as a pointer to a `struct margo_init_info` instance. This instance can be used to provide any of the following.

- `json_config`: a JSON-formatted, null-terminated string;
- `progress_pool`: an existing Argobots pool in which to run the Mercury progress loop;
- `rpc_pool`: an existing Argobots pool in which to run RPC handlers by default;
- `hg_class`: an existing Mercury class;
- `hg_context`: an existing Mercury context;
- `hg_init_info`: an `hg_init_info` structure to pass to Mercury when initializing its class.

The bellow code examplifies the use of the `margo_init_ext` function.

```
#include <assert.h>
#include <stdio.h>
#include <margo.h>

int main(int argc, char** argv)
{
    struct margo_init_info args = {
        .json_config   = NULL, /* const char*           */
        .progress_pool = NULL, /* ABT_pool              */
        .rpc_pool      = NULL, /* ABT_pool              */
        .hg_class      = NULL, /* hg_class_t*           */
        .hg_context    = NULL, /* hg_context_t*         */
        .hg_init_info  = NULL  /* struct hg_init_info* */
    };

    margo_instance_id mid = margo_init_ext("tcp", MARGO_SERVER_MODE, &args);
    assert(mid);
    margo_set_log_level(mid, MARGO_LOG_INFO);

    char* config = margo_get_config(mid);
    margo_info(mid, "%s", config);
    free(config);

    margo_finalize(mid);

    return 0;
}
```

This code also shows the use of the `margo_get_config` function, which returns a JSON string representing the *exact* internal configuration of the Margo instance. When called from this program, for example, we get the following configuration.

```
{
  "progress_timeout_ub_msec":100,
  "enable_profiling":false,
  "enable_diagnostics":false,
  "handle_cache_size":32,
  "profile_sparkline_timeslice_msec":1000,
  "mercury":{
```

(continues on next page)

(continued from previous page)

```
    "version":"2.0.0",
    "request_post_incr":256,
    "request_post_init":256,
    "auto_sm":false,
    "no_bulk_eager":false,
    "no_loopback":false,
    "stats":false,
    "na_no_block":false,
    "na_no_retry":false,
    "max_contexts":1,
    "address":"ofi+tcp;ofi_rxm://10.0.2.15:42925",
    "listening":true
  },
  "argobots":{
    "abt_mem_max_num_stacks":8,
    "abt_thread_stacksize":2097152,
    "version":"1.0",
    "pools":[
      {
        "name":"__primary__",
        "kind":"fifo_wait",
        "access":"mpmc"
      }
    ],
    "xstreams":[
      {
        "name":"__primary__",
        "cpubind":-1,
        "affinity":[
        ],
        "scheduler":{
          "type":"basic_wait",
          "pools":[
            0
          ]
        }
      }
    ]
  },
  "progress_pool":0,
  "rpc_pool":0
}
```

Such a configuration string can be passed as the `json_config` field of the `margo_init_info` structure to reinitialize the margo instance in the exact same manner. It can also be modified to tune Margo's internal configuration.

Important: This configuration is also very useful to provide to Mochi developers whenever you encounter performance issues with Margo or Thallium.

Let's examine this configuration in more details.

Important: None of the configuration fields is mandatory. You may provide a configuration including just the fields you want to change from their default values (shown above).

- `progress_timeout_ub_msec` is the number of milliseconds that will be passed to Mercury's progress function as maximum timeout;
- `enable_profiling` enables internal profiling (extensive statistics about each RPC);
- `enable_diagnostics` enables diagnostics collection (simple statistics);
- `handle_cache_size` is the size of an internal cache that lets Margo reuse RPC handles instead of allocating new ones;
- `profiling_sparkline_timeslice_msec` is the granularity of data collection for sparklines (when profiling is enabled);
- The `mercury` section provides Mercury parameters:
  - `version` will be filled by Margo and does not need to be provided;
  - `request_post_init` is the number of requests for unexpected messages that Mercury will initially post;
  - `request_post_incr` is the increment to the above number of requests, when Mercury runs out of posted requests at any given time;
  - `auto_sm` makes Mercury automatically use shared memory when the sender and receiver processes are on the same node;
  - `no_bulk_eager` prevents Mercury from sending bulk data in RPC if this data is small enough;
  - `no_loopback` prevents a Mercury process from sending RPCs to itself;
  - `stats` enables internal statistics collection;
  - `na_no_block` makes Mercury use busy-spinning instead of blocking on file descriptors;
  - `na_no_retry` prevents Mercury from retrying operations;
  - `max_contexts` is the maximum number of Mercury contexts that can be created;
  - `address` is completed by Margo to provide the process address;
  - `listening` indicates whether the process is listening (server) or not (client);
- The `argobots` section configures the Argobots run time:
  - `abt_mem_max_num_stacks` is the maximum number of pre-allocated stacks;
  - `abt_thread_stacksize` provides the default ULT stack size;
  - `version` is complete by Margo to indicate the version of Argobots in use;
  - `pools` is an array of pool objects. Each pool object has a name (which should be a valid C identifier), a kind (*fifo* or *fifo_wait*), and an access type (*private*, *mpmc*, *spmc*, *spsc*, or *spmc*, indicating multiple or single producers, and multiple or single consumers);
  - `xstreams` is an array of execution streams. Each xstream has a name, a binding to a particular CPU (or -1 for any CPU), and affinity to some CPUs, and a scheduler. The scheduler has a type (*default*, *basic*, *basic_wait*, *prio*, or *randws*) and an array of pools (referenced either by index or by name) that the scheduler is taking work from. Note that one of the xstream must be named "__primary__". If no __primary__ xstream is found by Margo, it will automatically be added, along with a __primary__ pool.
- `progress_pool` is the pool to use for Mercury to run the progress loop. It can be referenced by name or by index. -1 is provided to indicate that the pool is externally provided via the progress_pool field in the margo_init_info structure.
- `rpc_pool` is the pool to use for running RPC handlers by default. It can be referenced by name or by index. -1 is provided to indicate that the pool is externally provided via the rpc_pool field in the margo_init_info structure.

The margo JSON configuration system provides a simple mechanism to initialize and configure a bunch of things, including Mercury and Argobots. Don't hesitate to use it instead of hard-coding these initialization steps, it can greatly help when testing various parameters later on.

Note: This configuration format is also used by Bedrock to standardize Margo's initialization and configuration.

5.2.10 Customizing Margo logging

In the previous tutorials we have used functions like `margo_info` to display informations, usually setting an Margo instance's logging level to `MARGO_LOG_INFO` accordingly. By default, Margo's internal logger will use fprintf to write the messages to `stderr`. It is however possible to inject custom logging functions into a Margo instance. The code bellow showcases how to do this.

server.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <margo.h>

static const int TOTAL_RPCS = 4;
static int num_rpcs = 0;

static void my_trace(void* uargs, const char* str) {
    printf("[trace] %s\n", str);
}

static void my_debug(void* uargs, const char* str) {
    printf("[debug] %s\n", str);
}

static void my_info(void* uargs, const char* str) {
    printf("[info] %s\n", str);
}

static void my_warning(void* uargs, const char* str) {
    printf("[warning] %s\n", str);
}

static void my_error(void* uargs, const char* str) {
    printf("[error] %s\n", str);
}

static void my_critical(void* uargs, const char* str) {
    printf("[critical] %s\n", str);
}

static struct margo_logger custom_logger = {
    .uargs    = NULL,
    .trace    = my_trace,
    .debug    = my_debug,
    .info     = my_info,
    .warning  = my_warning,
    .error    = my_error,
    .critical = my_critical
};
```

(continues on next page)

(continued from previous page)

```c
static void hello_world(hg_handle_t h);
DECLARE_MARGO_RPC_HANDLER(hello_world)

int main(int argc, char** argv)
{
    margo_set_global_log_level(MARGO_LOG_INFO);

    margo_info(MARGO_INSTANCE_NULL,
               "This message uses the global logger");

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 0, -1);
    assert(mid);
    margo_set_logger(mid, &custom_logger);
    margo_set_log_level(mid, MARGO_LOG_INFO);

    margo_info(mid,
               "This message uses an instance's logger");

    hg_addr_t my_address;
    margo_addr_self(mid, &my_address);
    char addr_str[128];
    size_t addr_str_size = 128;
    margo_addr_to_string(mid, addr_str, &addr_str_size, my_address);
    margo_addr_free(mid,my_address);

    margo_info(mid, "Server running at address %s", addr_str);

    hg_id_t rpc_id = MARGO_REGISTER(mid, "hello", void, void, hello_world);
    margo_registered_disable_response(mid, rpc_id, HG_TRUE);

    margo_wait_for_finalize(mid);

    return 0;
}

static void hello_world(hg_handle_t h)
{
    hg_return_t ret;

    margo_instance_id mid = margo_hg_handle_get_instance(h);

    margo_info(mid, "Hello World!");
    num_rpcs += 1;

    ret = margo_destroy(h);
    assert(ret == HG_SUCCESS);

    if(num_rpcs == TOTAL_RPCS) {
        margo_finalize(mid);
    }
}
DEFINE_MARGO_RPC_HANDLER(hello_world)
```

5.3 Thallium

This section will walk you through a series of tutorials on how to use Thallium. We highly recommend to read at least up to the tutorial on providers, which will give you a good picture of how to make a truely object-oriented Mochi service with Thallium.

Thallium also provide a complete object-oriented wrapper for Argobots. One important thing to keep in mind is that these wrappers should be used in place of the C++ threading library.

Important: One of the most frequent source of bug we encounter is developers mistakenly using `std::mutex` or `std::thread` intead of their Thallium counterparts `thallium::mutex` and `thallium::thread`.

5.3.1 Initializing Thallium

Thallium's main class is the engine. It is used to initialize the underlying libraries (Margo, Mercury, and Argobots), to define RPCs, expose segments of memory for RDMA, and lookup addresses.

Server

Here is a simple example of server.

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_SERVER_MODE);
    std::cout << "Server running at address " << myEngine.self() << std::endl;

    return 0;
}
```

You can compile this program with the following command (using GCC):

```
g++ -std=c++14 -o myServer myServer.cpp -lthallium -lmargo -lmercury -labt
```

The first argument of the constructor is the server's protocol (tcp). You can also provide a full address (e.g. *tcp://127.0.0.1:1234*) in particular if you want to force using a particular port number. You can refer to the Mercury documentation to see a list of available protocols. The second argument, `THALLIUM_SERVER_MODE`, indicates that this engine is a server. When running this program, it will print the server's address, then block on the destructor of myEngine. Server engines are indeed supposed to wait for incoming RPCs. We will see in the next tutorial how to properly shut it down.

Client

The following code initialize the engine as a client:

```
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);

    return 0;
}
```

You can compile this program with the following command (using GCC):

```
g++ -std=c++14 -o myServer myServer.cpp -lthallium -lmargo -lmercury -labt
```

Contrary to the server, this program will exit normally. Client engine are not supposed to wait for anything. We use `THALLIUM_CLIENT_MODE` to specify that the engine is a client. We can simply provide the protocol, here "tcp", since a client is not receiving on any address.

5.3.2 Simple Hello World RPC

We will now define a simple RPC handler that prints "Hello World".

Server

Let's take again the server code from the previous section and improve it.

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

void hello(const tl::request& req) {
    std::cout << "Hello World!" << std::endl;
}

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_SERVER_MODE);
    myEngine.define("hello", hello).disable_response();
    std::cout << "Server running at address " << myEngine.self() << std::endl;

    return 0;
}
```

The `engine::define` method is used to define an RPC. The first argument is the name of the RPC (a string), the second is a function. This function should take a const reference to a `thallium::request` as argument. We will see in a future example what this request object is used for. The `disable_response()` method is called to indicate that the RPC is not going to send any response back to the client.

Client

Let's now take a look at the client code.

```
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    if(argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <address>" << std::endl;
        exit(0);
    }

    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure hello = myEngine.define("hello").disable_response();
    tl::endpoint server = myEngine.lookup(argv[1]);
    hello.on(server)();

    return 0;
}
```

The client does not declare the `hello` function, since its code is on the server. Instead, it calls `engine::define` with only the name of the RPC, indicating that there exists on the server a RPC that goes by this name. Again we call `disable_response()` to indicate that this RPC does not send a response back. We then use the engine to perform an address lookup. This call returns an `endpoint` representing the server.

Finally we can call the `hello` RPC by associating it with an `endpoint`. `hello.on(server)` actually returns an instance of a class `callable_remote_procedure` which has its parenthesis operator overloaded to make it usable like a function.

5.3.3 Sending arguments, returning values

In this example we will define a "sum" RPC that will take two integers and return their sum.

Server

Here is the server code.

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

void sum(const tl::request& req, int x, int y) {
    std::cout << "Computing " << x << "+" << y << std::endl;
    req.respond(x+y);
}

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);
    std::cout << "Server running at address " << myEngine.self() << std::endl;
    myEngine.define("sum", sum);

    return 0;
}
```

Notice that our `sum` function now takes two integers in addition to the const reference to a `thallium::request`. You can also see that this request object is used to send a response back to the client. Because the server now sends something back to the client, we do not call `ignore_response()` when defining the RPC.

Client

Let's now take a look at the client code.

```cpp
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {
    if(argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <address>" << std::endl;
        exit(0);
    }
    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure sum = myEngine.define("sum");
    tl::endpoint server = myEngine.lookup(argv[1]);
    int ret = sum.on(server)(42,63);
    std::cout << "Server answered " << ret << std::endl;

    return 0;
}
```

The client calls the remote procedure with two integers and gets an integer back. This way of passing parameters and returning a value hides many implementation details that are handled with a lot of template metaprogramming. Effectively, what happens is the following. When passing the `sum` function to `engine::define`, the compiler deduces from its signature that clients will send two integers. Thus it creates the code necessary to deserialize two integers before calling the function.

On the client side, calling `sum.on(server)(42,63)` makes the compiler realize that the client wants to serialize two integers and send them along with the RPC. It therefore also generates the code for that. The same happens when calling `req.respond(...)` in the server, the compiler generates the code necessary to serialize whatever object has been passed.

Back on the client side, `sum.on(server)(42,63)` does not actually return an integer. It returns an instance of `thallium::packed_response`, which can be cast into any type, here an integer. Asking the `packed_response` to be cast into an integer also instructs the compiler to generate the right deserialization code.

Warning: A common miskate consists of changing the arguments accepted by an RPC handler but forgetting to update the calls to that RPC on clients. This can lead to data corruptions or crashes. Indeed, Thallium has no way to check that the types passed by the client to the RPC call are the ones expected by the server.

Warning: Another common mistake is to use integers of different size on client and server. For example `sum.on(server)(42,63);` on the client side will serialize two int values, because int is the default for integer litterals. If the corresponding RPC handler on the server side had been `void sum(const tl::request& req, int64_t x, int64_t y)`, the call would have led to data corruptions and potential crash. One way to ensure that the right types are used is to explicitely cast the litterals: `sum.on(server)(static_cast<int64_t>(42), static_cast<int64_t>(63));`.

Timeout

It can sometime be useful for an operation to be given a certain amount of time before timing out. This can be done using the `callable_remote_procedure::timed()` function. This function behaves like the `operator()` but takes a first parameter of type `std::chrono::duration` representing an amount of time after which the call will throw a `thallium::timeout` exception. For instance in the above client code, `int ret = sum.on(server)(42,63);` would become `int ret = sum.on(server).timed(std::chrono::milliseconds(5), 42 ,63);` to allow for a 5ms timeout.

5.3.4 Using lambdas and objects to define RPC handlers

So far we have seen how to define an RPC handler using a function. This method is not necessarily the best, since it forces the use of global variables to access anything outside the function.

Fortunately, Thallium can use lambdas as RPC handlers, and even any object that has parenthesis operator overloaded. These objects and lambdas must be converted into `std::function` objects first. Here is how to rewrite the sum server using a lambda.

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);

    std::function<void(const tl::request&, int, int)> sum =
        [](const tl::request& req, int x, int y) {
            std::cout << "Computing " << x << "+" << y << std::endl;
            req.respond(x+y);
        };

    myEngine.define("sum", sum);

    return 0;
}
```

The big advantage of lambdas is their ability to capture local variables, which prevents the use of global variables to pass user-provided data into RPC handlers. This will become handy in the next tutorial. . .

5.3.5 Properly stopping a Thallium server

Stopping a server is done by calling the `engine::finalize()` function. By passing the `engine` object into the closure of a lambda, as a reference, we can make this very easy. For example:

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);

```

(continues on next page)

(continued from previous page)

```
    std::function<void(const tl::request&, int, int)> sum =
        [&myEngine](const tl::request& req, int x, int y) {
            std::cout << "Computing " << x << "+" << y << std::endl;
            req.respond(x+y);
            myEngine.finalize();
        };

    myEngine.define("sum", sum);

    return 0;
}
```

In this version of the server, the server will shut down after the first RPC is received. Note that the engine is captured by reference. Thallium engines are indeed non-copyable.

5.3.6 Sending and returning STL containers

So far we have passed integers as arguments to the RPC handler and returned integers as well. Note that Thallium is able to understand and serialize all arithmetic types, that is, all integer types (char, long, uint64_t, etc.) and floating point types (float, double, etc.). Provided that the type(s) they contain are serializable, Thallium is also capable of serializing all containers of the C++14 Standard Template Library.

List of containers

Here is a full list of containers that Thallium can serialize.

- `std::array<T>`
- `std::complex<T>`
- `std::deque<T>`
- `std::forward_list<T>`
- `std::list<T>`
- `std::map<K,V>`
- `std::multimap<K,V>`
- `std::multiset<T>`
- `std::pair<U,V>`
- `std::set<T>`
- `std::string`
- `std::tuple<T...>`
- `std::unordered_map<K,V>`
- `std::unordered_multimap<K,V>`
- `std::unordered_multiset<T>`
- `std::unordered_set<T>`
- `std::vector<T>`

For instance, Thallium will be able to serialize the following type:

`std::vector<std:tuple<std::pair<int,double>,std::list<int>>>`

Indeed, Thallium knows how to serialize ints and doubles, so it knows how to serialize `std::pair<int, double>` and `std::list<int>`, so it knows how to serialize `std:tuple<std::pair<int,double>, std::list<int>>`, so it knows how to serialize `std::vector<std:tuple<std::pair<int,double>, std::list<int>>>`.

In order for Thallium to know how to serialize a given type, however, you need to include the proper header in the files containing code requiring serialization. For instance to make Thallium understand how to serialize an `std::vector`, you need `#include <thallium/serialization/stl/vector.hpp>`.

The following is a revisited Hello World example in which the client sends its name as an `std::string`.

Server

```
#include <string>
#include <iostream>
#include <thallium.hpp>
#include <thallium/serialization/stl/string.hpp>

namespace tl = thallium;

void hello(const tl::request& req, const std::string& name) {
    std::cout << "Hello " << name << std::endl;
}

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);
    myEngine.define("hello", hello).disable_response();

    return 0;
}
```

Client

```
#include <string>
#include <thallium.hpp>
#include <thallium/serialization/stl/string.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure hello = myEngine.define("hello").disable_response();
    tl::endpoint server = myEngine.lookup("tcp://127.0.0.1:1234");
    std::string name = "Matthieu";
    hello.on(server)(name);

    return 0;
}
```

Note: We explicitly define `std::string name = "Matthieu";` before passing it as an argument. If we were to write `hello.on(server)("Matthieu");`, the compiler would consider `"Matthieu"` as a `const char*` variable, not a `std::string`, and Thallium is not able to serialize pointers. Alternatively, `hello.on(server)(std::string("Matthieu"));` is valid.

