Developing Custom HPC Data Services using Mochi

Matthieu Dorier[1], Phil Carns[1], Marc-André Vef[2]
ISC 2023, Hamburg – May 21st

[1] Argonne National Laboratory
[2] Johannes Gutenberg University Mainz

A note on the hands-on

This tutorial contains 3 hands-on session of 30min each.

If you haven't already, follow the initial setup here to setup your docker image:

bit.ly/3leh0yz

Introduction

What is Mochi?

Phil Carns
🕑 10 min

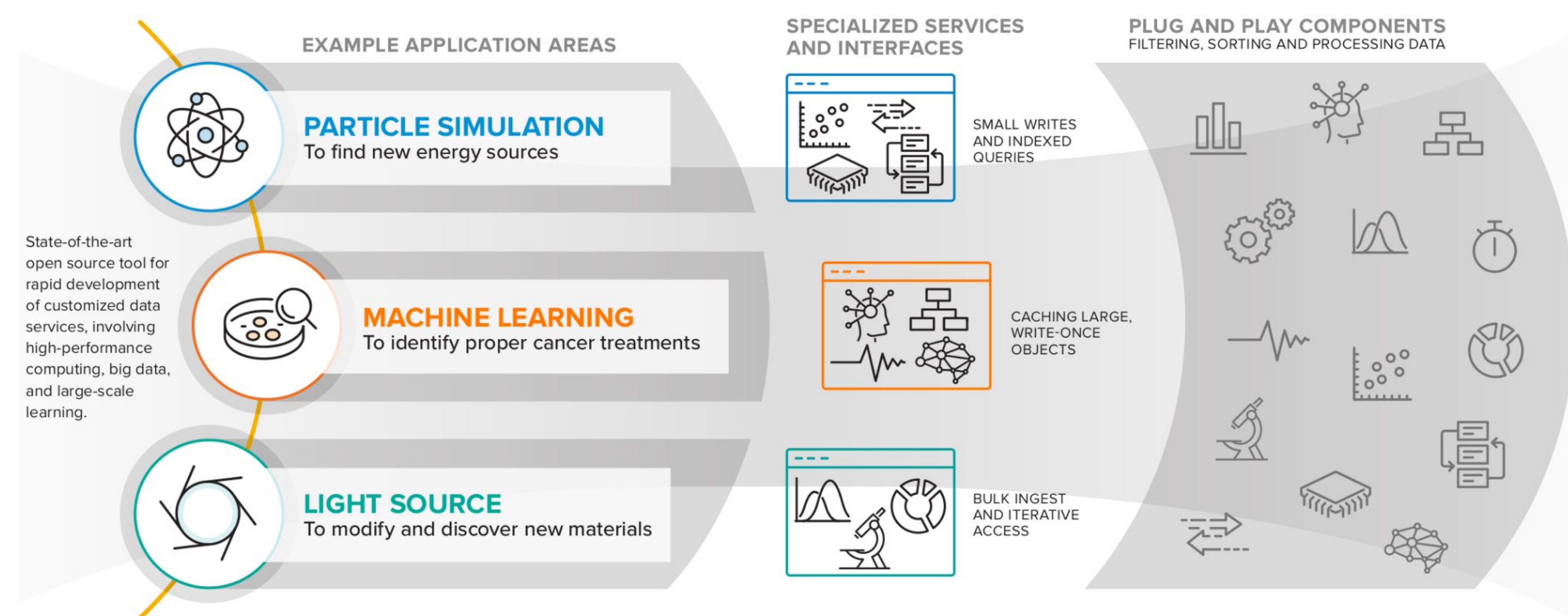
Mochi motivation

- HPC systems are typically deployed with a "one size fits all" data service (e.g., a parallel file system) for all applications.
- Libraries may be layered atop it, but ultimately all storage access uses the same semantics, policies, and interface.

Mochi seeks to transform this data service monoculture into **an ecosystem of specialized services that are tailored to suit specific use cases and problem domains**.