5.3.7 Sending and returning custom classes

We have seen previously that all STL containers can be serialized and deserialized by Thallium. Thallium can also serialize and deserialize user-defined classes, with a little help. This help comes in the form of a `serialize` function template put inside the class.

3D point example

Here is an example with a point class representing a 3d point.

```
class point {

    private:

        double x;
        double y;
        double z;

    public:

        point(double a=0.0, double b=0.0, double c=0.0)
        : x(a), y(b), z(c) {}

        template<typename A>
        void serialize(A& ar) {
             ar & x;
             ar & y;
             ar & z;
        }
};
```

You will also need the class to be default-constructible.

The template parameter (A) and the function's parameter (ar) represent an archive, from which one can serialize/deserialize data. Note that you don't need to provide two separate functions: Thallium is smart enough to use the same `serialize` function for both reading and writing, depending on the context.

The `serialize` function calls the operator `&` of the archive to either write or read the class' data members. Provided that those data members are basic types, user-defined types with a `serialize` method, or STL containers of a serializable type, Thallium will know how to serialize the class.

Asymmetric serialization

In some cases it may not be possible for the serialize function to present a symmetric behavior for reading and for writing. For instance this may happen if the deserialization function needs to allocate a pointer. In these cases, one can, instead of a `serialize` method, provide a `save` method and a `load` method. `save` will be used for serialization, `load` will be used for deserialization.

Reading/writing raw data

It may also be convenient to read or write raw data from/to the archive. For this, note that archive objects used for serialization provide a `write` method:

```
template<typename T> write(T* const t, std::size_t count=1)
```

and archive objects used for deserialization provide a `read` method:

```
template<typename T> read(T* t, std::size_t count=1)
```

The `write` method should be used only in a `save` template method, while the `read` method should be used only in a `load` template method.

Non-member serialization functions

In some contexts it may not be possible to add member functions to a class. In those cases, you can add `serialize` or `load`/`save` functions outside of the class, as follows:

```
template<typename A>
void serialize(A& ar, MyType& x) {
  ...
}
```

or

```
template<typename A>
void save(A& ar, const MyType& x) {
  ...
}

template<typename A>
void load(A& ar, MyType& x) {
  ...
}
```

5.3.8 Transferring data over RDMA

In this tutorial, we will learn how to transfer data over RDMA. The class at the core of this tutorial is `thallium::bulk`. This object represents a series of segments of memory within the current process or in a remote process, that is exposed for remote memory accesses.

Client

Here is an example of a client sending a "do_rdma" RPC with a bulk object as argument.

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {
```

(continues on next page)

(continued from previous page)

```
    tl::engine myEngine("tcp", MARGO_CLIENT_MODE);
    tl::remote_procedure remote_do_rdma = myEngine.define("do_rdma").disable_
↪response();
    tl::endpoint server_endpoint = myEngine.lookup("tcp://127.0.0.1:1234");

    std::string buffer = "Matthieu";
    std::vector<std::pair<void*,std::size_t>> segments(1);
    segments[0].first  = (void*)(&buffer[0]);
    segments[0].second = buffer.size()+1;

    tl::bulk myBulk = myEngine.expose(segments, tl::bulk_mode::read_only);

    remote_do_rdma.on(server_endpoint)(myBulk);

    return 0;
}
```

In this client, we define a buffer with the content "Matthieu" (because it's a string, there is actually a null-terminating character). We then define segments as a vector of pairs of `void*` and `std::size_t`. Each segment (here only one) is characterized by its starting address in local memory and its size. We call `engine::expose` to expose the buffer and get a `bulk` instance from it. We specify `tl::bulk_mode::read_only` to indicate that the memory will only be read by other processes (alternatives are `tl::bulk_mode::read_write` and `tl::bulk_mode::write_only`). Finally we send an RPC to the server, passing the bulk object as an argument.

Server

Here is the server code now:

```
#include <iostream>
#include <thallium.hpp>
#include <thallium/serialization/stl/string.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);

    std::function<void(const tl::request&, tl::bulk&)> f =
        [&myEngine](const tl::request& req, tl::bulk& b) {
            tl::endpoint ep = req.get_endpoint();
            std::vector<char> v(6);
            std::vector<std::pair<void*,std::size_t>> segments(1);
            segments[0].first  = (void*)(&v[0]);
            segments[0].second = v.size();
            tl::bulk local = myEngine.expose(segments, tl::bulk_mode::write_only);
            b.on(ep) >> local;
            std::cout << "Server received bulk: ";
            for(auto c : v) std::cout << c;
            std::cout << std::endl;
        };
    myEngine.define("do_rdma",f).disable_response();
}
```

In the RPC handler, we get the client's `endpoint` using `req.get_endpoint()`. We then create a buffer of size 6. We initialize `segments` and expose the buffer to get a `bulk` object from it. The call to the >> operator pulls data from the remote bulk object b and the local bulk object. Since the local bulk is smaller (6 bytes) than the remote one (9 bytes), only 6 bytes are pulled. Hence the loop will print *Matthi*. It is worth noting that an endpoint is needed for Thallium to know in which process to find the memory we are pulling. That's what bulk::on(endpoint) does.

Understanding local and remote bulk objects

A bulk object created using engine::expose is local. When such a bulk object is sent to another process, it becomes remote. Operations can only be done between a local bulk object and a remote bulk object resolved with an endpoint, e.g.,

```
myRemoteBulk.on(myRemoteProcess) >> myLocalBulk;
```

or

```
myLocalBulk >> myRemoteBulk.on(myRemoteProcess);
```

The << operator is, of course, also available.

Transferring subsections of bulk objects

It is possible to select part of a bulk object to be transferred. This is done as follows, for example.

```
myRemoteBulk(3,45).on(myRemoteProcess) >> myLocalBulk(13,45);
```

Here we are pulling 45 bytes of data from the remote bulk starting at offset 3 into the local bulk starting at its offset 13. We have specified 45 as the number of bytes to be transferred. If the sizes had been different, the smallest one would have been picked.

5.3.9 Working in terms of providers

It is often desirable for RPC to target a specific instance of a class on the server side. Classes that can accept RPC requests are called *providers*. A provider object is characterized by a provider id (of type uint16_t). You will need to make sure that no two providers use the same provider id. If they do, they must expose RPC methods with different names (e.g., providers of different services can have the same provider id since they typically don't expose the same RPC names).

Server

The following code sample illustrates a custom provider class, my_sum_provider, which exposes a number of its methods as RPC.

```
#include <iostream>
#include <thallium.hpp>
#include <thallium/serialization/stl/string.hpp>

namespace tl = thallium;

class my_sum_provider : public tl::provider<my_sum_provider> {

    private:

```

(continues on next page)

(continued from previous page)

```
    void prod(const tl::request& req, int x, int y) {
        std::cout << "Computing " << x << "*" << y << std::endl;
        req.respond(x+y);
    }

    int sum(int x, int y) {
        std::cout << "Computing " << x << "+" << y << std::endl;
        return x+y;
    }

    void hello(const std::string& name) {
        std::cout << "Hello, " << name << std::endl;
    }

    int print(const std::string& word) {
        std::cout << "Printing " << word << std::endl;
        return word.size();
    }

    public:

    my_sum_provider(tl::engine& e, uint16_t provider_id=1)
    : tl::provider<my_sum_provider>(e, provider_id) {
        define("prod", &my_sum_provider::prod);
        define("sum", &my_sum_provider::sum);
        define("hello", &my_sum_provider::hello);
        define("print", &my_sum_provider::print, tl::ignore_return_value());
    }

    ~my_sum_provider() {
        get_engine().wait_for_finalize();
    }
};

int main(int argc, char** argv) {

    uint16_t provider_id = 22;
    tl::engine myEngine("tcp", THALLIUM_SERVER_MODE);
    std::cout << "Server running at address " << myEngine.self()
        << " with provider id " << provider_id << std::endl;
    my_sum_provider myProvider(myEngine, provider_id);

    return 0;
}
```

This code defines the `my_sum_provider` class, and creates an instance of this class (passing the `engine` as parameter and a provider id). The `my_sum_provider` class inherits from `thallium::provider<my_sum_provider>` to indicate that this is a provider.

The RPC methods are exposed in the class constructor using the `define` method of the base provider class. Note that multiple definitions of members are possible and exemplified here.

- "prod" is defined the same way as we previously defined RPCs using the engine, with a function that returns `void` and takes a `const thallium::request&` as first parameter.
- "sum" is defined without the `const thallium::request&` parameter. Since it returns an `int`, Thallium will assume that this is what needs to be sent back to the client. It will therefore respond to the client with this return value.

- "hello" does not have a `const thallium::request&` parameter either, and returns `void`. Thallium will implicitly call `.disable_response()` on this RPC to indicate that it does not send a response back to the client.
- "print" does not have a `const thallium::request&` parameter, and returns an `int`. By default Thallium would consider that we want this return value to be sent to the client. To prevents this, we add the `thallium::ignore_return_value()` argument, which indicates Thallium that the function should be treated as if it returned void.

Client

Let's now take a look at the client code.

```
#include <iostream>
#include <thallium/serialization/stl/string.hpp>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {
    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <address> <provider_id>" << std::endl;
        exit(0);
    }
    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure sum   = myEngine.define("sum");
    tl::remote_procedure prod  = myEngine.define("prod");
    tl::remote_procedure hello = myEngine.define("hello").disable_response();
    tl::remote_procedure print = myEngine.define("print").disable_response();
    tl::endpoint server = myEngine.lookup(argv[1]);
    uint16_t provider_id = atoi(argv[2]);
    tl::provider_handle ph(server, provider_id);
    int ret = sum.on(ph)(42,63);
    std::cout << "(sum) Server answered " << ret << std::endl;
    ret = prod.on(ph)(42,63);
    std::cout << "(prod) Server answered " << ret << std::endl;
    std::string name("Matthieu");
    hello.on(ph)(name);
    print.on(ph)(name);

    return 0;
}
```

This client takes a provider id in addition to the server's address. It uses it to define a `thallium::provider_handle` object encapsulating the server address and the provider id. This provider handle is then used in place of the usual `thallium::endpoint` to send RPCs to a specific instance of provider.

Important: We have called `disable_response()` on the "hello" RPC here because there is no way for Thallium to infer here that this RPC does not send a response.

5.3.10 Properly finalizing providers

In the previous tutorial, we have seen how to implement a provider class using Thallium. For convenience in the example, the provider was created on the stack and the call to `wait_for_finalize` was put inside its destructor.

This ensured that the main function would block when the provider instance goes out of scope, and wait for the engine to be finalized.

This design works fine when the program runs only one provider, but it is flawed when we want to work with multiple providers, or when we want to be able to destroy providers before the engine is finalized (e.g. for services that dynamically spawn providers).

In this tutorial, we will look at another design that is more suitable to the general case of multiple providers per process.

Client

The client code is the same as in the previous tutorial, though we provide it here again for convenience. We have also added a call to `shutdown_remote_engine` from the client to shutdown the engine on the server and trigger the engine and providers finalization.

```cpp
#include <iostream>
#include <thallium/serialization/stl/string.hpp>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {
    if(argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <address> <provider_id>" << std::endl;
        exit(0);
    }
    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure sum   = myEngine.define("sum");
    tl::remote_procedure prod  = myEngine.define("prod");
    tl::remote_procedure hello = myEngine.define("hello").disable_response();
    tl::remote_procedure print = myEngine.define("print").disable_response();
    tl::endpoint server = myEngine.lookup(argv[1]);
    uint16_t provider_id = atoi(argv[2]);
    tl::provider_handle ph(server, provider_id);
    int ret = sum.on(ph)(42,63);
    std::cout << "(sum) Server answered " << ret << std::endl;
    ret = prod.on(ph)(42,63);
    std::cout << "(prod) Server answered " << ret << std::endl;
    std::string name("Matthieu");
    hello.on(ph)(name);
    print.on(ph)(name);
    myEngine.shutdown_remote_engine(ph);
    return 0;
}
```

Server

The following code sample illustrates another version of our custom provider class, `my_sum_provider`.

```cpp
#include <iostream>
#include <thallium.hpp>
#include <thallium/serialization/stl/string.hpp>

namespace tl = thallium;

class my_sum_provider : public tl::provider<my_sum_provider> {
```

(continues on next page)

(continued from previous page)

```
private:

tl::remote_procedure m_prod;
tl::remote_procedure m_sum;
tl::remote_procedure m_hello;
tl::remote_procedure m_print;

void prod(const tl::request& req, int x, int y) {
    std::cout << "Computing " << x << "*" << y << std::endl;
    req.respond(x+y);
}

int sum(int x, int y) {
    std::cout << "Computing " << x << "+" << y << std::endl;
    return x+y;
}

void hello(const std::string& name) {
    std::cout << "Hello, " << name << std::endl;
}

int print(const std::string& word) {
    std::cout << "Printing " << word << std::endl;
    return word.size();
}

my_sum_provider(tl::engine& e, uint16_t provider_id=1)
: tl::provider<my_sum_provider>(e, provider_id)
  // keep the RPCs in remote_procedure objects so we can deregister them.
, m_prod(define("prod", &my_sum_provider::prod))
, m_sum(define("sum", &my_sum_provider::sum))
, m_hello(define("hello", &my_sum_provider::hello))
, m_print(define("print", &my_sum_provider::print, tl::ignore_return_value()))
{
    // setup a finalization callback for this provider, in case it is
    // still alive when the engine is finalized.
    get_engine().push_finalize_callback(this, [p=this]() { delete p; });
}

public:

// this factory method and the private constructor prevent users
// from putting an instance  of my_sum_provider on  the stack.
static my_sum_provider* create(tl::engine& e, uint16_t provider_id=1) {
    return new my_sum_provider(e, provider_id);
}

~my_sum_provider() {
    m_prod.deregister();
    m_sum.deregister();
    m_hello.deregister();
    m_print.deregister();
    // pop the finalize callback. If this destructor was called
    // from the finalization callback, there is nothing to pop
    get_engine().pop_finalize_callback(this);
}
```

(continues on next page)

(continued from previous page)

```
};

int main(int argc, char** argv) {

    tl::engine myEngine("tcp", THALLIUM_SERVER_MODE);
    myEngine.enable_remote_shutdown();
    std::cout << "Server running at address " << myEngine.self()
        << " with provider ids 22 and 23 " << std::endl;
    // create a pointer to the provider instance using the factory methods.
    my_sum_provider* myProvider22 = my_sum_provider::create(myEngine, 22);
    my_sum_provider* myProvider23 = my_sum_provider::create(myEngine, 23);

    myEngine.wait_for_finalize();
    // the finalization callbacks will ensure that providers are freed.

    return 0;
}
```

First, we are making the provider's constructor private and force users to use the `create` factory method. This will ensure that any instance of the provider is created on the heap, not on the stack.

Then, we add a bunch of `tl::remote_procedure` fields to keep the registered RPCs. We use these fields to deregister the RPCs in the provider's destructor.

The constructor of the provider also installs a finalization callback in the engine that will call `delete` on the provider's pointer when the engine is finalized. Because we may want to delete the provider ourselves earlier than that, we don't forget to add a call to `pop_finalize_callback` in the destructor.

Important: This design works fine if the provider pushes only one finalization callback, but is flawed if multiple callbacks are pushed by the provider. Suppose the provider pushes two callbacks `f` and `g`, in that order, and `g` calls `delete`. Upon finalizing, the engine will call `g` first, which itself will call `pop_finalize_callback`, which will pop `f` out. Ultimately, `f` will never be called by the engine.

The design presented in this tutorial is only an example of how to better handle the life time of provider objects. Obviously other designs could be envisioned (e.g. with pointers to implementation, or with smart pointers, etc.).

Important: Whatever the design you chose, it is important to rememver that *all thallium resources (mutex, endpoints, etc.) should imperatively be destroyed before the engine itself finalizes.*

5.3.11 Introduction to Argobots wrappers

For convenience, Thallium provides C++ wrappers to all the Argobots functions and structures. We strongly encourage users to familiarize themselves with Argobots to understand how to use execution streams, threads, tasks, pools, etc.

The following code initializes Argobots through Thallium.

```
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {
    // the constructor of the tl::abt object initializes Argobots
```

(continues on next page)

(continued from previous page)

```
    tl::abt a;
    // its destructor finalizes it
    return 0;
}
```

Note: If you are creating a `thallium::engine`, this engine already initializes Argobots in its constructor and finalizes it in its destructor. The code above should therefore be used either if Thallium is used only for its Argobots wrappers (no RPC definition), or if there is a need to initialize Argobots *before* initializing the engine (for example when creating custom Argobots scheduler and pools that the engine will then use).

5.3.12 List of Argobots wrapper classes

Thallium class	Argobot type
barrier	ABT_barrier
condition_variable	ABT_cond
eventual	ABT_eventual
future	ABT_future
mutex	ABT_mutex
pool	ABT_pool
recursive_mutex	ABT_mutex
rwlock	ABT_rwlock
scheduler	ABT_sched
task	ABT_task
thread	ABT_thread
timer	ABT_timer
xstream	ABT_xstream
xstream_barrier	ABT_xstream_barrier

The following class are resource-managed (that is, they create a valid internal Argobot handle when their constructor is called, destroy it when their destructor is called, they are non-copy-constructible, non-copy-assignable, but they are move-constructible and move-assignable): `barrier`, `condition_variable`, `eventual`, `future`, `mutex`, `recursive_mutex`, `rwlock`, `timer`, `xstream_barrier`.

The following classes are only wrappers to Argobots resources but do not destroy them when their destructor is called: `pool`, `scheduler`, `task`, `thread`, `xstream`. To manage the resource inside these classes, you need to use the `managed<T>` template class (e.g. `managed<thread>`). Managed resources are created using the *create()* factory functions in their respective classes, or the *make_thread* and *make_task* functions of the *pool* and *xstream* classes for *managed<thread>* and *managed<task>*.

Warning: An issue with Argobots currently makes *managed<pool>* and *managed<scheduler>* not behave correctly. Their internal resource will be freed automatically if they are default pools or default schedulers. If they are custom pools and schedulers they will not be freed at all.

5.3.13 Using Argobots pools with Thallium RPCs

Thallium allows RPC handlers to be associated with a particular Argobots pool, so that any incoming request for that RPC gets dispatched to the specified pool.

Server

The following code exemplifies using a custom pool in a server.

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

void sum(const tl::request& req, int x, int y) {
    std::cout << "Computing " << x << "+" << y << std::endl;
    req.respond(x+y);
}

int main(int argc, char** argv) {

    tl::abt scope;

    tl::engine myEngine("tcp", THALLIUM_SERVER_MODE);

    std::vector<tl::managed<tl::xstream>> ess;
    tl::managed<tl::pool> myPool = tl::pool::create(tl::pool::access::spmc);
    for(int i=0; i < 4; i++) {
        tl::managed<tl::xstream> es
            = tl::xstream::create(tl::scheduler::predef::deflt, *myPool);
        ess.push_back(std::move(es));
    }
    std::cout << "Server running at address " << myEngine.self() << std::endl;
    myEngine.define("sum", sum, 0, *myPool);

    myEngine.wait_for_finalize();

    for(int i=0; i < 4; i++) {
        ess[i]->join();
    }

    return 0;
}
```

We are explicitly calling `wait_for_finalize()` (which is normally called in the engine's destructor) before joining the execution streams because we don't want the primary ES to be blocking on the `join()` calls.