- **The objective of the Mochi project is to design methodologies and tools for the rapid development of distributed HPC data services.**
- Mochi relies heavily on composition: common capabilities such as communication, data storage, concurrency management, and group membership are provided under Mochi along with building blocks such as bulk data and key-value stores.
- These building blocks can be combined as needed to suit the task at hand.

The Mochi concept



Example Mochi components

	Component	Summary
Core	Argobots	Argobots provides user-level thread capabilities for managing concurrency.
	Mercury	Mercury is a library implementing remote procedure calls (RPCs).
	Margo	Margo is a C library using Argobots to simplify building RPC-based services.
	Thallium	Thallium allows development of Mochi services using modern C++.
	SSG	SSG provides tools for managing groups of providers in Mochi.
Utilities	ABT-IO	ABT-IO enables POSIX file access with the Mochi framework.
	Bedrock	Bedrock is a bootstrapping and configuration system for Mochi components.
	ch_placement	ch-placement is a library implementing multiple hashing algorithms.
	Shuffle	Shuffle provides a scalable all-to-all data shuffling service.
Microservices	BAKE	Bake enables remote storage and retrieval of blobs of data.
	POESIE	Poesie embeds language interpreters in Mochi services.
	REMI	REMI is a microservice that handles migrating sets of files between nodes.
	Sonata	Sonata is a Mochi service for JSON document storage based on UnQLite.
	Yokan	Yokan enables RPC-based access to multiple key-value backends.

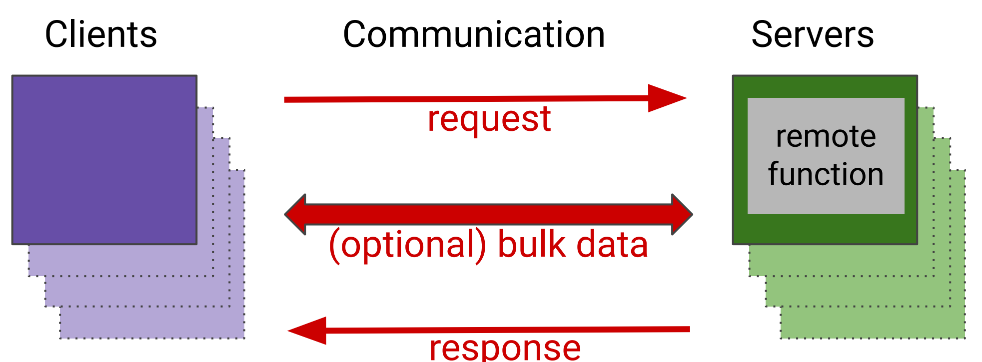
- Our hands-on exercises will focus on Margo (C), Thallium (C++), and Bedrock (composition).
- Mochi components are carefully designed for interoperability: reuse as much as you can!

Example services built with Mochi

Service	Institution(s)	Summary
Chimbuko	Brookhaven	Workflow-level scalable performance trace analysis tool
DAOS	Intel	Object store that provides high bandwidth, low latency, and high I/O operations per second (IOPS) storage containers to HPC applications
DataSpaces	Univ. of Utah	Programming system and data management framework for coupled workflows
GekkoFS	JGU Mainz	Temporary distributed file system for HPC applications
Hermes	IIT, THG, UIUC	User-space platform for distributing data structures
HXHIM	Los Alamos	Hexadimensional hashing indexing middleware
Proactive Data Containers	Berkeley	Object-centric data management system to take advantage of deep memory and storage hierarchy
Seer	Los Alamos	Lightweight in situ wrapper library adding in situ capabilities to simulations
Unify	LLNL and ORNL	Suite of specialized, flexible file systems that can be included in a user's job
	Kitware	Platform for ubiquitous access to visualization results during runtime
CHFS	Tsukuba	ad hoc file system for persistent memory based on consistent hashing

- Conventional data services, performance tuning, in situ analytics, and more.
- … anything that requires high-performance data exchange, decoupled from the application
- Marc Vef will walk us through an example full-featured Mochi service later on today.

What is an RPC-based data service?