We are also using a `tl::abt` object to initialize Argobots because this prevents the engine from taking ownership of the Argobots environment and destroy it on the `wait_for_finalize()` call.

Important: This feature requires to provide a non-zero provider id (passed to the define call) when defining the RPC (here 1). Hence you also need to use provider handles on clients, even if you do not define a provider class.

Client

Here is the corresponding client.

```
#include <iostream>
#include <thallium.hpp>
```

(continues on next page)

(continued from previous page)

```
namespace tl = thallium;

int main(int argc, char** argv) {
    if(argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <address>" << std::endl;
        exit(0);
    }
    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure sum = myEngine.define("sum");
    tl::endpoint server = myEngine.lookup(argv[1]);
    int ret = sum.on(server)(42,63);
    std::cout << "Server answered " << ret << std::endl;

    return 0;
}
```

5.3.14 Custom schedulers and pools

Argobots enables you to define custom pools and schedulers. Thallium wraps this feature and makes it object oriented. The best way to define a custom pool and a custom scheduler is to look at the following example.

```
#include <iostream>
#include <unistd.h>
#include <deque>
#include <mutex> // to use std::lock_guard
#include <algorithm>
#include <thallium.hpp>

#define NUM_XSTREAMS 1
#define NUM_THREADS  16

namespace tl = thallium;

class my_unit;
class my_pool;
class my_sched;

class my_unit {

    tl::thread          m_thread;
    tl::task            m_task;
    tl::unit_type       m_type;
    bool                m_in_pool;

    friend class my_pool;

    public:

    my_unit(const tl::thread& t)
    : m_thread(t), m_type(tl::unit_type::thread), m_in_pool(false) {}

    my_unit(const tl::task& t)
    : m_task(t), m_type(tl::unit_type::task), m_in_pool(false) {}

    tl::unit_type get_type() const {
```

(continues on next page)

(continued from previous page)

```
        return m_type;
    }

    const tl::thread& get_thread() const {
        return m_thread;
    }

    const tl::task& get_task() const {
        return m_task;
    }

    bool is_in_pool() const {
        return m_in_pool;
    }

    ~my_unit() = default;
};

class my_pool {

    mutable tl::mutex    m_mutex;
    std::deque<my_unit*> m_units;

    public:

    static const tl::pool::access access_type = tl::pool::access::mpmc;

    my_pool() = default;

    size_t get_size() const {
        std::lock_guard<tl::mutex> lock(m_mutex);
        return m_units.size();
    }

    void push(my_unit* u) {
        std::lock_guard<tl::mutex> lock(m_mutex);
        u->m_in_pool = true;
        m_units.push_back(u);
    }

    my_unit* pop() {
        std::lock_guard<tl::mutex> lock(m_mutex);
        if(m_units.empty())
            return nullptr;
        my_unit* u = m_units.front();
        m_units.pop_front();
        u->m_in_pool = false;
        return u;
    }

    void remove(my_unit* u) {
        std::lock_guard<tl::mutex> lock(m_mutex);
        auto it = std::find(m_units.begin(), m_units.end(), u);
        if(it != m_units.end()) {
            (*it)->m_in_pool = false;
            m_units.erase(it);
        }
```

(continues on next page)

(continued from previous page)

```
    }

    ~my_pool() = default;
};

class my_scheduler : private tl::scheduler {

    public:

    template<typename ... Args>
    my_scheduler(Args&&... args)
    : tl::scheduler(std::forward<Args>(args)...) {}

    void run() {

        int n = num_pools();
        my_unit* unit;
        int target;
        unsigned seed = time(NULL);

        while(true) {
            /* Execute one work unit from the scheduler's pool 0 */
            unit = get_pool(0).pop<my_unit>();
            if(unit != nullptr) {
                get_pool(0).run_unit(unit);
            } else if (n > 1) {
                /* Steal a work unit from other pools */
                target = (n == 2) ? 1 : (rand_r(&seed) % (n-1) + 1);
                unit = get_pool(target).pop<my_unit>();
                if(unit != nullptr)
                    get_pool(target).run_unit(unit);
            }

            if(has_to_stop()) break;

            tl::xstream::check_events(*this);
        }
    }

    tl::pool get_migr_pool() const {
        return get_pool(0);
    }

    ~my_scheduler() = default;
};

void hello() {
    tl::xstream es = tl::xstream::self();
    std::cout << "Hello World from ES "
        << es.get_rank() << ", ULT "
        << tl::thread::self_id()
        << std::endl;
}

int main(int argc, char** argv) {

    tl::abt scope;
```

(continues on next page)

(continued from previous page)

```
    // create pools
    std::vector<tl::managed<tl::pool>> pools;
    for(int i=0; i < NUM_XSTREAMS; i++) {
        pools.push_back(tl::pool::create<my_pool, my_unit>());
    }

    // create schedulers
    std::vector<tl::managed<tl::scheduler>> scheds;
    for(int i=0; i < NUM_XSTREAMS; i++) {
        std::vector<tl::pool> pools_for_sched_i;
        for(int j=0; j < pools.size(); j++) {
            pools_for_sched_i.push_back(*pools[j+i % pools.size()]);
        }
        scheds.push_back(tl::scheduler::create<my_scheduler>(pools_for_sched_i.
↪begin(), pools_for_sched_i.end()));
    }

    std::vector<tl::managed<tl::xstream>> ess;

    for(int i=0; i < NUM_XSTREAMS; i++) {
        tl::managed<tl::xstream> es = tl::xstream::create(*scheds[i]);
        ess.push_back(std::move(es));
    }

    std::vector<tl::managed<tl::thread>> ths;
    for(int i=0; i < NUM_THREADS; i++) {
        tl::managed<tl::thread> th
            = ess[i % ess.size()]->make_thread([]() {
                    hello();
        });
        ths.push_back(std::move(th));
    }

    for(auto& mth : ths) {
        mth->join();
    }

    for(int i=0; i < NUM_XSTREAMS; i++) {
        ess[i]->join();
    }

    return 0;
}
```

5.3.15 Non-blocking RPCs

Non-blocking RPCs are available in Thallium using the `async()` method of the `callable_remote_procedure` objects. In this example, we take again the sum server and show how to use the `async()` function on the client.

Server

As a reminder, here is the server code.

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

void sum(const tl::request& req, int x, int y) {
    std::cout << "Computing " << x << "+" << y << std::endl;
    req.respond(x+y);
}

int main(int argc, char** argv) {

    tl::engine myEngine("tcp://127.0.0.1:1234", THALLIUM_SERVER_MODE);
    std::cout << "Server running at address " << myEngine.self() << std::endl;
    myEngine.define("sum", sum);

    return 0;
}
```

Client

Let's now take a look at the client code.

```
#include <iostream>
#include <thallium.hpp>

namespace tl = thallium;

int main(int argc, char** argv) {
    if(argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <address>" << std::endl;
        exit(0);
    }
    tl::engine myEngine("tcp", THALLIUM_CLIENT_MODE);
    tl::remote_procedure sum = myEngine.define("sum");
    tl::endpoint server = myEngine.lookup(argv[1]);
    auto request = sum.on(server).async(42,63);
    // do something else ...
    // check if request completed
    bool completed =  request.received();
    // ...
    // actually wait on the request and get the result out of it
    int ret = request.wait();
    std::cout << "Server answered " << ret << std::endl;

    return 0;
}
```

Timeout

There is an equivalent `timed_async` function that allows one to specify a timeout value in the form of an `std::chrono::duration` object. If the RPC times out, `wait()` on the resulting `async_response` object will throw a `timeout` exception.

5.4 Mercury

This section contains a series of tutorial on Mercury. These tutorials are aimed at people who want to use Mercury without using Margo or Thallium (e.g., when using another threading library). Note that it is not necessary for Margo and Thallium users to follow these turorials in order to use Margo and Thallium.

5.4.1 Initializing Mercury

In this tutorial you will learn how to initialize Mercury as a client and as a server.

Initializing as a client

The following code exemplifies how to initialize Mercury as a client. We first need to call `HG_Init` to create an `hg_class` instance, then call `HG_Context_create` to create a context.

This code then immediately calls `HG_Context_destroy` and `HG_Finalize` to destroy the context and finalize Mercury, respectively, before terminating.

client.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <mercury.h>

static hg_class_t*     hg_class     = NULL; /* Pointer to the Mercury class */
static hg_context_t*   hg_context   = NULL; /* Pointer to the Mercury context */

int main(int argc, char** argv)
{
    hg_return_t ret;
    /*
     * Initialize an hg_class.
     * Here we only specify the protocal since this is a client
     * (no need for an address and a port). HG_FALSE indicates that
     * the client will not listen for incoming requests.
     */
    hg_class = HG_Init("tcp", HG_FALSE);
    assert(hg_class != NULL);

    /* Creates a context for the hg_class. */
    hg_context = HG_Context_create(hg_class);
    assert(hg_context != NULL);

    /* Destroy the context. */
    ret = HG_Context_destroy(hg_context);
    assert(ret == HG_SUCCESS);

    /* Finalize the hg_class. */
    hg_return_t err = HG_Finalize(hg_class);
    assert(err == HG_SUCCESS);
    return 0;
}
```

Initializing as a server

The following code exemplifies how to initialize Mercury as a server. Just like for a client, we call `HG_Init` and `HG_Context_create`, but this time we pass `HG_TRUE` to `HG_Init` to indicate that this process is going to listen for incoming requests.

server.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <mercury.h>

static hg_class_t*     hg_class   = NULL; /* the mercury class */
static hg_context_t*   hg_context = NULL; /* the mercury context */

int main(int argc, char** argv)
{
    hg_return_t ret;

    /* Initialize Mercury and get an hg_class handle.
     * bmi+tcp is the protocol to use.
     * localhost is the address of the server (not useful at the server itself).
     * HG_TRUE is here to specify that mercury will listen for incoming requests.
     * (HG_TRUE on servers, HG_FALSE on clients).
     */
    hg_class = HG_Init("tcp", HG_TRUE);
    assert(hg_class != NULL);

    /* Get the address of the server */
    char hostname[128];
    hg_size_t hostname_size;
    hg_addr_t self_addr;
    HG_Addr_self(hg_class, &self_addr);
    HG_Addr_to_string(hg_class, hostname, &hostname_size, self_addr);
    printf("Server running at address %s\n",hostname);
    HG_Addr_free(hg_class, self_addr);

    /* Creates a Mercury context from the Mercury class. */
    hg_context = HG_Context_create(hg_class);
    assert(hg_context != NULL);

    /* Progress loop */
    do
    {
        /* count will keep track of how many RPCs were treated in a given
         * call to HG_Trigger.
         */
        unsigned int count;
        do {
            /* Executes callbacks.
             * 0 = no timeout, the function just returns if there is nothing to␣
↪process.
             * 1 = the max number of callbacks to execute before returning.
             * After the call, count will hold the number of callbacks executed.
             */
            ret = HG_Trigger(hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count);
```

(continues on next page)

(continued from previous page)

```
        /* Exit the loop if no event has been processed. */

        /* Make progress on receiving/sending data.
         * 100 is the timeout in milliseconds, for which to wait for network events.␣
↪*/
        HG_Progress(hg_context, 100);
    } while(1); /* another condition should be put here for the loop to terminate */

    /* Destroys the Mercury context. */
    ret = HG_Context_destroy(hg_context);
    assert(ret == HG_SUCCESS);

    /* Finalize Mercury. */
    ret = HG_Finalize(hg_class);
    assert(ret == HG_SUCCESS);

    return 0;
}
```

This code also exemplifies a typical Mercury progress loop. This progress loop alternates `HG_Progress`, which makes progress on network events (sending and receiving data), and `HG_Trigger`, which calls registered callbacks based on the events that happened in `HG_Progress`.

Note: Since this server does not expose any RPC yet, it will just keep running until you kill it.

5.4.2 Simple Hello World RPC

In this tutorial, we will register an RPC that simply prints "Hello World" on the server's standard output.

Server code

The following code shows how to register the RPC on the server.

server.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <mercury.h>

static hg_class_t*     hg_class   = NULL; /* the mercury class */
static hg_context_t*   hg_context = NULL; /* the mercury context */

/* after serving this number of rpcs, the server will shut down. */
static const int TOTAL_RPCS = 10;
/* number of RPCS already received. */
static int num_rpcs = 0;

/*
 * hello_world function to expose as an RPC.
 * This function just prints "Hello World"
 * and increment the num_rpcs variable.
```

(continues on next page)

(continued from previous page)

```
 *
 * All Mercury RPCs must have a signature
 *   hg_return_t f(hg_handle_t h)
 */
hg_return_t hello_world(hg_handle_t h);

/*
 * main function.
 */
int main(int argc, char** argv)
{
    hg_return_t ret;

    if(argc != 2) {
        printf("Usage: %s <protocol>\n", argv[0]);
        exit(0);
    }

    hg_class = HG_Init(argv[1], HG_TRUE);
    assert(hg_class != NULL);

    char hostname[128];
    hg_size_t hostname_size;
    hg_addr_t self_addr;
    HG_Addr_self(hg_class, &self_addr);
    HG_Addr_to_string(hg_class, hostname, &hostname_size, self_addr);
    printf("Server running at address %s\n",hostname);
    HG_Addr_free(hg_class, self_addr);

    hg_context = HG_Context_create(hg_class);
    assert(hg_context != NULL);

    /* Register the RPC by its name ("hello").
     * The two NULL arguments correspond to the functions user to
     * serialize/deserialize the input and output parameters
     * (hello_world doesn't have parameters and doesn't return anything, hence NULL).
     */
    hg_id_t rpc_id = HG_Register_name(hg_class, "hello", NULL, NULL, hello_world);

    /* We call this function to tell Mercury that hello_world will not
     * send any response back to the client.
     */
    HG_Registered_disable_response(hg_class, rpc_id, HG_TRUE);

    do
    {
        unsigned int count;
        do {
            ret = HG_Trigger(hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count);
        HG_Progress(hg_context, 100);
    } while(num_rpcs < TOTAL_RPCS);
    /* Exit the loop if we have reached the given number of RPCs. */

    ret = HG_Context_destroy(hg_context);
    assert(ret == HG_SUCCESS);
```

(continues on next page)

(continued from previous page)

```
    ret = HG_Finalize(hg_class);
    assert(ret == HG_SUCCESS);

    return 0;
}

/* Implementation of the hello_world RPC. */
hg_return_t hello_world(hg_handle_t h)
{
    hg_return_t ret;

    printf("Hello World!\n");
    num_rpcs += 1;
    /* We are not going to use the handle anymore, so we should destroy it. */
    ret = HG_Destroy(h);
    assert(ret == HG_SUCCESS);
    return HG_SUCCESS;
}
```

To register a function as an RPC, it must take an `hg_handle_t` as argument and return a value of type `hg_return_t` (typically `HG_SUCCESS` if the handler executed correctly).

This function (`hello_workd` in our case) is registered as an RPC using `HG_Register_name`. Which returns an identifier for the RPC. We also call `HG_Registered_disable_response` to indicate that this RPC is not going to send any response back to the client.

Inside the definition of `hello_world`, we simply print "Hello World" on the standard output, then call `HG_Destroy` to destroy the RPC handle passed to the function.

Client code

The following is the corresponding client code.

client.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <mercury.h>

static hg_class_t*     hg_class   = NULL; /* Pointer to the Mercury class */
static hg_context_t*   hg_context = NULL; /* Pointer to the Mercury context */
static hg_id_t         hello_rpc_id;      /* ID of the RPC */
static int completed = 0;                 /* Variable indicating if the call has␣
↪completed */

/*
 * This callback will be called after looking up for the server's address.
 * This is the function that will also send the RPC to the servers, then
 * set the completed variable to 1.
 */
hg_return_t lookup_callback(const struct hg_cb_info *callback_info);

int main(int argc, char** argv)
{
    hg_return_t ret;
```

(continues on next page)

(continued from previous page)

```
    if(argc != 3) {
        printf("Usage: %s <protocol> <server_address>\n",argv[0]);
        printf("Example: %s tcp ofi+tcp://1.2.3.4:1234\n",argv[0]);
        exit(0);
    }

    char* protocol = argv[1];
    char* server_address = argv[2];

    hg_class = HG_Init(protocol, HG_FALSE);
    assert(hg_class != NULL);

    hg_context = HG_Context_create(hg_class);
    assert(hg_context != NULL);

    /* Register a RPC function.
     * The first two NULL correspond to what would be pointers to
     * serialization/deserialization functions for input and output datatypes
     * (not used in this example).
     * The third NULL is the pointer to the function (which is on the server,
     * so NULL here on the client).
     */
    hello_rpc_id = HG_Register_name(hg_class, "hello", NULL, NULL, NULL);

    /* Indicate Mercury that we shouldn't expect a response from the server
     * when calling this RPC.
     */
    HG_Registered_disable_response(hg_class, hello_rpc_id, HG_TRUE);

    /* Lookup the address of the server, this is asynchronous and
     * the result will be handled by lookup_callback once we start the progress loop.
     * NULL correspond to a pointer to user data to pass to lookup_callback (we don't␣
↪use
     * any here). The 4th argument is the address of the server.
     * The 5th argument is a pointer a variable of type hg_op_id_t, which identifies␣
↪the operation.
     * It can be useful to get this identifier if we want to be able to cancel it␣
↪using
     * HG_Cancel. Here we don't use it so we pass HG_OP_ID_IGNORE.
     */
    ret = HG_Addr_lookup(hg_context, lookup_callback, NULL, server_address, HG_OP_ID_
↪IGNORE);

    /* Main event loop: we do some progress until completed becomes TRUE. */
    while(!completed)
    {
        unsigned int count;
        do {
            ret = HG_Trigger(hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count && !completed);
        HG_Progress(hg_context, 100);
    }

    ret = HG_Context_destroy(hg_context);
    assert(ret == HG_SUCCESS);
```

(continues on next page)

(continued from previous page)

```
    /* Finalize the hg_class. */
    hg_return_t err = HG_Finalize(hg_class);
    assert(err == HG_SUCCESS);
    return 0;
}

/*
 * This function is called when the address lookup operation has completed.
 */
hg_return_t lookup_callback(const struct hg_cb_info *callback_info)
{
    hg_return_t ret;

    /* First, check that the lookup went fine. */
    assert(callback_info->ret == 0);

    /* Get the address of the server. */
    hg_addr_t addr = callback_info->info.lookup.addr;

    /* Create a call to the hello_world RPC. */
    hg_handle_t handle;
    ret = HG_Create(hg_context, addr, hello_rpc_id, &handle);
    assert(ret == HG_SUCCESS);

    /* Send the RPC. The first NULL correspond to the callback
     * function to call when receiving the response from the server
     * (we don't expect a response, hence NULL here).
     * The second NULL is a pointer to user-specified data that will
     * be passed to the response callback.
     * The third NULL is a pointer to the RPC's argument (we don't
     * use any here).
     */
    ret = HG_Forward(handle, NULL, NULL, NULL);
    assert(ret == HG_SUCCESS);

    /* Free the handle */
    ret = HG_Destroy(handle);
    assert(ret == HG_SUCCESS);

    /* Set completed to 1 so we terminate the loop. */
    completed = 1;
    return HG_SUCCESS;
}
```

Just like in the server, we use `HG_Register_name` to register the RPC, this time passing NULL instead of a function pointer as last argument. We also call `HG_Registered_disable_response` to indicate that the server will not send a response back.

`HG_Addr_lookup` is used to lookup the address of the server. This function takes a callback as its second argument. This callback must be a function that takes a `const struct hg_cb_info*` and return a value of type `hg_return_t`. It will be called when the address lookup completes.

Next, we enter a progress loop similar to that of the server. This is because we are waiting for `HG_Addr_lookup` to complete. The provided callback will be executed from inside `HG_Trigger`.

Inside the `lookup_callback` function, we can the get the address of the server using `callback_info->info.lookup.addr`. This address can be used to create an instance of RPC using `HG_Create`, and forward it using `HG_Forward`.

Since we don't expect any response, we can immediately call HG_Destroy to destroy the RPC handle we just forwarded. We set completed to 1 to exit the progress loop in main.

5.4.3 Passing around a context

The previous tutorial used global static variables to make things like the hg_context and hg_class accessible from within callbacks. Since any good developer would ban such a practice, we will revisit the previous tutorial with local variables instead.

Client

On the client side, we encapsulate a context in the client_data_t structure. By passing a pointer to this structure as third argument of HG_Addr_lookup, we can recover it as callback_info->arg in the callback. This lets us carry the hg_class, hg_context, and hello_rpc_id from main to the lookup_callback function.

client.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <mercury.h>

typedef struct {
    hg_class_t*   hg_class;
    hg_context_t* hg_context;
    hg_id_t       hello_rpc_id;
    int           completed;
} client_data_t;

hg_return_t lookup_callback(const struct hg_cb_info *callback_info);

int main(int argc, char** argv)
{
    hg_return_t ret;

    if(argc != 3) {
        printf("Usage: %s <protocol> <server_address>\n",argv[0]);
        printf("Example: %s tcp ofi+tcp://1.2.3.4:1234\n",argv[0]);
        exit(0);
    }

    client_data_t client_data = {
        .hg_class     = NULL,
        .hg_context   = NULL,
        .hello_rpc_id = 0,
        .completed    = 0
    };

    char* protocol = argv[1];
    char* server_address = argv[2];

    client_data.hg_class = HG_Init(protocol, HG_FALSE);
    assert(client_data.hg_class != NULL);

    client_data.hg_context = HG_Context_create(client_data.hg_class);
```

(continues on next page)

(continued from previous page)

```
    assert(client_data.hg_context != NULL);

    client_data.hello_rpc_id = HG_Register_name(client_data.hg_class, "hello", NULL,␣
↪NULL, NULL);

    HG_Registered_disable_response(client_data.hg_class, client_data.hello_rpc_id, HG_
↪TRUE);

    /* We pass a pointer to the client's data as 3rd argument */
    ret = HG_Addr_lookup(client_data.hg_context, lookup_callback, &client_data,␣
↪server_address, HG_OP_ID_IGNORE);

    while(!client_data.completed)
    {
        unsigned int count;
        do {
            ret = HG_Trigger(client_data.hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count && !client_data.completed);
        HG_Progress(client_data.hg_context, 100);
    }

    ret = HG_Context_destroy(client_data.hg_context);
    assert(ret == HG_SUCCESS);

    hg_return_t err = HG_Finalize(client_data.hg_class);
    assert(err == HG_SUCCESS);
    return 0;
}

hg_return_t lookup_callback(const struct hg_cb_info *callback_info)
{
    hg_return_t ret;
    assert(callback_info->ret == 0);

    /* Get the client's data */
    client_data_t* client_data = (client_data_t*)(callback_info->arg);

    hg_addr_t addr = callback_info->info.lookup.addr;

    hg_handle_t handle;
    ret = HG_Create(client_data->hg_context, addr, client_data->hello_rpc_id, &
↪handle);
    assert(ret == HG_SUCCESS);

    ret = HG_Forward(handle, NULL, NULL, NULL);
    assert(ret == HG_SUCCESS);

    ret = HG_Destroy(handle);
    assert(ret == HG_SUCCESS);

    client_data->completed = 1;
    return HG_SUCCESS;
}
```

Server

On the server side, we encapsulate our information in a `server_data_t` structure. We use `HG_Register_data` to attach a pointer to the structure to the RPC handler (the fourth argument, `NULL`, corresponds to a function to be called to free the pointer when the RPC handler is deregistered. Since our structure is on the stack, we do not need to provide any such function).

Within the `hello_world` handler, we recover the pointer to our `server_data_t` structure by using `HG_Get_info` and `HG_Registered_data`.

server.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <mercury.h>

typedef struct {
    hg_class_t*   hg_class;
    hg_context_t* hg_context;
    int           max_rpcs;
    int           num_rpcs;
} server_data_t;

hg_return_t hello_world(hg_handle_t h);

int main(int argc, char** argv)
{
    hg_return_t ret;

    if(argc != 2) {
        printf("Usage: %s <protocol>\n", argv[0]);
        exit(0);
    }

    server_data_t server_data = {
        .hg_class = NULL,
        .hg_context = NULL,
        .max_rpcs = 4,
        .num_rpcs = 0
    };

    server_data.hg_class = HG_Init(argv[1], HG_TRUE);
    assert(server_data.hg_class != NULL);

    char hostname[128];
    hg_size_t hostname_size;
    hg_addr_t self_addr;
    HG_Addr_self(server_data.hg_class, &self_addr);
    HG_Addr_to_string(server_data.hg_class, hostname, &hostname_size, self_addr);
    printf("Server running at address %s\n",hostname);
    HG_Addr_free(server_data.hg_class, self_addr);

    server_data.hg_context = HG_Context_create(server_data.hg_class);
    assert(server_data.hg_context != NULL);

    hg_id_t rpc_id = HG_Register_name(server_data.hg_class, "hello", NULL, NULL,␣
↪hello_world);
```

(continues on next page)

(continued from previous page)

```
    /* Register data with the RPC handler */
    HG_Register_data(server_data.hg_class, rpc_id, &server_data, NULL);

    HG_Registered_disable_response(server_data.hg_class, rpc_id, HG_TRUE);

    do
    {
        unsigned int count;
        do {
            ret = HG_Trigger(server_data.hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count);
        HG_Progress(server_data.hg_context, 100);
    } while(server_data.num_rpcs < server_data.max_rpcs);

    ret = HG_Context_destroy(server_data.hg_context);
    assert(ret == HG_SUCCESS);

    ret = HG_Finalize(server_data.hg_class);
    assert(ret == HG_SUCCESS);

    return 0;
}

/* Implementation of the hello_world RPC. */
hg_return_t hello_world(hg_handle_t h)
{
    hg_return_t ret;

    /* Get the hg_class_t instance from the handle */
    const struct hg_info *info = HG_Get_info(h);
    hg_class_t* hg_class = info->hg_class;
    hg_id_t     rpc_id   = info->id;

    /* Get the data attached to the RPC handle */
    server_data_t* server_data = (server_data_t*)HG_Registered_data(hg_class, rpc_id);

    printf("Hello World!\n");
    server_data->num_rpcs += 1;

    ret = HG_Destroy(h);
    assert(ret == HG_SUCCESS);
    return HG_SUCCESS;
}
```

5.4.4 RPC arguments and return values

In the previous tutorials we haven't passed nor returned any data to/from the RPC handler. In this tutorial we will see how to send data as arguments to the RPC, and return data from the RPC.

We will take the example of an RPC that computes the sum of two numbers sent by the client.

Input/output structure

First, we need to declare the types of the RPC arguments and return values. This is done using the Mercury macros found in the mercury_macros.h header, as follows.

types.h (show/hide)

```
#ifndef PARAM_H
#define PARAM_H

#include <mercury.h>
#include <mercury_macros.h>

MERCURY_GEN_PROC(sum_in_t,
        ((int32_t)(x))\
        ((int32_t)(y)))

MERCURY_GEN_PROC(sum_out_t, ((int32_t)(ret)))

#endif
```

Client code

The following code looks up the address of the server, then sends an RPC to the server.

client.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <mercury.h>
#include "types.h"

typedef struct {
    hg_class_t*   hg_class;
    hg_context_t* hg_context;
    hg_id_t       sum_rpc_id;
    int           completed;
} client_state_t;

hg_return_t lookup_callback(const struct hg_cb_info *callback_info);
hg_return_t sum_completed(const struct hg_cb_info *info);

int main(int argc, char** argv)
{
    hg_return_t ret;

    if(argc != 3) {
        printf("Usage: %s <protocol> <server_address>\n",argv[0]);
        printf("Example: %s tcp tcp://1.2.3.4:1234\n",argv[0]);
        exit(0);
    }
    char* protocol = argv[1];
    char* server_address = argv[2];

    client_state_t state;
    state.completed = 0;
```

(continues on next page)

(continued from previous page)

```
    state.hg_class = HG_Init(protocol, HG_FALSE);
    assert(state.hg_class != NULL);

    state.hg_context = HG_Context_create(state.hg_class);
    assert(state.hg_context != NULL);

    state.sum_rpc_id = MERCURY_REGISTER(state.hg_class, "sum", sum_in_t, sum_out_t,␣
↪NULL);

    ret = HG_Addr_lookup(state.hg_context, lookup_callback, &state, server_address,␣
↪HG_OP_ID_IGNORE);

    while(!state.completed)
    {
        unsigned int count;
        do {
            ret = HG_Trigger(state.hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count && !state.completed);
        HG_Progress(state.hg_context, 100);
    }

    ret = HG_Context_destroy(state.hg_context);
    assert(ret == HG_SUCCESS);

    hg_return_t err = HG_Finalize(state.hg_class);
    assert(err == HG_SUCCESS);
    return 0;
}


hg_return_t lookup_callback(const struct hg_cb_info *callback_info)
{
    hg_return_t ret;

    /* We get the pointer to the engine_state here. */
    client_state_t* state = (client_state_t*)(callback_info->arg);

    assert(callback_info->ret == 0);
    hg_addr_t addr = callback_info->info.lookup.addr;

    hg_handle_t handle;
    ret = HG_Create(state->hg_context, addr, state->sum_rpc_id, &handle);
    assert(ret == HG_SUCCESS);

    sum_in_t in;
    in.x = 42;
    in.y = 23;

    ret = HG_Forward(handle, sum_completed, state, &in);
    assert(ret == HG_SUCCESS);

    ret = HG_Addr_free(state->hg_class, addr);
    assert(ret == HG_SUCCESS);

    return HG_SUCCESS;
}
```

(continues on next page)

(continued from previous page)

```
hg_return_t sum_completed(const struct hg_cb_info *info)
{
    hg_return_t ret;

    client_state_t* state = (client_state_t*)(info->arg);

    sum_out_t out;
    assert(info->ret == HG_SUCCESS);

    ret = HG_Get_output(info->info.forward.handle, &out);
    assert(ret == HG_SUCCESS);

    printf("Got response: %d\n", out.ret);

    ret = HG_Free_output(info->info.forward.handle, &out);
    assert(ret == HG_SUCCESS);

    ret = HG_Destroy(info->info.forward.handle);
    assert(ret == HG_SUCCESS);

    state->completed = 1;

    return HG_SUCCESS;
}
```

The main difference compared with the previous tutorial is that we pass a pointer to a `sum_in_t` structure to `HG_Forward`, as well as a completion callback `sum_completed`. This completion callback will be called when the server has responded. In this callback, `HG_Get_output` is used to retrieve the output data sent by the server. We need to call `HG_Free_output` to free the output after using it. Note also that `HG_Destroy` is now used in the completion callback rather than after `HG_Forward`.

Server code

server.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <mercury.h>
#include "types.h"

typedef struct {
    hg_class_t*     hg_class;
    hg_context_t*   hg_context;
    int             num_rpcs;
} server_state;

static const int TOTAL_RPCS = 10;

hg_return_t sum(hg_handle_t h);

int main(int argc, char** argv)
{
    hg_return_t ret;

    if(argc != 2) {
```

(continues on next page)

(continued from previous page)

```
        printf("Usage: %s <server address>\n", argv[0]);
        exit(0);
    }

    const char* server_address = argv[1];

    server_state state; // Instance of the server's state
    state.num_rpcs = 0;

    state.hg_class = HG_Init(server_address, HG_TRUE);
    assert(state.hg_class != NULL);

    char hostname[128];
    hg_size_t hostname_size;
    hg_addr_t self_addr;
    HG_Addr_self(state.hg_class, &self_addr);
    HG_Addr_to_string(state.hg_class, hostname, &hostname_size, self_addr);
    printf("Server running at address %s\n",hostname);
    HG_Addr_free(state.hg_class, self_addr);

    state.hg_context = HG_Context_create(state.hg_class);
    assert(state.hg_context != NULL);

    hg_id_t rpc_id = MERCURY_REGISTER(state.hg_class, "sum", sum_in_t, sum_out_t,␣
↪sum);

    /* Attach the local server_state to the RPC so we can get a pointer to it when
     * the RPC is invoked. */
    ret = HG_Register_data(state.hg_class, rpc_id, &state, NULL);

    do
    {
        unsigned int count;
        do {
            ret = HG_Trigger(state.hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count);

        HG_Progress(state.hg_context, 100);
    } while(state.num_rpcs < TOTAL_RPCS);

    ret = HG_Context_destroy(state.hg_context);
    assert(ret == HG_SUCCESS);

    ret = HG_Finalize(state.hg_class);
    assert(ret == HG_SUCCESS);

    return 0;
}

hg_return_t sum(hg_handle_t handle)
{
    hg_return_t ret;
    sum_in_t in;
    sum_out_t out;

    const struct hg_info* info = HG_Get_info(handle);
    server_state* state = HG_Registered_data(info->hg_class, info->id);
```

(continues on next page)

(continued from previous page)

```
    ret = HG_Get_input(handle, &in);
    assert(ret == HG_SUCCESS);

    out.ret = in.x + in.y;
    printf("%d + %d = %d\n",in.x,in.y,in.x+in.y);
    state->num_rpcs += 1;

    ret = HG_Respond(handle,NULL,NULL,&out);
    assert(ret == HG_SUCCESS);

    ret = HG_Free_input(handle, &in);
    assert(ret == HG_SUCCESS);
    ret = HG_Destroy(handle);
    assert(ret == HG_SUCCESS);

    return HG_SUCCESS;
}
```

On the server side, we use `HG_Get_input` to deserialize the input data into a `sum_in_t` structure. We use `HG_Free_input` when we are done with the input data. `HG_Respond` now takes a pointer to a `sum_out_t` object to return to the client.

5.4.5 RDMA transfers

Mercury can use RDMA to transfer large amounts of data. In this tutorial we will demonstrate how to use this feature by transfering the content of a file from a client to a server.

Input/output structures

Like in our earlier examples, we need to define the structures used for RPC inputs and outputs. These are as follows.

types.h (show/hide)

```
#ifndef PARAM_H
#define PARAM_H

#include <mercury.h>
#include <mercury_bulk.h>
#include <mercury_proc_string.h>
#include <mercury_macros.h>

MERCURY_GEN_PROC(save_in_t,
    ((hg_string_t)(filename))\
        ((hg_size_t)(size))\
    ((hg_bulk_t)(bulk_handle)))

MERCURY_GEN_PROC(save_out_t, ((int32_t)(ret)))

#endif
```

The client will send the name of the file (`hg_string_t`), its size (`hg_size_t`), and a bulk handle representing the region of memory exposed by the client and containing the content of the file.

The server will simply respond with an integer indicating whether the operation was succesful.

Client code

The client code is as follows.

client.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <unistd.h>
#include <mercury.h>
#include "types.h"

typedef struct {
    hg_class_t*   hg_class;
    hg_context_t* hg_context;
    hg_id_t       save_rpc_id;
    int           completed;
} client_state;

typedef struct {
    client_state*   state;
    hg_bulk_t       bulk_handle;
    void*           buffer;
    size_t          size;
    char*           filename;
} save_operation;

hg_return_t lookup_callback(const struct hg_cb_info *callback_info);
hg_return_t save_completed(const struct hg_cb_info *info);

int main(int argc, char** argv)
{
    if(argc != 4) {
        fprintf(stderr,"Usage: %s <protocol> <server address> <filename>\n", argv[0]);
        exit(0);
    }

    hg_return_t ret;

    const char* protocol = argv[1];

    /* Local instance of the client_state. */
    client_state state;
    state.completed = 0;
    // Initialize an hg_class.
    state.hg_class = HG_Init(protocol, HG_FALSE);
    assert(state.hg_class != NULL);

    // Creates a context for the hg_class.
    state.hg_context = HG_Context_create(state.hg_class);
    assert(state.hg_context != NULL);

    // Register a RPC function
    state.save_rpc_id = MERCURY_REGISTER(state.hg_class, "save", save_in_t, save_out_
↪t, NULL);

    // Create the save_operation structure
    save_operation save_op;
```

(continues on next page)

(continued from previous page)

```
    save_op.state = &state;
    save_op.filename = argv[3];
    if(access(save_op.filename, F_OK) == -1) {
        fprintf(stderr,"File %s doesn't exist or cannot be accessed.\n",save_op.
↪filename);
        exit(-1);
    }

    char* server_address = argv[2];
    ret = HG_Addr_lookup(state.hg_context, lookup_callback, &save_op, server_address,␣
↪HG_OP_ID_IGNORE);

    // Main event loop
    while(!state.completed)
    {
        unsigned int count;
        do {
            ret = HG_Trigger(state.hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count && !state.completed);
        HG_Progress(state.hg_context, 100);
    }

    // Destroy the context
    ret = HG_Context_destroy(state.hg_context);
    assert(ret == HG_SUCCESS);

    // Finalize the hg_class.
    hg_return_t err = HG_Finalize(state.hg_class);
    assert(err == HG_SUCCESS);
    return 0;
}


hg_return_t lookup_callback(const struct hg_cb_info *callback_info)
{
    hg_return_t ret;

    assert(callback_info->ret == 0);

    /* We get the pointer to the client_state here. */
    save_operation* save_op = (save_operation*)(callback_info->arg);
    client_state* state = save_op->state;

    /* Check file size to allocate buffer. */
    FILE* file = fopen(save_op->filename,"r");
    fseek(file, 0L, SEEK_END);
    save_op->size = ftell(file);
    fseek(file, 0L, SEEK_SET);
    save_op->buffer = calloc(1, save_op->size);
    fread(save_op->buffer,1,save_op->size,file);
    fclose(file);

    hg_addr_t addr = callback_info->info.lookup.addr;
    hg_handle_t handle;
    ret = HG_Create(state->hg_context, addr, state->save_rpc_id, &handle);
    assert(ret == HG_SUCCESS);
```

(continues on next page)

(continued from previous page)

```
    save_in_t in;
    in.filename = save_op->filename;
    in.size     = save_op->size;

    ret = HG_Bulk_create(state->hg_class, 1, (void**) &(save_op->buffer), &(save_op->
↪size),
            HG_BULK_READ_ONLY, &(save_op->bulk_handle));
    assert(ret == HG_SUCCESS);
    in.bulk_handle = save_op->bulk_handle;

    /* The state pointer is passed along as user argument. */
    ret = HG_Forward(handle, save_completed, save_op, &in);
    assert(ret == HG_SUCCESS);

    /* Free the address. */
    ret = HG_Addr_free(state->hg_class, addr);
    assert(ret == HG_SUCCESS);

    return HG_SUCCESS;
}

hg_return_t save_completed(const struct hg_cb_info *info)
{
    hg_return_t ret;

    /* Get the state pointer from the user-provided arguments. */
    save_operation* save_op = (save_operation*)(info->arg);
    client_state* state = (client_state*)(save_op->state);

    save_out_t out;
    assert(info->ret == HG_SUCCESS);

    ret = HG_Get_output(info->info.forward.handle, &out);
    assert(ret == HG_SUCCESS);

    printf("Got response: %d\n", out.ret);

    ret = HG_Bulk_free(save_op->bulk_handle);
    assert(ret == HG_SUCCESS);

    ret = HG_Free_output(info->info.forward.handle, &out);
    assert(ret == HG_SUCCESS);

    ret = HG_Destroy(info->info.forward.handle);
    assert(ret == HG_SUCCESS);

    state->completed = 1;

    return HG_SUCCESS;
}
```

We define a `save_operation` structure to keep information about the on-going operation. This structure will be passed by pointer as user-provided argument to callbacks.