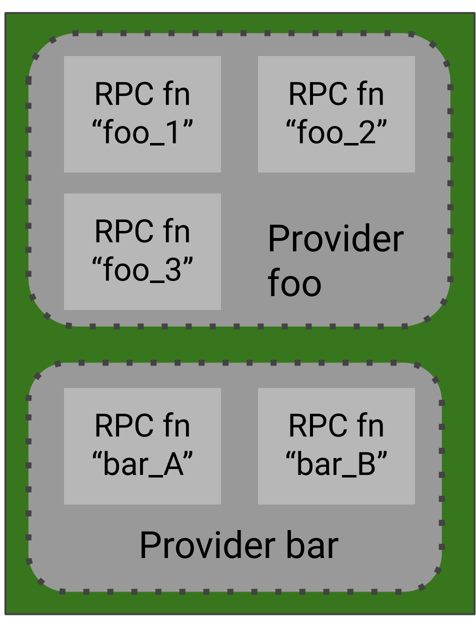


- RPC = “remote procedure call”
- Clients ask servers to execute remote functions on their behalf.
- Function inputs and outputs are encoded into request and response messages.

RPC systems have been around in various forms for decades. What’s unique about Mochi?

- Designed for high concurrency
- Explicit bulk data transfers (e.g., a fast path for I/O operations)
- Support for HPC hardware and protocols
  - But independent of MPI!
- Can operate in user space without escalated privileges

High-level Mochi concepts and terminology



RPCs and providers

- RPCs are grouped together into “**providers**” that collectively implement a service or broker access to a resource.
- There can be multiple providers per server
  - Even multiple instances of the same type of provider
- Providers can talk to each other.
  - One provider may delegate some functionality to another provider on the same server or elsewhere.

“**Everything is an RPC**”: Regardless of location, even among providers, use RPCs to interact with a provider.

High-level Mochi concepts and terminology

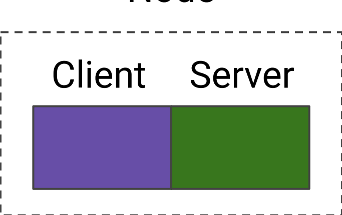
Why are RPCs the primary interface to providers?

This is crucial to composability.

- API conventions and addressing do not change based on deployment/composition (transparent transport selection is handled by Mochi components).
- RPC framework becomes the common substrate for monitoring and profiling.
- Let Mochi optimize the RPC communication path for you.

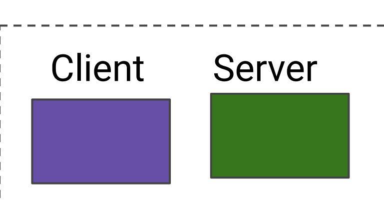
Node

RPC transport: Function call & memory copy



Node

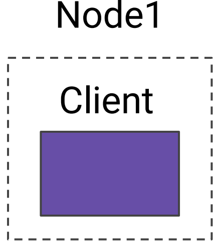
RPC transport: Local pipe & fast shared memory