In the lookup callback, we open the file and read its content into a buffer. We then use `HG_Bulk_create` to expose the buffer for RDMA operations. This gives us an `hg_bulk_t` object that can be sent over RPC to the server.

Once the RPC has completed and a response is received, the `hg_bulkt_t` object is freed using `HG_Bulk_free`.

Server code

The following code corresponds to the server.

server.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <mercury.h>
#include "types.h"

/* This structure will encapsulate data about the server. */
typedef struct {
    hg_class_t*     hg_class;
    hg_context_t*   hg_context;
} server_state;

typedef struct {
    char*        filename;
    hg_size_t    size;
    void*        buffer;
    hg_bulk_t    bulk_handle;
    hg_handle_t handle;
} rpc_state;

static hg_return_t save_bulk_completed(const struct hg_cb_info *info);
static hg_return_t save(hg_handle_t h);

int main(int argc, char** argv)
{
    hg_return_t ret;

    if(argc != 2) {
        printf("Usage: %s <server address>\n", argv[0]);
        exit(0);
    }

    const char* server_address = argv[1];

    server_state state; // Instance of the server's state

    state.hg_class = HG_Init(server_address, HG_TRUE);
    assert(state.hg_class != NULL);

    /* Get the address of the server */
    char hostname[128];
    hg_size_t hostname_size;
    hg_addr_t self_addr;
    HG_Addr_self(state.hg_class,&self_addr);
    HG_Addr_to_string(state.hg_class, hostname, &hostname_size, self_addr);
    printf("Server running at address %s\n",hostname);

    state.hg_context = HG_Context_create(state.hg_class);
    assert(state.hg_context != NULL);

    hg_id_t rpc_id = MERCURY_REGISTER(state.hg_class, "save", save_in_t, save_out_t,␣
↪save);
```

(continues on next page)

(continued from previous page)

```
    /* Attach the local server_state to the RPC so we can get a pointer to it when
     * the RPC is invoked. */
    ret = HG_Register_data(state.hg_class, rpc_id, &state, NULL);

    do
    {
        unsigned int count;
        do {
            ret = HG_Trigger(state.hg_context, 0, 1, &count);
        } while((ret == HG_SUCCESS) && count);

        HG_Progress(state.hg_context, 100);
    } while(1);

    ret = HG_Context_destroy(state.hg_context);
    assert(ret == HG_SUCCESS);

    ret = HG_Finalize(state.hg_class);
    assert(ret == HG_SUCCESS);

    return 0;
}

hg_return_t save(hg_handle_t handle)
{
    hg_return_t ret;
    save_in_t in;

    // Get the server_state attached to the RPC.
    const struct hg_info* info = HG_Get_info(handle);
    server_state* stt = HG_Registered_data(info->hg_class, info->id);

    ret = HG_Get_input(handle, &in);
    assert(ret == HG_SUCCESS);

    rpc_state* my_rpc_state = (rpc_state*)calloc(1,sizeof(rpc_state));
    my_rpc_state->handle = handle;
    my_rpc_state->filename = strdup(in.filename);
    my_rpc_state->size = in.size;
    my_rpc_state->buffer = calloc(1,in.size);

    ret = HG_Bulk_create(stt->hg_class, 1, &(my_rpc_state->buffer),
            &(my_rpc_state->size), HG_BULK_WRITE_ONLY, &(my_rpc_state->bulk_handle));
    assert(ret == HG_SUCCESS);

    /* initiate bulk transfer from client to server */
    ret = HG_Bulk_transfer(stt->hg_context, save_bulk_completed,
            my_rpc_state, HG_BULK_PULL, info->addr, in.bulk_handle, 0,
            my_rpc_state->bulk_handle, 0, my_rpc_state->size, HG_OP_ID_IGNORE);
    assert(ret == HG_SUCCESS);

    ret = HG_Free_input(handle, &in);
    assert(ret == HG_SUCCESS);
    return HG_SUCCESS;
}

hg_return_t save_bulk_completed(const struct hg_cb_info *info)
```

(continues on next page)

(continued from previous page)

```
{
    assert(info->ret == 0);

    rpc_state* my_rpc_state = info->arg;
    hg_return_t ret;

    FILE* f = fopen(my_rpc_state->filename,"w+");
    fwrite(my_rpc_state->buffer, 1, my_rpc_state->size, f);
    fclose(f);

    printf("Writing file %s\n", my_rpc_state->filename);

    save_out_t out;
    out.ret = 0;

    ret = HG_Respond(my_rpc_state->handle, NULL, NULL, &out);
    assert(ret == HG_SUCCESS);
    (void)ret;

    HG_Bulk_free(my_rpc_state->bulk_handle);
    HG_Destroy(my_rpc_state->handle);
    free(my_rpc_state->filename);
    free(my_rpc_state->buffer);
    free(my_rpc_state);

    return HG_SUCCESS;
}
```

On the server, the `rpc_state` structure will be used to keep track of information about an on-going operation. In particular, it contains the `hg_handle_t` object of the on-going RPC, and the `hg_bulk_t` object of the local buffer exposed to receive the data.

Upon receiving an RPC, we enter the `save` callback. This function allocates a local buffer to receive the data and exposes it using `HG_Bulk_create`.

We issue the RDMA operation using `HG_Bulk_transfer`, specifying the `HG_BULK_PULL` type of operation, and `save_bulk_completed` as a callback to call once the the RDMA operation has completed. It is important to note that this function returns immediately and the RDMA operation has not be completed at this point. The `save` callback will return and the Mercury progress loop will continue running, eventually calling `save_bulk_completed` when the RDMA operation has finished.

Note that we don't respond to the client in the `save` callback, we do in the `save_bulk_completed` callback, hence the `save` callback does not destroy the RPC's `hg_handle_t` object. This object is kept and freed in `save_bulk_completed`.

Ahhh, callbacks. . . (now you understand how much easier Margo and Thallium are).

5.4.6 Serializing complex data structures

Let's come back to serializing/deserializing data structures. In previous tutorials, we have always used structures that can be defined using Mercury's `MERCURY_GEN_PROC` macro. If the structure contains pointers, things get more complicated.

Let's assume that we have a type `int_list_t` that represents a pointer to a linked list of integers.

```
typedef struct int_list {
   int32_t          value;
   struct int_list* next;
} *int_list_t;
```

We will need to define a function `hg_return_t hg_proc_int_list_t(hg_proc_t proc, void *data)`. More generally for any custom type `X` that we want to send or receive, and that hasn't been created using the Mercury macro, we need a function of the form `hg_return_t hg_proc_X(hg_proc_t proc, void *data)`.

This function, in our case will be as follows.

types.h (show/hide)

```
#ifndef __TYPES_H
#define __TYPES_H

#include <mercury.h>

typedef struct int_list {
    int32_t          value;
    struct int_list* next;
} *int_list_t;

static inline hg_return_t hg_proc_int_list_t(hg_proc_t proc, void* data)
{
    hg_return_t ret;
    int_list_t* list = (int_list_t*)data;

    hg_size_t length = 0;
    int_list_t tmp   = NULL;
    int_list_t prev  = NULL;

    switch(hg_proc_get_op(proc)) {

        case HG_ENCODE:
            tmp = *list;
            // find out the length of the list
            while(tmp != NULL) {
                tmp = tmp->next;
                length += 1;
            }
            // write the length
            ret = hg_proc_hg_size_t(proc, &length);
            if(ret != HG_SUCCESS)
                break;
            // write the list
            tmp = *list;
            while(tmp != NULL) {
                ret = hg_proc_int32_t(proc, &tmp->value);
                if(ret != HG_SUCCESS)
                    break;
                tmp = tmp->next;
            }
            break;

        case HG_DECODE:
            // find out the length of the list
```

(continues on next page)

(continued from previous page)

```
            ret = hg_proc_hg_size_t(proc, &length);
            if(ret != HG_SUCCESS)
                break;
            // loop and create list elements
            *list = NULL;
            while(length > 0) {
                tmp = (int_list_t)calloc(1, sizeof(*tmp));
                if(*list == NULL) {
                    *list = tmp;
                }
                if(prev != NULL) {
                    prev->next = tmp;
                }
                ret = hg_proc_int32_t(proc, &tmp->value);
                if(ret != HG_SUCCESS)
                    break;
                prev = tmp;
                length -= 1;
            }
            break;

        case HG_FREE:
            tmp = *list;
            while(tmp != NULL) {
                prev = tmp;
                tmp  = prev->next;
                free(prev);
            }
            ret = HG_SUCCESS;
    }
    return ret;
}

#endif
```

Any proc function must have three part, separated by a switch. The `HG_ENCODE` part is used when the `proc` handle is serializing an existing object into a buffer. The `HG_DECODE` part is used when the `proc` handle is creating an new object from the content of its buffer. The `HG_FREE` part is used when freeing the object, e.g. when calling `HG_Free_input` or `HG_Free_output`.

Note that here the type we are processing is `int_list_t`, so the `void* data` argument is actually a pointer to an `int_list_t`, which is itself a pointer to a structure.

We use the `hg_proc_int32_t` and `hg_proc_hg_size_t` functions to serialize/deserialize `int32_t` and `hg_size_t` respectively. Most basic datatypes have such a function defined in Mercury. To serialize/deserialize raw memory, you can use `hg_proc_raw(hg_proc_t proc, void* data, hg_size_t size)`, which will copy *size* bytes of the content of the memory pointed by *data*.

5.5 Argobots

The Argobots tutorials are not available yet. For now you will find more information about Argobots here.

5.6 ABT-IO

Executing I/O operations (`read`, `write`, etc.) from an Argobots execution stream blocks this ES until the I/O operation has completed. This is problematic if the ES is used to run a Mercury progress loop, or to execute Mercury RPC handlers, since it prevents any progress on network activities or the execution of other RPC handlers. ABT-IO was developed specifically to address this problem. It creates one or more ES to which I/O operations can be pushed, hence blocking these ES on I/O activities rather than the ES the operations originate from. This section gives a list of tutorials on how to use ABT-IO.

5.6.1 Initializing ABT-IO

The following code shows how to initialize and finalize ABT-IO. ABT-IO must be initialized after Argobots and finalized before it. The `abt_io_init` function takes the number of ES to create for handling I/O operations. Alternatively, the `abt_io_init_pool` function can be used to make ABT-IO use an Argobots pool that was alread created.

```
#include <abt-io.h>

int main(int argc, char** argv)
{
    // Argobots must be initialized, either with ABT_init
    // or with margo_init, thallium::engine, or thallium::abt.
    ABT_init(argc, argv);

    // abt_io_init takes the number of ES to create as a parameter.
    abt_io_instance_id abtio = abt_io_init(2);

    // ABT-IO must be finalized before Argobots it finalized.
    abt_io_finalize(abtio);

    ABT_finalize();

    return 0;
}
```

5.6.2 Issuing I/O operations

Once initialized, ABT-IO can be used to issue I/O operations as follows.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <abt-io.h>

int main(int argc, char** argv)
{
    // Argobots must be initialized, either with ABT_init
    // or with margo_init, thallium::engine, or thallium::abt.
    ABT_init(argc, argv);

    // abt_io_init takes the number of ES to create as a parameter.
    abt_io_instance_id abtio = abt_io_init(2);

```

(continues on next page)

(continued from previous page)

```
    // open a file
    int fd = abt_io_open(abtio, "test.txt", O_WRONLY | O_APPEND | O_CREAT, 0600);

    // write to the file
    abt_io_pwrite(abtio, fd, "This is a test", 14, 0);

    // close the file
    abt_io_close(abtio, fd);

    // ABT-IO must be finalized before Argobots it finalized.
    abt_io_finalize(abtio);

    ABT_finalize();

    return 0;
}
```

The list of available I/O operations is the following.

ABT-IO function	Corresponding POSIX function
abt_io_open	open
abt_io_write	write
abt_io_pwrite	pwrite
abt_io_read	read
abt_io_pread	pread
abt_io_mkostemp	mkostemp
abt_io_unlink	unlink
abt_io_close	close

Important: We highly recommend using the `pwrite` and `pread` wrappers rather than the `write` and `read` wrappers, especially if you either post I/O operations from multiple ES, or setup ABT-IO to execute I/O operations in multiple ES. `abt_io_read` and `abt_io_write` rely on the file descriptor's internal cursor, and Argobots cannot guarantee the order of I/O operations posted, which may lead to unpredictable reordering.

5.6.3 Non-blocking I/O operations

ABT-IO provides an API to issue I/O operations in a non-blocking manner and wait for completion later. The following code examplifies this feature,

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <abt-io.h>

int main(int argc, char** argv)
{
    // Argobots must be initialized, either with ABT_init
    // or with margo_init, thallium::engine, or thallium::abt.
    ABT_init(argc, argv);

    // abt_io_init takes the number of ES to create as a parameter.
```

(continues on next page)

(continued from previous page)

```
    abt_io_instance_id abtio = abt_io_init(2);

    // open a file
    int fd = abt_io_open(abtio, "test.txt", O_WRONLY | O_APPEND | O_CREAT, 0600);

    // write to the file without blocking
    ssize_t ret;
    abt_io_op_t* op = abt_io_pwrite_nb(abtio, fd, "This is a test", 14, 0, &ret);

    // wait for the request to be completed
    abt_io_op_wait(op);

    // free the request
    abt_io_op_free(op);

    // close the file
    abt_io_close(abtio, fd);

    // ABT-IO must be finalized before Argobots it finalized.
    abt_io_finalize(abtio);

    ABT_finalize();

    return 0;
}
```

The full list of non-blocking functions is the following.

ABT-IO function	Corresponding POSIX function
abt_io_open_nb	open
abt_io_pwrite_nb	pwrite
abt_io_write_nb	write
abt_io_pread_nb	pread
abt_io_read_nb	read
abt_io_mkostemp_nb	mkostemp
abt_io_unlink_nb	unlink
abt_io_close_nb	close

Compared with their blocking versions, these function take an additional parameter which is a pointer to the returned value (will be set when the operation has completed) and return a pointer to an `abt_io_op_t` object.

The user can then use `abt_io_op_wait` to wait for the operation to complete, and `abt_io_op_free` to free the operation once completed.

5.7 SSG

SSG is a C library that manages groups of processes. It can be used to monitor group membership for the purport of fault tolerance or dynamic scaling, and to bootstrap a group from other systems like MPI and PMIx. SSG internally uses the SWIM protocol to monitor processes, detect faults, and keep an eventually consistant view of the group using a gossip protocol.

5.7.1 Initializing SSG

The following code sample shows how to initialize and finalize SSG. Though SSG uses Margo for communications, it does not rely on it at initialization time.

main.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <ssg.h>

int main(int argc, char** argv)
{
    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    return 0;
}
```

5.7.2 Creating a group

The sample code hereafter shows how to create an SSG group. The `ssg_group_create()` function requires a margo instance id, a group name, an array of null-terminated strings representing the list of addresses of processes that are members of this group, the number of addresses in this array, a configuration structure, a membership update callback, and a pointer to user data for this callback.

main.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <ssg.h>

static void my_membership_update_cb(void* uargs,
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
{
    switch(update_type) {
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);
```

(continues on next page)

(continued from previous page)

```
    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
    assert(mid);

    hg_addr_t my_addr;
    margo_addr_self(mid, &my_addr);
    char my_addr_str[128];
    size_t my_addr_str_size = 128;
    margo_addr_to_string(mid, my_addr_str, &my_addr_str_size, my_addr);
    margo_addr_free(mid, my_addr);

    const char* group_addr_strs[] = { my_addr_str };
    ssg_group_config_t config = {
        .swim_period_length_ms = 1000,
        .swim_suspect_timeout_periods = 5,
        .swim_subgroup_member_count = -1,
        .ssg_credential = -1
    };

    ssg_group_id_t gid = ssg_group_create(
            mid, "mygroup", group_addr_strs, 1,
            &config, my_membership_update_cb, NULL);

    // ...
    // do stuff using the group
    // ...

    ret = ssg_group_leave(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    return 0;
}
```

In this example we initialize a group of only one process. When multiple processes create a group by this way, all the members of the group have to provide the same input parameters (group name, array of addresses, and configuration).

Important: Because SSG group members have to send messages to each other, they need to be initialized as Margo servers and have an actively running process loop. Anything preventing the progress loop from running will prevent the process from responding in the SWIM protocol, which may lead to the process being marked as dead.

Group configuration

The group configuration structure `ssg_group_config_t` includes the following parameters. * `swim_period_length_ms`: the number of milliseconds between each invokation of the SWIM protocol. * `swim_suspect_timeout_periods`: the number of periods of the SWIM protocol that should pass without a process answering for this process to be marked as *suspected*. * :code: *swim_subgroup_member_count*: when a process A cannot reach a process B directly during the execution of the SWIM protocol, it will ask *swim_subgroup_member_count* to try reaching it on its behalf before considering it suspected. * :code: *ssg_credential*: some credential information.

Group membership callback

The `my_membership_update_cb()` function will be called whenever a membership change is detected. This membership change is indicated by the `ssg_member_update_type_t` argument, and the `ssg_member_id_t` argument indicates which member joined, left, or died.

Destroying a group

The `ssg_group_leave()` function is used to notify other members that the caller is leaving the group, and to free the local resources associated with the group id.

Important: SSG also provides `ssg_group_destroy()`. This function is reserved for processes that have opened a group id (e.g, to observe it) but are not members.

5.7.3 Creating a group from a file

Should the list of members' addresses be available in a text file (one address per line), the following code sample shows how to create a group from this file.

main.c (show/hide)