Node1

Node2

RPC transport: HPC network fabric & RDMA



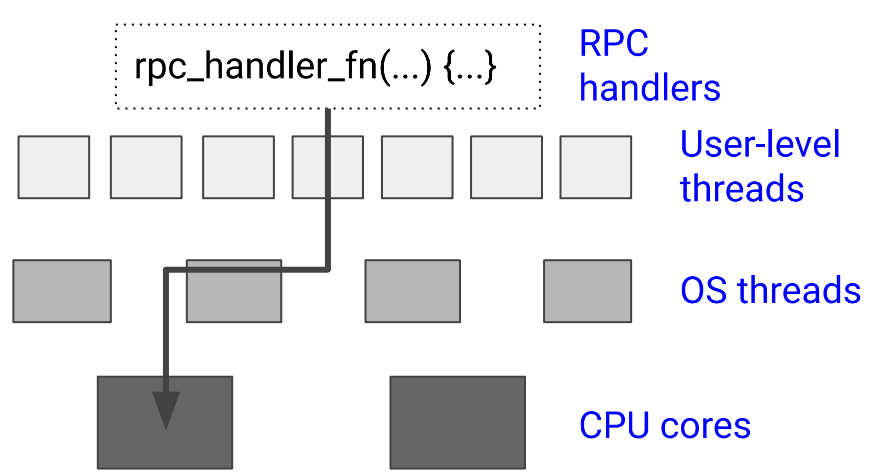
Server

High-level Mochi concepts and terminology

Decoupling software concurrency from hardware concurrency

- When you register a new RPC, you will define an RPC handler function to service it.
- Mochi will automatically execute handlers on user-level threads, which map to operating system threads, which map to CPU cores.

How do you manage this mapping in your service?
It's easy: don't! Use as much concurrency as you need. In Mochi, this resource mapping challenge is a configuration problem, not a software architecture problem.



Mochi uses the Argobots threading package. You can call Argobots functions directly (in C) or with Thallium wrappers (C++) if you have a need for explicit concurrency control: there are equivalents of all major pthread functions, plus more.

Mochi’s core libraries

Service development as easy as it can be

Margo

Programming model and API to develop a data service in C

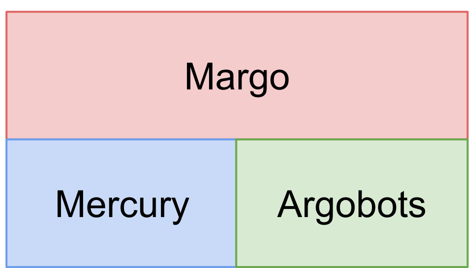
Matthieu Dorier

10 min

Margo is a library combining Mercury and Argobots

Mercury

- Provides RPC and RDMA
- Wide range of transport backends (tcp, verbs, gni,...)
- Callback-driven
- User has to implement their progress loop (call HG_Progress/HG_Trigger periodically)



Argobots

- Provides user-level threading runtime
- Very flexible in how and where user-level threads (ULTs) execute

Margo is a library built on top of Mercury and Argobots, hiding a Mercury progress loop in a ULT and converting RPC handlers into UTLs.

Using Margo allows programs to make other progress while waiting for network operations to complete.

The margo instance

```
#include <margo.h>

margo_instance_id mid = margo_init("tcp", MARGO_SERVER_MODE, true, 4);

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);

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

margo_wait_for_finalize(mid);
```

or `MARGO_CLIENT_MODE`

Whether to create a dedicated progress thread

Number of RPC threads. Passing 0 will make RPCs execute in the main thread. Passing -1 will make them execute in the Mercury progress thread.

Will block until `margo_finalize` is called (by another thread). Client should call `margo_finalize`.

Registering RPC handlers

Server

All RPC handlers have this signature

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

```
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!");
    margo_respond(h, NULL);
    margo_destroy(h);
}
DEFINE_MARGO_RPC_HANDLER(hello_world)
```

This macro generates a `hello_world_ult` function that will wrap the RPC in a ULT

On servers, provide the function pointer

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

Client

On clients, pass NULL

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

Calling the RPC

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

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

hg_handle_t handle;
ret = margo_create(mid, svr_addr, hello_rpc_id, &handle);
ret = margo_forward(handle, NULL);
ret = margo_destroy(handle);

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

Lookup the server's address from a string

Create a handle

Send the handle to the server along with RPC arguments (here NULL – we will talk about argument serialization later)

Let’s use arguments and return values

```
#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)))
```

In most cases, Mercury macros can be used to generate both the structure definition and the code that serializes it. If you need more complicated serialization, see https://mochi.readthedocs.io/en/latest/margo/08_proc.html

```
hg_id_t rpc_id = MARGO_REGISTER(mid, "sum", sum_in_t, sum_out_t, sum);
```

Now you can provide input and output types to the `MARGO_REGISTER` macro!

Client

```
sum_in_t in = { 42, 33 };
margo_forward(handle, &in);
sum_out_t out;
margo_get_output(handle, &out);
margo_free_output(handle, &out);
```

Server

```
sum_in_t in;
margo_get_input(handle, &in);
sum_out_t out = {.ret = in.x + in.y };
margo_respond(handle, &out);
margo_free_input(handle, &in);
```

The client needs to free the output, the server needs to free the input

Using RDMA: the `hg_bulk_t` handle

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

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

hg_bulk_t handle can be serialized. This does not serialize the data, but the information to access the process' memory remotely.

```
sum_in_t args;

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

hg_bulk_t bulk;
margo_bulk_create(mid, 1, seg_ptrs, seg_sizes, HG_BULK_READ_ONLY, &bulk);

args.n = 10;
args.bulk = bulk;
…
margo_bulk_free(bulk);
```

An hg_bulk_t handle can expose multiple non-contiguous segments

Tell margo what the remote process will do with the memory, using `HG_BULK_READ_ONLY`, `WRITE_ONLY`, or `READWRITE`.

Tip: most transports can reasonably handle a few segments.

If you have a lot of segments (more than 4), then it will probably be faster to pack them yourself.

Using RDMA: the `hg_bulk_t` handle

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

int32_t* values = calloc(in.n, sizeof(*values));
hg_size_t buf_size = in.n * sizeof(*values);

hg_bulk_t local_bulk;
ret = margo_bulk_create(mid, 1, (void**)&values, &buf_size,
HG_BULK_WRITE_ONLY, &local_bulk);
ret = margo_bulk_transfer(mid, HG_BULK_PULL, client_addr, in.bulk, 0,
                          local_bulk, 0, buf_size);
...
ret = margo_bulk_free(local_bulk);
```

Inside an RPC handler, you can get the address of the sender using `margo_get_info` on the handle.

Let's allocate a local buffer into which to transfer the data from the client

Create a local bulk handle for the local buffer

Tell margo to PULL from the remote bulk handle at offset 0 to the local bulk handle at offset 0

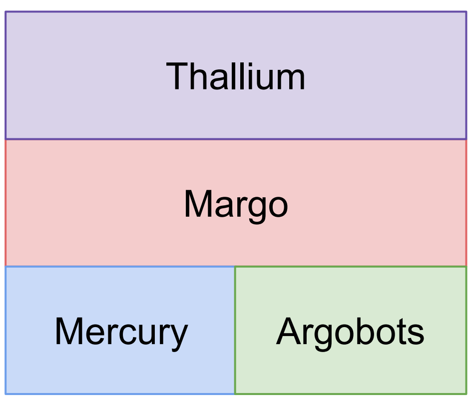
Note: some transports don't support non-contiguous memory (i.e. exposing more than one segment in `margo_bulk_create`). Exposing multiple segments will work, but transfer may be slower than one big contiguous segment.

Thallium

C++ for convenience and faster development

Matthieu Dorier

Thallium: modern C++ on top of Margo



Thallium

- C++14, object-oriented design
- Template metaprogramming allows turning any function and any lambda into an RPC handler
- Serialization done with the Cereal library instead of Mercury's preprocessor macros

The thallium engine

```
#include <thallium.hpp>

namespace tl = thallium;

// equivalent of margo_init
tl::engine engine{"tcp", THALLIUM_SERVER_MODE, true, 8};
std::cout << "Engine address is: " << engine.self() << std::endl;
...
// equivalent of margo_finalize
engine.finalize();
// equivalent of margo_wait_for_finalize
engine.wait_for_finalize();
```

or `THALLIUM_CLIENT_MODE`

Whether to create a dedicated progress thread

Number of RPC threads

- If initialized as server, the engine will call `wait_for_finalize` in its destructor
- If initialized as client, the engine will call `finalize` in its destructor
- The engine class is copy-constructible, multiple copies will refer to the same instance, only the last one to be destroyed will call `(wait_for_)finalize`

Registering RPC handlers

```
#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("ofi+tcp", THALLIUM_SERVER_MODE);
    std::cout << "Server running at address " << myEngine.self() << std::endl;
    myEngine.define("sum", sum);

    return 0;
}
```

Optional request object. If not provided, the return value of the function will be used as the response.

Response type serialization automatically deduced via templates.

Argument type serialization automatically deduced via templates.

Lambdas work too!

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

namespace tl = thallium;

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

    tl::engine myEngine("ofi+tcp", 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((int)(x+y));
        };

    myEngine.define("sum", sum);

    return 0;
}
```