```c
#include <assert.h>
#include <stdio.h>
#include <ssg.h>

static void my_membership_update_cb(void* uargs,
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
{
    switch(update_type) {
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
    assert(mid);

    ssg_group_config_t config = {
        .swim_period_length_ms = 1000,
        .swim_suspect_timeout_periods = 5,
```

(continues on next page)

(continued from previous page)

```
        .swim_subgroup_member_count = -1,
        .ssg_credential = -1
    };

    ssg_group_id_t gid = ssg_group_create_config(
            mid, "mygroup", "address_list.txt",
            &config, my_membership_update_cb, NULL);

    // ...
    // do stuff using the group
    // ...

    ret = ssg_group_leave(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    return 0;
}
```

5.7.4 Creating a group from MPI

In the context of an MPI application, it is possible to create an SSG group from an MPI comminucator using the `ssg_group_create_mpi` function as follows.

main.c (show/hide)

```
#include <mpi.h>
#include <assert.h>
#include <stdio.h>
#include <ssg.h>
#include <ssg-mpi.h>

static void my_membership_update_cb(void* uargs,
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
{
    switch(update_type) {
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
```

(continues on next page)

(continued from previous page)

```
    MPI_Init(&argc, &argv);

    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
    assert(mid);

    ssg_group_config_t config = {
        .swim_period_length_ms = 1000,
        .swim_suspect_timeout_periods = 5,
        .swim_subgroup_member_count = -1,
        .ssg_credential = -1
    };

    ssg_group_id_t gid = ssg_group_create_mpi(
            mid, "mygroup", MPI_COMM_WORLD,
            &config, my_membership_update_cb, NULL);

    // ...
    // do stuff using the group
    // ...

    ret = ssg_group_leave(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    MPI_Finalize();

    return 0;
}
```

MPI will be used for processes to exchange their address. MPI will *not* be used for subsequent communications in SSG.

5.7.5 Creating a group from PMIx

The Process Management Interface for Exascale (PMIx) provides another way of bootstraping an SSG group. The following code sample illustrates how to do so using the `ssg_group_create_pmix` function.

main.c (show/hide)

```
#include <mpi.h>
#include <assert.h>
#include <stdio.h>
#include <ssg.h>
#include <ssg-pmix.h>

static void my_membership_update_cb(void* uargs,
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
```

(continues on next page)

(continued from previous page)

```
{
    switch(update_type) {
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
    pmix_proc_t proc;
    PMIx_Init(&proc, NULL, 0);

    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
    assert(mid);

    ssg_group_config_t config = {
        .swim_period_length_ms = 1000,
        .swim_suspect_timeout_periods = 5,
        .swim_subgroup_member_count = -1,
        .ssg_credential = -1
    };

    ssg_group_id_t gid = ssg_group_create_pmix(
            mid, "mygroup", proc,
            &config, my_membership_update_cb, NULL);

    // ...
    // do stuff using the group
    // ...

    ret = ssg_group_leave(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    PMIx_Finalize(NULL, 0);

    return 0;
}
```

5.7.6 Joining and leaving a group

In the following example, Process 1 bellow creates a group with only itself as member, then puts the main ULT to sleep for 10 seconds before destroying it and terminating. Mercury communication proceed in the backgroup to detect joining and leaving processes.

The process uses `ssg_group_id_store()` to store information about the group in a file. The first argument of this function is the file name. The second is the group id. The third is the number of current member addresses to include in the file.

proc1.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <ssg.h>

static void my_membership_update_cb(void* uargs,
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
{
    switch(update_type) {
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
    assert(mid);

    hg_addr_t my_addr;
    margo_addr_self(mid, &my_addr);
    char my_addr_str[128];
    size_t my_addr_str_size = 128;
    margo_addr_to_string(mid, my_addr_str, &my_addr_str_size, my_addr);
    margo_addr_free(mid, my_addr);

    const char* group_addr_strs[] = { my_addr_str };
    ssg_group_config_t config = {
        .swim_period_length_ms = 1000,
        .swim_suspect_timeout_periods = 5,
        .swim_subgroup_member_count = 1,
        .ssg_credential = -1
    };

    ssg_group_id_t gid = ssg_group_create(
            mid, "mygroup", group_addr_strs, 1,
```

(continues on next page)

(continued from previous page)

```
            &config, my_membership_update_cb, NULL);

    ret = ssg_group_id_store("mygroup.ssg", gid, 1);

    // ...
    // do stuff using the group
    // ...
    margo_thread_sleep(mid, 10000);

    ret = ssg_group_leave(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    return 0;
}
```

Process 2, whose code is shown bellow, reads the group file using `ssg_group_id_load()`. It then joins the group by calling `ssg_group_join()`. Note that this function takes a membership callback and user-data pointer, but does not include the group name nor the group configuration. These pieces of information are retrieved from the group itself.

The process sleeps for 2 seconds, then calls `ssg_group_leave()` to leave the group. When running this code, you should see Process 1 display messages upon Process 2 joining and leaving.

proc2.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <ssg.h>

static void my_membership_update_cb(void* uargs,
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
{
    switch(update_type) {
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
```

(continues on next page)

(continued from previous page)

```
    assert(mid);

    ssg_group_id_t gid;

    int num_addrs = 1;
    ret = ssg_group_id_load("mygroup.ssg", &num_addrs, &gid);
    assert(ret == SSG_SUCCESS);

    ret = ssg_group_join(mid, gid, my_membership_update_cb, NULL);
    assert(ret == SSG_SUCCESS);

    margo_thread_sleep(mid, 10000);
    // ...
    // do stuff using the group
    // ...

    ret = ssg_group_leave(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    return 0;
}
```

Note that `ssg_group_join` and `ssg_group_leave` both have a variant, `ssg_group_join_target` and `ssg_group_leave_target`, respectively, which can be used to specify to address of another member of the group to inform of the process joining/leaving. The `ssg_group_join` and `ssg_group_leave` functions simply contact a random group member for this purpose.

5.7.7 Observing an SSG group

Warning: The concept of "observer" is not yet fully implemented in SSG. Ultimately, the rational is that observers are processes that can get updated on the group's state, without being part of the group itself. Right now however, this feature remains unimplemented.

It may be useful for a process to be aware of an SSG group (i.e. be able to access the address of its members) without being itself a member of the group. We call such a process an *observer*.

In the code samples bellow, Process 1 creates a group and stays alive for 10 seconds. Process 2 reads the group id from the file created by Process 1, and start observing it using `ssg_group_observe`. It then stops observing it using `ssg_group_unobserve`. Note the use of `ssg_group_destroy` instead of `ssg_group_leave` in the observer process.

proc1.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <ssg.h>

static void my_membership_update_cb(void* uargs,
```

(continues on next page)

(continued from previous page)

```
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
{
    switch(update_type) {
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
    assert(mid);

    hg_addr_t my_addr;
    margo_addr_self(mid, &my_addr);
    char my_addr_str[128];
    size_t my_addr_str_size = 128;
    margo_addr_to_string(mid, my_addr_str, &my_addr_str_size, my_addr);
    margo_addr_free(mid, my_addr);

    const char* group_addr_strs[] = { my_addr_str };
    ssg_group_config_t config = {
        .swim_period_length_ms = 1000,
        .swim_suspect_timeout_periods = 5,
        .swim_subgroup_member_count = -1,
        .ssg_credential = -1
    };

    ssg_group_id_t gid = ssg_group_create(
            mid, "mygroup", group_addr_strs, 1,
            &config, my_membership_update_cb, NULL);

    ret = ssg_group_id_store("mygroup.ssg", gid, 1);

    // ...
    // do stuff using the group
    // ...
    margo_thread_sleep(mid, 10000);

    ret = ssg_group_leave(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

proc2.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <ssg.h>

static void my_membership_update_cb(void* uargs,
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
{
    switch(update_type) {
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
    assert(mid);

    ssg_group_id_t gid;

    int num_addrs = 1;
    ret = ssg_group_id_load("mygroup.ssg", &num_addrs, &gid);
    assert(ret == SSG_SUCCESS);

    ret = ssg_group_observe(mid, gid);
    assert(ret == SSG_SUCCESS);

    ret = ssg_group_unobserve(gid);
    assert(ret == SSG_SUCCESS);

    ret = ssg_group_destroy(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    return 0;
}
```

5.7.8 Using information from a group

Once a process is a member or an observer of a group, a number of operations may be done the resulting group id. The code bellow illustrates most of them.

main.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <ssg.h>

static void my_membership_update_cb(void* uargs,
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
{
    switch(update_type) {
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
    assert(mid);

    hg_addr_t my_addr;
    margo_addr_self(mid, &my_addr);
    char my_addr_str[128];
    size_t my_addr_str_size = 128;
    margo_addr_to_string(mid, my_addr_str, &my_addr_str_size, my_addr);
    margo_addr_free(mid, my_addr);

    const char* group_addr_strs[] = { my_addr_str };
    ssg_group_config_t config = {
        .swim_period_length_ms = 1000,
        .swim_suspect_timeout_periods = 5,
        .swim_subgroup_member_count = -1,
        .ssg_credential = -1
    };

    ssg_group_id_t gid = ssg_group_create(
            mid, "mygroup", group_addr_strs, 1,
            &config, my_membership_update_cb, NULL);

    // get the current process' member id
    ssg_member_id_t self_id = ssg_get_self_id(mid);
    // get the group size
```

(continues on next page)

(continued from previous page)

```
    int size = ssg_get_group_size(gid);
    // get the address from a member id (here self_id)
    hg_addr_t self_addr = ssg_get_group_member_addr(gid, self_id);
    // get the rank of the current process in the group
    int self_rank = ssg_get_group_self_rank(gid);
    // get the rank from a member id (here self_id)
    int rank = ssg_get_group_member_rank(gid, self_id);
    // get a member id from a rank
    ssg_member_id_t member_id = ssg_get_group_member_id_from_rank(gid, rank);
    // get an array of member ids from a range of ranks [0,size[
    ssg_member_id_t member_ids[size];
    ret = ssg_get_group_member_ids_from_range(gid, 0, size, member_ids);
    // get a string address from a rank
    const char* addr_str = ssg_group_id_get_addr_str(gid, 0);

    ret = ssg_group_leave(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);

    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    return 0;
}
```

In a group, a member is uniquely identified by a member id. The member id of a process can be obtained from this process using `ssg_get_self_id()`. A member id can be used to retrieve the corresponding process' address using `ssg_get_group_member_addr()` Member ids of the members in a group are non-contiguous. Hence, each process can be identified with a rank as well, from 0 to *group_size-1*. The rank of the calling process can be obtained using `ssg_get_group_self_rank()`, and functions are available to get a member id from a rank and vis-versa.

Important: While the member id of a process is unique and remains the same regardless of changes to the group, the rank assigned to a process may change over time as processes join and leave the group.

5.7.9 Serializing an SSG group

It may sometimes be useful to serialize a group id into a buffer to send this buffer to another process. The following code sample illustrates how to do this using `ssg_group_id_serialize()` and `ssg_group_id_deserialize()`.

main.c (show/hide)

```
#include <assert.h>
#include <stdio.h>
#include <ssg.h>

static void my_membership_update_cb(void* uargs,
        ssg_member_id_t member_id,
        ssg_member_update_type_t update_type)
{
    switch(update_type) {
```

(continues on next page)

(continued from previous page)

```c
    case SSG_MEMBER_JOINED:
        printf("Member %ld joined\n", member_id);
        break;
    case SSG_MEMBER_LEFT:
        printf("Member %ld left\n", member_id);
        break;
    case SSG_MEMBER_DIED:
        printf("Member %ld died\n", member_id);
        break;
    }
}

int main(int argc, char** argv)
{
    int ret = ssg_init();
    assert(ret == SSG_SUCCESS);

    margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, 1, 0);
    assert(mid);

    hg_addr_t my_addr;
    margo_addr_self(mid, &my_addr);
    char my_addr_str[128];
    size_t my_addr_str_size = 128;
    margo_addr_to_string(mid, my_addr_str, &my_addr_str_size, my_addr);
    margo_addr_free(mid, my_addr);

    const char* group_addr_strs[] = { my_addr_str };
    ssg_group_config_t config = {
        .swim_period_length_ms = 1000,
        .swim_suspect_timeout_periods = 5,
        .swim_subgroup_member_count = -1,
        .ssg_credential = -1
    };

    ssg_group_id_t gid = ssg_group_create(
            mid, "mygroup", group_addr_strs, 1,
            &config, my_membership_update_cb, NULL);

    // serialization into a buffer
    char* buf = NULL;
    size_t buf_size = 0;
    ssg_group_id_serialize(gid, 1, &buf, &buf_size);

    // deserialization from a buffer
    ssg_group_id_t gid2;
    int num_addrs = 1;
    ssg_group_id_deserialize(buf, buf_size, &num_addrs, &gid2);

    // the buffer needs to be freed after serialization
    free(buf);

    ret = ssg_group_leave(gid);
    assert(ret == SSG_SUCCESS);

    margo_finalize(mid);
```

(continues on next page)

(continued from previous page)

```
    ret = ssg_finalize();
    assert(ret == SSG_SUCCESS);

    return 0;
}
```

Note: The `ssg_group_id_serialize` takes a pointer to a buffer and will perform an allocation of this buffer. It is the caller's responsibility to free this buffer.

5.8 Templates

To help users develop their own Mochi microservices, we provide two templates: one in C using Margo, the other in C++ using Thallium. Feel free to jump to either of them depending on your preferred language.

5.8.1 Margo microservice template (C)

The Margo microservice template is available here. Though this project provides many examples of how to use the Margo API, you may want to refer to the Margo documentation for more detail.

The Mochi philosophy

The philosophy of the Mochi project consists of providing a set of building blocks for developing HPC data service. Each building block is meant to offer **efficient**, **location-agnostic** access to a simple set of functionalities through **modular backends**, while **seamlessly sharing hardware** (compute and network) with other building blocks.

A **simple set of functionalities** may be, for example, *"storing and retrieving small key/value pairs"*, a common feature found in many storage systems to manage metadata. The **location-agnostic** aspect aims at making such a feature available to user programs in the same manner and through the same API regardless of whether the service runs in the same process, on the same node but on different processes, or on a different node across a network. The **modular backends** aspect makes it possible to abstract the implementation of such a feature and, if not providing multiple implementations, at least providing the means for someone to easily swap the default implementation of the feature for their own. Because multiple building blocks may be running on the same node, such a building block should efficiently share the compute and network hardware with other building blocks. This is done by sharing a common networking and threading layer: Margo.

Design overview

The typical design of a Mochi microservice revolves around four libraries: *server*, *client*, *admin*, and *bedrock module*.

- The server library contains a service **provider**, that is, an object that can receive some predefined RPCs to offer a particular functionality. Within the same process, multiple providers of the same service may be instantiated, using distinct **provider ids** (*uint16_t*). A provider is responsible for managing a set of **resources**. In the example of a storage for key/value pairs, a resource may be a database. The functionalities of a provider may be enabled by multiple **backends**. For example, a database may be implemented using LevelDB, BerkeleyDB, or simply using an in-memory hash table. Programs sending requests to a provider should **not** be aware of the backend used to implement the requested functionality. This allows multiple backends to be tested, and for backends to evolve independently from user applications.

- The **client** library is the library through which user applications or higher-level services interact with providers. It will typically provide a client structure that is used to register the set of RPCs that can be invoked, and a **resource handle** structure that references a particular resource located on a particular provider. User applications will typically initialize a singe client object for a service, and from this client object instantiate as many resource handles as needed to interact with available resources. Resources are identified by a **resource id**, which are generally either a name, an integer, or a uuid (this template project uses uuids).
- The **admin** library is the library through which a user application can send requests that are meant for the provider itself rather than for a resource. A few most common such requests include the creation and destruction of resources, their migration, etc. It can be useful to think of the admin library as the set of features you would want to provide to the person or application that sets up the service, rather than the person or application that uses its functionalities.
- The **bedrock module** library enables using your component with Bedrock. It is implemented in *src/bedrock-module.c*.

Organization of the template project

The template project illustrates how a Margo-based microservice could be architected. It can be compiled as-is, and provides a couple of functionalities that make the provider print a "Hello World" message on its standard output, or compute the sum of two integers.

This template project uses *alpha* as the name of your microservice. Functions, types, files, and libraries therefore use the *alpha* prefix. The first step in setting up this project for your microservice will be to replace this prefix. The generic name *resource* should also be replaced with a more specific name, such as *database*. This renaming step can be done by using the *setup.py* script at the root of this repository (see next section).

The *include* directory of this template project provides public header files.

- *alpha/alpha-common.h* contains APIs that are common to the three libraries, such as error codes or common types;
- *alpha/alpha-client.h* contains the client-side functions to create and destroy a client object;
- *alpha/alpha-resource.h* contains the client-side functions to create and destroy resource handles, and to interact with a resource through a resource handle;
- *alpha/alpha-server.h* contains functions to register and destroy a provider;
- *alpha/alpha-backend.h* contains the definition of a structure that one would need to implement in order to provide a new backend for your microservice;
- *alpha/alpha-admin.h* contains the functions to create and destroy an admin object, as well as admin functions to interact with a provider;
- *alpha/alpha-provider-handle.h* contains the definition of a provider handle. This type of construct is often used in Mochi services to encapsulate an address and a provider id.

The implementation of all these functions is located in the *src* directory. The source also includes functionalities such as a small header-based logging library. The *src/dummy* directory provides a default implementation of a backend. This backend also exemplifies the use of the [json-c](https://github.com/json-c/json-c) library for JSON-based resource configuration. We recommend that you implement a dummy backend for your service, as a way of testing application logic and RPCs without the burden of complex external dependencies. For instance, a dummy backend may be a backend that simply acknowledges requests but does not process them, or provides mock results.

The *examples* directory contains an example using the microservice: the server example will start a provider and print its address (if logging was enabled). The admin example will connect to this provider and have it create a resource, then print the resource id. The client example can be run next to interact with the resource.

The *tests* directory contains a set of unit tests for your service. It relies on [µnit](https://nemequ.github.io/munit) (included in the repository), a C unit-test library under an MIT license. Feel free to continue using it as you add more functionalities to your microservice; unit-testing is just good software development practice in general.

The template also contains a *spack.yaml* file at its root that can be used to install its dependencies. You may add additional dependencies into this file as your microservice gets more complex.

As you modify this project to implement your own microservice, feel free to remove any dependencies you don't like (such as json-c or µnit) and adapt it to your needs!

Setting up your project

Let's assume you want to create a microservice called "yellow", which manages a phone directory (association between names and phone numbers). The following shows how to setup your project:

```
git clone https://xgitlab.cels.anl.gov/sds/templates/margo-microservice-template.git
mv margo-microservice-template yellow
cd yellow
rm -rf .git
python setup.py
$ Enter the name of your service: yellow
$ Enter the name of the resources (e.g., database): phonebook
```

The python script will edit and rename all the files, replacing *alpha* with *yellow* and *resource* with *phonebook*.

Note: The *setup.py* script requires Python 3.

Building the project

The project's dependencies may be build using spack. You will need to have setup sds-repo as external namespace for spack, which can be done as follows.