Client side: calling the RPC

```
int main(int argc, char** argv) {
      tl::engine myEngine("ofi+tcp", THALLIUM_CLIENT_MODE);
      tl::remote_procedure sum = myEngine.define("sum");
      tl::endpoint server = myEngine.lookup(argv[1]);

      int ret = sum.on(server)((int)42,(int)63);

      std::cout << "Server answered " << ret << std::endl;
      return 0;
}
```

Only the name is specified

Lots of template magic happening here

- `.on(endpoint)` creates a `callable_remote_procedure` bound to the target
- `(42, 63)` uses template deduction to serialize the arguments and call the RPC
- a `packed_data` object is returned, representing the response (not deserialized yet)
- casting to `int` deserializes the data automatically
- **Important: you are responsible for correctly matching types in the server and client!**
  - Compile with `-DTHALLIUM_DEBUG_RPC_TYPES` during development to add type checking (adds an overhead)

Custom type serialization

```
class point {

    private:

        double x;
        double y;

    public:

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

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

- Use the Cereal library to handle serialization of all primitive types and STL containers
- Write your own `serialize` function for custom types
- Or `save` and `load` functions
- They can be defined outside the class

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

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

Using RDMA: the client side

```
tl::engine myEngine("tcp", MARGO_CLIENT_MODE);
tl::remote_procedure remote_do_rdma = myEngine.define("do_rdma");
tl::endpoint server = myEngine.lookup(argv[1]);

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)(myBulk);
```

- You can expose multiple segments of memory as one `bulk` object
- `tl::bulk` objects can be sent as RPC argument
- Sending a `tl::bulk` object does NOT send the data it exposes
- You need to ensure the exposed memory is valid until the remote party is done accessing it

Using RDMA: the server side

```
std::function<void(const tl::request&, tl::bulk&)> f =
        [&myEngine](const tl::request& req, tl::bulk& remote_bulk) {
            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);
            remote_bulk.on(ep) >> local;
            req.respond();
        };
myEngine.define("do_rdma",f);
```

- `req.get_endpoint()` returns the address of the sender. It is needed to bind the received `bulk`
- We have a local buffer of 6 bytes, too few to transfer the whole remote memory, so the `>>` operator will figure it out and transfer only 6 bytes
- The `>>` operator does not have a typical stream semantics: subsequent operations will still transfer from the start of the remote memory, not from an offset

Using RDMA: the server side