```
# from outside of your project directory
git clone git@xgitlab.cels.anl.gov:sds/sds-repo.git
spack repo add sds-repo
```

The easiest way to setup the dependencies for this project is to create a spack environment using the *spack.yaml* file located at the root of the project, as follows.

```
# create an anonymous environment
cd margo-microservice-template
spack env activate .
spack install
```

or as follows.

```
# create an environment named myenv
cd margo-microservice-template
spack env create myenv spack.yaml
spack env activate myenv
spack install
```

Once the dependencies have been installed, you may build the project as follows.

```
mkdir build
cd build
cmake .. -DENABLE_TESTS=ON -DENABLE_EXAMPLES=ON -DENABLE_BEDROCK=ON
make
```

You can test the project using `make test` from the build directory.

5.8.2 Thallium microservice template (C++)

The Thallium microservice template is available here. Though this project provides many examples of how to use the Thallium API, you may want to refer to the Thallium documentation for more detail.

The Mochi philosophy and design overview

Please refer to the margo template documation to have an overview of the Mochi philosophy and the design of a Mochi microservice.

Organization of the template project

This template project illustrates how a Thallium-based microservice could be architectured. It can be compiled as-is, and provides a couple of functionalities that make the provider print a "Hello World" message on its standard output, or compute the sum of two integers.

This template project uses **alpha** as the name of your microservice. Functions, types, files, and libraries therefore use the **alpha** prefix. The first step in setting up this project for your microservice will be to replace this prefix. The generic name **resource** should also be replaced with a more specific name, such as **database**. This renaming step can be done by using the *setup.py* script at the root of this repository (see next section).

The *include* directory of this template project provides public header files.

- *alpha/Client.hpp* contains the Client class;
- *alpha/ResourceHandle.hpp* contains the ResourceHandle class, which provides functions to interact with a resource on a provider;
- *alpha/Provider.hpp* contains the Provider class;
- *alpha/Backend.hpp* contains the definition of an abstract clas that can be inherited from to implement new backends for the microservice.
- *alpha/Admin.hpp* contains the Admin class, as well as admin functions to interact with a provider.

This template project uses the [pimpl idiom](https://en.cppreference.com/w/cpp/language/pimpl) to hide the internal details of classes in implementation classes that are hidden from users once compiled. This was a particular choice we made but is not necessarily the only way of programming a Mochi component in C++. Feel free to develop your component with the paradigms and design patterns that suit you!

The implementation all the above classes is located in the *src* directory. The *src/dummy* directory provides a default implementation of a backend. We recommend that you implement a dummy backend for your service, as a way of testing application logic and RPCs without the burdon of complex external dependencies. For instance, a dummy backend may be a backend that simply acknowledges requests but does not process them, or provides mock results.

The project uses a number of dependencies:

- TCLAP for parsing program options in the examples;
- CppUnit for unit testing (in the tests directory);

- spdlog to provide logging.

The *examples* directory contains an example using the microservice: the server example will start one or more providers and print the server's address. The admin example can connect to a provider and have it create a resource (and print the resource id) or open, close, or destroy resources. The client example can be run next to interact with the resource.

The template also contains a *spack.yaml* file at its root that can be used to install its dependencies. You may add additional dependencies into this file as your microservice gets more complex.

As you modify this project to implement your own microservice, feel free to remove any dependencies you don't like (such as TCLAP, spdlog, CppUnit) and adapt it to your needs!

Setting up your project

Let's assume you want to create a microservice called "yellow", which manages a phone directory (association between names and phone numbers). The following shows how to setup your project:

```
git clone https://xgitlab.cels.anl.gov/sds/templates/thallium-microservice-template.
↪git
mv thallium-microservice-template yellow
cd yellow
rm -rf .git
python setup.py
$ Enter the name of your service: yellow
$ Enter the name of the resources (e.g., database): phonebook
```

The python script will edit and rename all the files, replacing *alpha* with *yellow* and *resource* with *phonebook* (with matching capitalization).

Building the project

The project's dependencies may be build using spack. You will need to have setup sds-repo as external namespace for spack, which can be done as follows.

```
# from outside of your project directory
git clone git@xgitlab.cels.anl.gov:sds/sds-repo.git
spack repo add sds-repo
```

The easiest way to setup the dependencies for this project is to create a spack environment using the *spack.yaml* file located at the root of the project, as follows.

```
# create an anonymous environment
cd thallium-microservice-template
spack env activate .
spack install
```

or as follows.

```
# create an environment named myenv
cd thallium-microservice-template
spack env create myenv spack.yaml
spack env activate myenv
spack install
```

Once the dependencies have been installed, you may build the project as follows.

```
mkdir build
cd build
cmake .. -DENABLE_TESTS=ON -DENABLE_EXAMPLES=ON -DENABLE_BEDROCK=ON
make
```

You can test the project using `make test` from the build directory.

5.9 Bedrock

Composing various Mochi components together into a daemon program running the service can be done manually, by writing a C, C++, or Python code that initializes various providers and resolves dependencies between providers. This method also requires to write custom bootstrapping mechanisms and custom ways of configuring the different components of the service. Alternatively, the composition and configuration of components can be delegated to Bedrock.

Bedrock is a bootstrapping and configuration system for Mochi components. It comes in the form of a program that can be run alone or with an MPI or a PMIx context. This program takes a JSON configuration file specifying the various components to instantiate and their dependencies. Bedrock also allows to retrieve the configuration of a service at any point during its run time.

This section of the documentation goes through the use of Bedrock, from deploying components, to writing Bedrock modules for your own components.

5.9.1 Starting with Bedrock

In this tutorial, we will install Bedrock and deploy a simple (empty) Bedrock-based service.

Installing Bedrock

Bedrock can be installed with Spack using the following command.

```
spack install mochi-bedrock
```

Starting a Bedrock process

Once installed a Bedrock process can be started as follows.

```
bedrock <protocol>
```

Where *<protocol>* is the protocol to use, for instance *na+sm*. This command starts an "empty" Bedrock process, in the sense that we haven't asked it to start any component apart from a Margo instance using the specified protocol. Hence the only thing we can do for now is query for its internal configuration, or shut it down.

This command can take additional parameters.

- `-c/--config <config.json>`: specifies the JSON configuration file.
- `-v/--verbose <level>`: logging level (*trace*, *debug*, *info*, *warning*, *error*, *critical*, or *off*).
- `--stdin`: pass the JSON configuration via stdin instead of `-c/--config`.

The next section will disect the content of a JSON configuration file.

5.9.2 Bedrock's JSON format

This section details how to configure a Bedrock daemon using a JSON file. The code bellow is an example of such a JSON configuration.

```
{
    "margo" : {
        "mercury" : {
        },
        "argobots" : {
        "abt_mem_max_num_stacks" : 8,
        "abt_thread_stacksize" : 2097152,
        "version" : "1.0.0",
        "pools" : [
            {
                "name" : "my_progress_pool",
                "kind" : "fifo_wait",
                "access" : "mpmc"
            },
            {
                "name" : "my_rpc_pool",
                "kind" : "fifo_wait",
                "access" : "mpmc"
            }
        ],
        "xstreams" : [
            {
                "name" : "my_progress_xstream",
                "cpubind" : 0,
                "affinity" : [ 0, 1 ],
                "scheduler" : {
                    "type" : "basic_wait",
                    "pools" : [ "my_progress_pool" ]
                }
            },
            {
                "name" : "my_rpc_xstream",
                "cpubind" : 2,
                "affinity" : [ 2, 3, 4, 5 ],
                "scheduler" : {
                    "type" : "basic_wait",
                    "pools" : [ "my_rpc_pool" ]
                }
            }
            ]
        },
        "progress_pool" : "my_progress_pool",
        "rpc_pool" : "my_rpc_pool"
    },
    "bedrock": {
        "pool": "my_rpc_pool",
        "provider_id": 0
    },
    "abt_io" : [
        {
            "name" : "my_abt_io",
            "pool" : "__primary__"
        }
```

(continues on next page)

(continued from previous page)

```
    ],
    "ssg" : [
        {
            "name" : "mygroup",
            "bootstrap" : "init",
            "group_file" : "mygroup.ssg"
        }
    ],
    "libraries" : {
        "module_a" : "examples/libexample-module-a.so",
        "module_b" : "examples/libexample-module-b.so"
    },
    "clients" : [
        {
            "name" : "ClientA",
            "type" : "module_a",
            "config" : {},
            "dependencies" : {}
        }
    ],
    "providers" : [
        {
            "name" : "ProviderA",
            "type" : "module_a",
            "provider_id" : 42,
            "pool" : "__primary__",
            "config" : {},
            "dependencies" : {}
        },
        {
            "name" : "ProviderB",
            "type" : "module_b",
            "provider_id" : 33,
            "pool" : "__primary__",
            "config" : {},
            "dependencies" : {
                "ssg_group" : "mygroup",
                "a_provider" : "ProviderA",
                "a_local" : [ "ProviderA@local" ],
                "a_client" : "module_a:client"
            }
        }
    ]
}
```

Margo section (optional)

The first section that such a JSON file may contain is a `margo` section. The format for this section is explained in the Margo tutorials and will not be covered here. Note that if not provided, default values will be used, hence the Margo instance created by Bedrock will still rely on an Argobots pool called `__primary__`.

Note: In the following sections, keep in mind that an Argobots configuration always has an implicite `__primary__` pool, even if it does not appear in the configuration. This pool can always be refered to by name.

Bedrock section (optional)

The `bedrock` section can provide Bedrock-specific configuration parameters, including the Argobots pool in which Bedrock RPCs should execute, and the provider id used by Bedrock. All the Bedrock daemons in a given service should use the same provider id. There is usually no reason to change this id, unless Bedrock is embedded inside an application and multiple Bedrock instances are running alongside each other.

ABT-IO section (optional)

The `abt_io` section describes ABT-IO instances to initialize. It is an array of objects in the form `{ "name" : "...", "pool" : "..." }`. The name given to an instance will be used later to resolve dependencies. The pool must refer to one of the Argobots pools defined in the `margo` section of the JSON file. The pool can be referred either by name or by index.

SSG section (optional)

The `ssg` section describes SSG groups to initialize or join. It is an array of objects in the following form.

```
{
    "name" : "<string>",
    "bootstrap" : "<string>",
    "group_file" : "<string>",
    "pool" : "<string|int>",
    "credential" : "<int>",
    "swim" : {
        "period_length_ms" : "<int>",
        "suspect_timeout_periods" : "<int>",
        "subgroup_member_count" : "<int>",
        "disabled" : "<bool>"
    }
}
```

- The “name” is used for other components to be able to refer to the group;
- The “bootstrap” method must be one of “init”, “join”, “mpi”, “pmix”;
- “group_file” should be the name of the file to create or use for this group;
- “pool” should refer to a pool defined in the `argobots` section, either by name or by index;
- “credential” is a number used when the underlying network requires a security token to be specified;
- The “swim” section provides the parameters for the SWIM protocol run by this group.

Apart from the name, bootstrap, and group file, the rest of the fields are optionals.

If the bootstrap method is “init”, the process will initialize a group with only itself as a member. If “mpi” is specified, the process will bootstrap a group from `MPI_COMM_WORLD`. If “pmix” is specified, the process will use PMIx to bootstrap the group. In these three cases, the group file specified will be created (or overwritten if it already exists), and the “swim” section will be used to setup the SWIM protocol. The last bootstrap method, “join”, lets a process join an existing group. The group file must already exist, and the “swim” parameters will be ignored and queried from existing members of the group.

Libraries, clients, and providers

The `libraries` section associates component (or “module”) names with shared libraries to load. These libraries tell Bedrock how to instantiate a provider, a client, and provider handles of a given component type. The next tutorial details how to write these libraries.

The `clients` section is an array of client objects of the following form.

```
{
    "name" : "<string>",
    "type" : "<string>",
    "config" : "<object>",
    "dependencies" : {
        "<key1>" : "<name1>",
        "<key2>" : "<name2>"
    }
}
```

The `name` will be the name by which the client can be referred to in other places of the configuration. The `type` must be one of the module names listed in the `libraries` section. `config` should be a JSON object formatted to comply with the component's specific JSON format. It will be passed as-is to the component's client creation function. You should refer to the component's documentation to know what is expected from this configuration field.

Finally the `dependencies` entry is an object associating *dependency names* to *references*. Bedrock will resolve these references and pass them to the client creation function when calling it.

Note: Bedrock can create clients as needed (e.g. when a provider depends on one or depends on a provider handle), so there is usually no need to fill up manually the `clients` section of the configuration. The only reasons for doing so would be (1) a client needs to be passed a specifid configuration, (2) a client needs to be provided with required dependencies, (3) you want to instantiate and use multiple clients of the same type.

The `providers` section is an array of provider objects of the following form.

```
{
    "name" : "<string>",
    "type" : "<string>",
    "provider_id" : "<int>",
    "pool" : "<string|int>",
    "config" : "<object>",
    "dependencies" : {
        "<key1>" : "<name1>",
        "<key2>" : "<name2>"
    }
}
```

The `name` will be the name by which the provider can be referred to in other places of the configuration. The `type` must be one of the module names listed in the `libraries` section. The `provider_id` must be an integer between 0 and 32767 (max uint16). Multiple providers of the same type need to have distinct provider ids. The `pool` must be a reference, either by name or index, to a pool defined in the `argobots` section of the configuration. `config` should be a JSON object formatted to comply with the component's specific JSON format. It will be passed as-is to the component's provider registration function. You should refer to the component's documentation to know what is expected from this configuration field.

Finally the `dependencies` entry is an object associating *dependency names* to *references*. Bedrock will resolve these references and pass them to the provider creation function when calling it.

Dependency resolution

The `dependencies` section in a provider or client lists dependency names associated with values. These values can be one of the following.

- The name of an SSG group (for SSG dependencies) will resolve into the correponding SSG group handle, defined in the `ssg` section of the configuration.
- The name of an ABT-IO instance (for ABT-IO dependencies) will resolve into the corresponding ABT-IO instance, defined in the `abt_io` section of the configuration.
- The name of a client defined in the same JSON file;
- For providers, the name of another provider defined in the same JSON file (such provider must have been defined before) will resolve to a handle to this provider;
- A string of the form `"<type>:client"` where <*type*> is a type of component will resolve into a handle for a client of the corresponding component type. If a client was already defined in the `clients` section for the requested type, it will be used, otherwise Bedrock will attempt to create one. If multiple clients of the same type have been defined, the first one will be used;
- A string of the form `"<name>@<location>"` or `"<type>:<id>@<location>"` will resolve into a provider handle pointing to a specific provider identified either by its <*name*> or by its <*type*> and provider <*id*> at the specified <*location*>. The location may be either `local`, to refer to the calling process, or a Mercury address (e.g. `na+sm://2317/0`) pointing to a remote Bedrock daemon, or an SSG address in the form `ssg://<group_name>/<rank>`, which will be resolved into the address of the specified rank in the specified SSG group.

The documentation for a given component should provide you with the list of dependencies that must be provided, as well as their types. Some of these dependencies may be listed as optional, some may be mandatory, in which case Bedrock will fail if the dependency isn't provided in the `dependencies` section of the provider. Some dependencies may be an array, some may be a single string.

Important: Clients are initialized before providers, hence they cannot depend on providers.

Working with multiple clients

Suppose we have a provider `ProviderA` of type `module_a`. Given the following provider handle specification: `ProviderA@ssg://mygroup/0`, Bedrock will by default use the first client of type `module_a` it finds to create the provider handle. If such a client does not exist, Bedrock will create a default one named `__module_a_client__`.

Let's now assume we have defined two clients `ClientA1` and `ClientA2` in the `clients` section of our JSON file, Bedrock will use `ClientA1` by default to create the provider handle. To sepcify that we want Bedrock to use `ClientA2`, we can write the provider handle specification as follows: `ClientA2->ProviderA@ssg://mygroup/0`.

Note: When querying Bedrock's configuration, you will find that the `clients` section is present regardless of whether you initially provided id, and has been completed with clients that Bedrock needed to create by itself. You will also note that all the provider handle specifications have been resolved into the following format: `client->type:id@address` where *client* is the name of the client used, *type* is the type of provider, *id* is the provider id, and *address* is the Mercury address.

5.9.3 Querying a Bedrock configuration

Once one or multiple Bedrock daemons have been deployed, it is possible to query their complete configuration, either via command line, or using a C++ program.

bedrock-query

`bedrock-query` is a program installed with Bedrock that can query the configuration of one or multiple daemons and print it on its standard output. It is called as follows.

```
bedrock-query <protocol> -a <address>
```

<protocol> is the protocol used by the Bedrock daemon (and consequently by the query program), e.g. `na+sm`. *<address>* is the Mercury address of the Bedrock daemon process.

The configuration is printed to the standard output in the form of a JSON object associating the address with its configuration.

Multiple addresses can be passed, e.g.:

```
bedrock-query <protocol> -a <address1> -a <address2> -a <address3>
```

In this case, the resulting JSON object will have three entries, one for each address. If all the daemons are gathered in an SSG group represented by a group file of a given *<filename>*, the following command will query all the members of the group.

```
bedrock-query <protocol> -s <filename>
```

The `-i/--provider-id` flag may be used to specify a provider id other than 0. for instance:

```
bedrock-query <protocol> -a <address> -i 42
```

Using a C++ program

The easiest way to learn how to do all of the above in C++ is to look at how `bedrock-query` is implemented, as well as the *Client.hpp* and *ServiceHandle.hpp* files in *include/bedrock*.

5.9.4 Writing a Bedrock module in C

If you have programmed your own Mochi component, writing a module to make it usable with Bedrock is really not difficult. Such a module consists of a single dynamic library (.so) that can be implemented as show in the example bellow.

```
#include <bedrock/module.h>
#include <abt-io.h>
#include <string.h>

static struct bedrock_dependency ModuleA_provider_dependencies[] = {
    { "io", "abt_io", BEDROCK_REQUIRED },
    { "sdskv_ph", "sdskv", BEDROCK_ARRAY | BEDROCK_REQUIRED },
    BEDROCK_NO_MORE_DEPENDENCIES
};

static struct bedrock_dependency ModuleA_client_dependencies[] = {
    BEDROCK_NO_MORE_DEPENDENCIES
};

static int ModuleA_register_provider(
        bedrock_args_t args,
        bedrock_module_provider_t* provider)
```

(continues on next page)

(continued from previous page)

```
{
    margo_instance_id mid  = bedrock_args_get_margo_instance(args);
    uint16_t provider_id   = bedrock_args_get_provider_id(args);
    ABT_pool pool          = bedrock_args_get_pool(args);
    const char* config     = bedrock_args_get_config(args);
    const char* name       = bedrock_args_get_name(args);

    abt_io_instance_id* io = bedrock_args_get_dependency(args, "io", 0);
    size_t num_databases   = bedrock_args_get_num_dependencies(args, "sdskv_ph");
    int i;
    for(i=0; i < num_databases; i++) {
        void* sdskv_ph = bedrock_args_get_dependency(args, "sdskv_ph", i);
        /* ... */
    }

    *provider = strdup("ModuleA:provider"); // just to put something in *provider
    printf("Registered a provider from module A\n");
    printf(" -> mid         = %p\n", (void*)mid);
    printf(" -> provider id = %d\n", provider_id);
    printf(" -> pool        = %p\n", (void*)pool);
    printf(" -> config      = %s\n", config);
    printf(" -> name        = %s\n", name);
    return BEDROCK_SUCCESS;
}

static int ModuleA_deregister_provider(
        bedrock_module_provider_t provider)
{
    free(provider);
    printf("Deregistered a provider from module A\n");
    return BEDROCK_SUCCESS;
}

static char* ModuleA_get_provider_config(
        bedrock_module_provider_t provider) {
    (void)provider;
    return strdup("{}");
}

static int ModuleA_init_client(
        bedrock_args_t args,
        bedrock_module_client_t* client)
{
    *client = strdup("ModuleA:client");
    printf("Registered a client from module A\n");
    return BEDROCK_SUCCESS;
}

static int ModuleA_finalize_client(
        bedrock_module_client_t client)
{
    free(client);
    printf("Finalized a client from module A\n");
    return BEDROCK_SUCCESS;
}

static char* ModuleA_get_client_config(
```

(continues on next page)

(continued from previous page)

```
        bedrock_module_client_t client) {
    (void)client;
    return strdup("{}");
}

static int ModuleA_create_provider_handle(
        bedrock_module_client_t client,
        hg_addr_t address,
        uint16_t provider_id,
        bedrock_module_provider_handle_t* ph)
{
    (void)client;
    (void)address;
    (void)provider_id;
    *ph = strdup("ModuleA:provider_handle");
    printf("Created provider handle from module A\n");
    return BEDROCK_SUCCESS;
}

static int ModuleA_destroy_provider_handle(
        bedrock_module_provider_handle_t ph)
{
    free(ph);
    printf("Destroyed provider handle from module A\n");
    return BEDROCK_SUCCESS;
}

static struct bedrock_module ModuleA = {
    .register_provider       = ModuleA_register_provider,
    .deregister_provider     = ModuleA_deregister_provider,
    .get_provider_config     = ModuleA_get_provider_config,
    .init_client             = ModuleA_init_client,
    .finalize_client         = ModuleA_finalize_client,
    .get_client_config       = ModuleA_get_client_config,
    .create_provider_handle  = ModuleA_create_provider_handle,
    .destroy_provider_handle = ModuleA_destroy_provider_handle,
    .provider_dependencies   = ModuleA_provider_dependencies,
    .client_dependencies     = ModuleA_client_dependencies
};

BEDROCK_REGISTER_MODULE(module_a, ModuleA)
```

Module dependencies

The first thing to declare in this module is the dependencies of providers and client. This is the list of dependencies that your component's providers and clients are expecting. It is expressed as two arrays of `bedrock_dependency` structures terminated by a `BEDROCK_NO_MORE_DEPENDENCIES` entry. Each of the elements in this array has a dependency name, a type, and a flag. The name is what will identify the dependency in the JSON configuration file. For instance, our provider here has two dependencies, *io* and *sdskv_ph*, hence the `dependencies` field of a *module_a* provider in a JSON configuration should have a `io` entry and an `sdskv_ph` entry. Our client has no dependency.

The *type* of dependency tells Bedrock what to look for. The `abt_io` type tells Bedrock to find an ABT-IO instance. The `sdskv` type tells Bedrock to find an object (provider, client, or provider handle) from the SDSKV module.

Finally the *flag* may be `BEDROCK_OPTIONAL` for optional dependencies, `BEDROCK_REQUIRED` for a required dependencies, `BEDROCK_ARRAY` for an array of dependencies (including an empty array), or `BEDROCK_REQUIRED`

| `BEDROCK_ARRAY` for an array of at least one element.

Given the above dependency declarations for our module, a valid provider instantiation in the JSON document might look like the following.

```
{
    "libraries" : {
        "module_a" : "path/to/libbedrock-module-a.so"
    },
    "providers" : [
        {
            "name" : "ProviderA",
            "type" : "module_a",
            "provider_id" : 42,
            "pool" : "my_pool",
            "config" : {},
            "dependencies" : {
                "io" : "my_abt_io",
                "sdskv_ph" : [ "my_sdskv@ssg://my_group/0", "other_sdskv@local" ]
            }
        }
    ]
}
```

The `libraries` section must associate the *module_a* type with the library we just built. Assuming an ABT-IO instance named "my_abt_io" was declared in the `abt_io` section of the document, the "io" dependency will be resolved to that instance. The "sdskv_ph" dependency will resolve to an array of two provider handles pointing to SDSKV providers.

Callback functions

The rest of the module consists of callback functions to register and deregister a provider, get a provider's configuration, initialize and finalize a client, and create and destroy a provider handle for your module.

The provider registration and client initialization callbacks are being passed a `bedrock_args_t` handle from which we can extrat various configuration parameters, including the Margo instance, the provider id (in provider registration), the Argobots pool (same), the configuration (`config` field from the JSON configuration of the provider or the client), the name of the provider/client, as well as its dependencies using `bedrock_args_get_num_dependencies` and `bedrock_args_get_dependency` (dependencies that are not an array are treated like an array of size 1. `bedrock_args_get_num_dependencies` can also be used to check for the presence of an optional dependency).

Note that Bedrock checks ahead of time that all required dependencies are present, hence the provider registration function does not have to check this.

Note: The function that returns a provider's configuration must return a heap-allocated string, that Bedrock will later `free` by itself.

Important: At the moment, Bedrock initializes clients before providers, hence clients cannot depend on providers. Clients can only depend on provider handles to SSG groups, ABT-IO instances, other clients defined earlier, and provider handles.

Registering the module

The last things to add to our library is to fill out a `bedrock_module` structure with our callbacks and dependency array, and to call `BEDROCK_REGISTER_MODULE(module_a, ModuleA)`. The first argument of this macro is the name under which we want our module to be known. The second is the name of the `bedrock_module` structure we have built.

Note: All the above functions as well as the dependency array and the module structure instance can safely be declared `static`. The only symbole required by the Bedrock to work with the library will be generated by the `BEDROCK_REGISTER_MODULE` macro.

5.9.5 Write a Bedrock module in C++

If your component is written in C++, it might be easier to write the Bedrock module library in C++. We still recommend reading the previous section to understand how a C module works.

The following code sample showcases a C++ module.

```
#include <bedrock/AbstractServiceFactory.hpp>
#include <iostream>

class ServiceBFactory : public bedrock::AbstractServiceFactory {

    public:

    ServiceBFactory() {
        m_provider_dependencies.push_back({ "ssg_group", "ssg", BEDROCK_REQUIRED });
        m_provider_dependencies.push_back({ "a_provider", "module_a", BEDROCK_
↪REQUIRED });
        m_provider_dependencies.push_back({ "a_local", "module_a", BEDROCK_REQUIRED |␣
↪BEDROCK_ARRAY});
        m_provider_dependencies.push_back({ "a_client", "module_a", BEDROCK_REQUIRED }
↪);
    }

    void* registerProvider(const bedrock::FactoryArgs& args) override {
        std::cout << "Registering a provider from module B" << std::endl;
        std::cout << " -> mid         = " << (void*)args.mid << std::endl;
        std::cout << " -> provider_id = " << args.provider_id << std::endl;
        std::cout << " -> pool        = " << (void*)args.pool << std::endl;
        std::cout << " -> config      = " << args.config << std::endl;
        std::cout << " -> name        = " << args.name << std::endl;
        for(auto& dep : args.dependencies) {
            std::cout << "dependency " << dep.first << " -> [ ";
            for(auto& s : dep.second) {
                std::cout << s.spec << " (" << s.handle << "), ";
            }
            std::cout << " ]" << std::endl;
        }
        return (void*)0x1; // new ProviderB(...)
    }

    void deregisterProvider(void* provider) override {
        (void)provider;
```

(continues on next page)

(continued from previous page)

```
        std::cout << "Deregistring provider from module B" << std::endl;
        // auto p = static_cast<ProviderB*>(provider);
        // delete p;
    }

    std::string getProviderConfig(void* provider) override {
        (void)provider;
        // auto p = static_cast<ProviderB*>(provider);
        // return p->getConfig();
        return "{}";
    }

    std::string getClientConfig(void* client) override {
        (void)client;
        // auto c = static_cast<ClientB*>(client);
        // return c->getConfig();
        return "{}";
    }

    void* initClient(const bedrock::FactoryArgs& args) override {
        (void)args;
        std::cout << "Initializing client from module B" << std::endl;
        return (void*)0x2;
    }

    void finalizeClient(void* client) override {
        (void)client;
        std::cout << "Finalizing client from module B" << std::endl;
    }

    void* createProviderHandle(void* client, hg_addr_t address, uint16_t provider_id)␣
↪override {
        (void)client;
        (void)address;
        (void)provider_id;
        std::cout << "Creating a provider handle from module B" << std::endl;
        return (void*)0x3;
    }

    void destroyProviderHandle(void* providerHandle) override {
        (void)providerHandle;
        std::cout << "Destroying provider handle from module B" << std::endl;
    }

    const std::vector<bedrock::Dependency>& getProviderDependencies() override {
        return m_provider_dependencies;
    }

    const std::vector<bedrock::Dependency>& getClientDependencies() override {
        return m_client_dependencies;
    }

    private:

    std::vector<bedrock::Dependency> m_provider_dependencies;
    std::vector<bedrock::Dependency> m_client_dependencies;
};
```

(continues on next page)

(continued from previous page)

```
BEDROCK_REGISTER_MODULE_FACTORY(module_b, ServiceBFactory)
```

In C++, a Bedrock module is created by writing a class that inherits from `bedrock::AbstractServiceFactory`. The class must satisfy the interface of its abstract parent class. What in the C module were functions now become member functions of our class, and some of the constructs differ slightly. For example, the `bedrock::Dependency` class uses `std::string` for the name and type of the dependency, and the arguments provided to the provider registration method and client initialization method is not opaque.

The `BEDROCK_REGISTER_MODULE_FACTORY` macro should be called in place of the C module's `BEDROCK_REGISTER_MODULE`.

5.10 General tutorials

This section contains some tutorials explaining common concepts used in the Mochi libraries, and pitfalls one may encounter while developing with the Mochi libraries.

5.10.1 Using Mochi in conjunction with MPI

There is no problem using Mochi libraries alongside MPI. However, some precautions must be taken. If you are using MPI in conjunction with either Thallium or Margo, and witness your code hanging, this page might be for you. This is a common mistake was have seen countless time. In the following, we will take Margo as an example, although the same applies to Thallium.

Understanding *where* the progress loop is running

Margo internally uses Argobots to manage user-level threads (ULT). ULTs are scheduled on execution streams (ES). Contrary to a preemptive multithreading system like pthreads, ULTs running on the same ES must explicitely yield to one another in order for each ULT to get a chance to run. Yielding happens either explicitely or when calling an Argobots function that will block the ULT (e.g., trying to lock a mutex). If a ULT blocks on a function that is not Argobots-aware, it will not yield to other ULT in its ES, which will block the entire ES.

In Margo, the Mercury progress loop is put in its own ULT. When initializing Margo using `margo_init`, the third argument indicates *where* this ULT will be placed: a value of 0 indicates that it will be running in the context of the ES that called `margo_init`. A value of 1 will make Margo create a new ES dedicated to running the progress loop.

Understanding *when* the progress loop is running

In a client program, if the progress loop doesn't have its own ES, it will execute whenever `margo_forward`, `margo_provider_forward`, or `margo_wait` are called. These are the functions that need to wait for the completion of a Mercury operation before returning. Hence, they have to run the progress loop until such completion happens. If the client is also using MPI, this generally does not pose any problem because the client will either block on an MPI call (and not need the progress loop to be running) or block on a Margo call (and not need MPI progress to happen).

If the progress loop has been placed in a dedicated ES, it will run continuously in the background and, once again, the client will not have to care about conflicting MPI and Margo calls.

The situation becomes more complicated for servers. If Margo was initialized without a dedicated ES for the progress loop, then the progress loop will run either when the main ES calls `margo_forward`, `margo_provider_forward`, or `margo_wait` (just like for clients), or when the main ES blocks on `margo_wait_for_finalize`. *Everytime the ES does anything else than these calls, the progress loop won't be running and the server will not be able to respond to incoming RPCs*. When using MPI in conjunction with Margo in this scenario, this means that whenever the main ES blocks on an MPI call, it will not make progress on Mercury operations. A common mistake we have seen users make is initializing servers as an MPI program, initializing Margo without a dedicated progress ES, then block on a call to `MPI_Barrier` (or another MPI operation) instead of blocking on `margo_wait_for_finalize`. The solution in such a scenario is either to use a dedicated ES for the mercury progress loop (set the third argument of `margo_init` to 1), or make sure not to use MPI calls when the progress loop needs to be running, or to put all MPI calls in an ES initialized separately.

If a server initializes Margo with a dedicated progress loop, there are still cases where a conflict between MPI and Margo could occure. These cases have to do with where the RPC handlers are executed.

Understanding *where* the RPC handlers are executed

In a server, when the Mercury progress loop receives an RPC request, it will create a new ULT to run the corresponding RPC handler. The fourth argument of `margo_init` indicates *where* these ULTs are executed. A value of -1 indicates that the RPC handlers will execute in the same ES as the one running the progress loop. A value of 0 indicates that they will execute in the ES that called `margo_init` (the main ES). Finally a positive value *N* will make Margo create *N* new ES where these RPC handlers will execute.

Depending on this parameter, conflicts with MPI operations may happen. If RPC handlers execute in the same ES as the progress loop, time spent in an RPC handler is time not spent in the progress loop. If an RPC handler blocks on an MPI call, it will block its entire ES and therefore prevent the progress loop from running.

If the RPC handlers execute in the same ES as `margo_init`, an MPI call that blocks the main ES will prevent RPC handlers from running. A frequent mistake we see users make is initializing Margo with a dedicated progress loop (third argument of `margo_init` set to 1), and RPC handlers running in the main ES (fourth argument set to 0), but block the main ES on an `MPI_Barrier`. The solution in this case is to either use -1 or a positive number as the fourth argument to `margo_init`.

Finally if the RPC handlers execute in some *N* dedicated ES, it is still worth keeping in mind that any blocking MPI call will prevent any ULT from running in the blocked ES. Hence, if *N* RPC handlers are blocked on MPI calls, no other RPC handlers will be executed. This scenario is however very unlikely since users generally don't issue MPI calls in parallel from several threads.

Conclusion

When using MPI in conjunction with Margo, users need to keep in mind that any blocking MPI call will prevent the calling ES from yielding control to other ULT, which in turn may prevent either the Mercury progress loop or some RPC handlers from running. A good practice consists of initializing Margo with a dedicated progress loop and run RPC handlers either in the progress loop (fourth argument of -1) or in dedicated ES (fourth argument greater than 0), and make MPI calls only from the main ES.

When initializing Margo with `margo_init_pool`, or when initializing providers and passing dedicated Argobots pools for their RPC handlers, users need to keep track of where the ULT placed in those pools may execute, to know whether they may conflict with MPI operations.

5.10.2 Using Mochi in conjunction with other threading libraries

Users should be especially careful when using Margo libraries in conjunction with other threading libraries. As explained in the previous tutorial about MPI, Margo uses Argobots internally for threading. Argobots provides user-level threads (ULT), which have to explicitely yield to one another in order to context-switch. Blocking on an Argobots mutex, lock, eventual, or future will also produce a context-switch to another ULT, allowing resource sharing.

Users relying on another threading library in conjunction with the Mochi library must keep in mind that blocking calls in these libraries will not make Argobots yield to other ULTs. These calls will block the entire execution stream from which they are made, preventing further progress altogether.

It is recommended, when using another threading library with Argobots, to isolate calls for this library in a dedicated ES, or to use dedicated ES for all Mochi-related work. For example, using a dedicated progress ES along with dedicated RPC handler ES when initializing Margo, and making sure RPC handlers don't make calls to the second threading library, is a good way to prevent any conflict from happening.

5.10.3 Understanding the RPC and ULT model

When developing a Mochi service either with Margo or Thallium, it can be useful to have in mind how RPCs translate into user-level threads (ULT) when reaching the server. The figure bellow summarizes what happens when a client sends an RPC to a server, and the RPC handler on the server side includes some RDMA operations.

In this figure, we show only one execution stream for the client, assuming it has initialized Margo (or Thallium) without a Mercury progress thread. The case of using a Mercury progress thread on a client is similar, as the progress thread simply takes care of network activities on behalf of the caller thread.

This figure shows a client using `margo_iforward`, which sends an RPC to a server in a non-blocking manner. The case of `margo_forward` can be viewed as the same scenario but with `margo_wait` invoked immediately after `margo_iforward`. In Thallium, the equivalent code would use the *async* member function of a *callable_remote_procedure* object.

Explanations

`margo_forward` and `margo_iforward` start by calling the serialization function (provided by the user when registering RPCs using `MARGO_REGISTER`) to serialize the RPC argument into an input buffer. Mercury then sends a request, including this buffer, to the server.

In the server, the Mercury progress loop (which may execute on a dedicated execution stream) eventually sees the request and invokes the corresponding callback (in yellow). This callback has been automatically generated by `DEFINE_MARGO_RPC_HANDLER` in the user's code. This callback (1) looks up the Argobots pool in which the RPC is supposed to execute, and (2) creates a ULT in that pool. This ULT will run the user's RPC handler.

RPC handler ULTs are posted in a pool that may use different execution streams (ES) than the pool used by the Mercury progress loop. For instance when calling `margo_init(..., 1, 8)`, 8 ES are created along with a shared pool in which RPC handler ULTs will be posed. When one of the ES is free, it pulls an ULT from the pool and executes it.

Generally, the RPC handler will start by deserializing the RPC's argument by calling *margo_get_input*. This invokes the user-provided serialization function to deserialize the content of the Mercury buffer into the user's input data structure.

When issuing a bulk transfer (RDMA) using `margo_bulk_transfer`, the ULT asks the Mercury progress loop to execute the transfer. Meanwhile, this ULT yields, so that the ES on which it runs can execute other ULTs (e.g. other RPC requests).

The Mercury progress loop eventually executes the RDMA operation and notifies the calling ULT. The calling ULT is marked as ready and will eventually resume.

When the RPC handler calls `margo_respond` to send a response to the client, it first calls the user-provided serialization function to encode the response into Mercury's buffer, then yields, waiting for the Mercury progress loop to send the response and allowing the ES to potentially execute other RPCs in the meantime.

Once the response is sent by Mercury, the Mercury progress loop notifies the RPC handler ULT, which eventually resumes and complete.

Finally, `margo_wait` completes in the client. The client can then call `margo_get_output` on the RPC handler to deserialize the RPC's output using the user-provided deserialization function.

Note: This model remains valid regardless of whether the Mercury progress loop runs on a separate ULT or not. Indeed you may notice that, from the point of view of a single RPC operation, the two ES shown here on the server could be merged into one. The advantage of dedicating an ES to Mercury progress is that multiple concurrent RPC handlers may rely on it with minimum interference. Should the Mercury progress loop run in the same ES as the RPC handlers, calling `margo_respond` in a handler could yield to another (potentially long-running) RPC handler instead of the progress loop, thus delaying the completion of the first RPC handler.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search