```
std::function<void(const tl::request&, tl::bulk&)> f =
        [&myEngine](const tl::request& req, tl::bulk& remote_bulk) {
            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);
            remote_bulk.on(ep) >> local;
            req.respond();
        };
myEngine.define("do_rdma",f);
```

- You can select the part of the memory you wish to transfer, and in which direction (provided you set the right permissions when calling `expose`)

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

Use Argobots wrappers

- Just like margo, thallium will initialize Argobots if it is not already initialized, and finalize it when the engine is finalized, if it initialized it
- You should make sure no Argobots construct (lock, mutex, pool, execution stream, etc.) outlives the engine, unless you have manually initialized Argobots first

Locking mechanisms

- Thallium has a full set of wrappers for Argobots objects, e.g. `ABT_mutex` ⇒ `tl::mutex`
- You should use Argobots (or these wrappers), NOT POSIX equivalents (e.g. `std::mutex`)
- You can still use things like `std::unique_lock<tl::mutex>` (`tl::mutex` implements `lock()`)

Argobots constructs

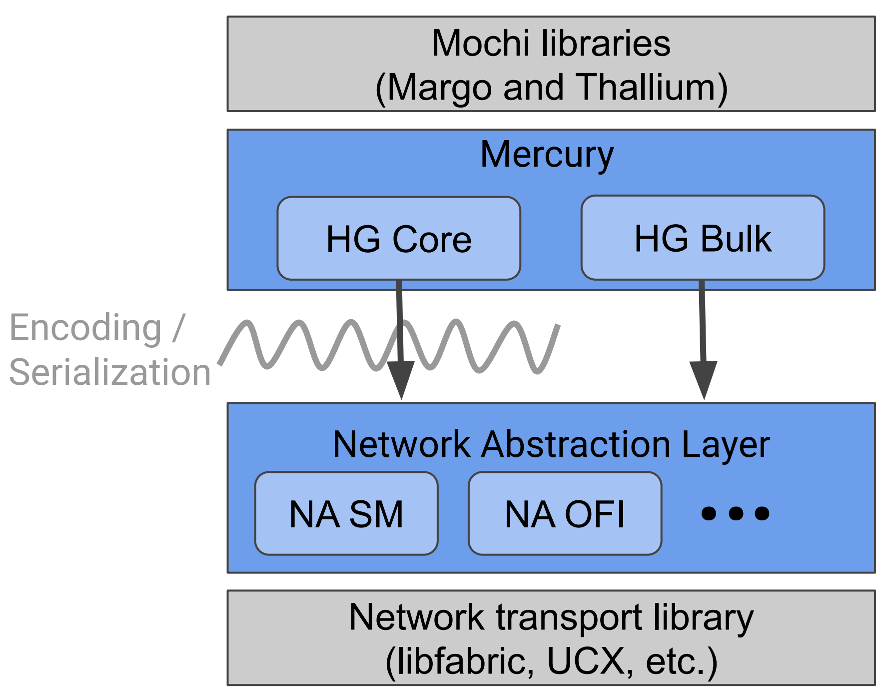
- Thallium has wrappers for `ABT_pool` (`tl::pool`) and `ABT_xstream` (`tl::xstream`), with functions to push work unit (ULTs into them). Don't use `std::async` or POSIX threads.

Using Argobots and Mercury features directly

Phil Carns

10 min

Mercury software architecture



Libfabric (OFI) is the preferred transport library.

Mercury is the core messaging layer in Mochi: it translates low level network primitives into more usable RPC constructs.

- RPCs (requests and responses) are sent through the HG Core routines.
  - Messages are encoded and buffered
  - Ideal for control and handshaking
- Bulk transfers are sent through HG Bulk routines.
  - Direct RDMA access
  - Ideal for larger data payload transfers

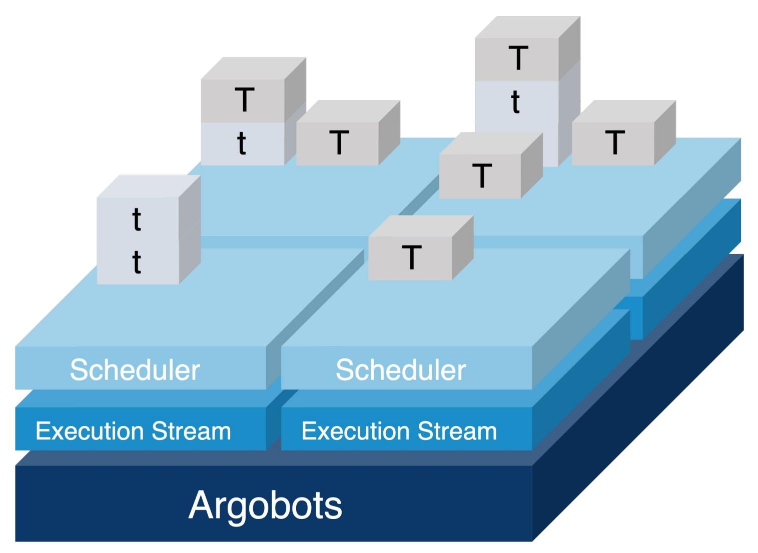
Mercury network protocols

- Every Mercury network address has a string form.
- It starts with a protocol specifier prefix like “**ofi+tcp://**”
  - In that example: use the TCP protocol provider in the OFI (libfabric) communication library.
  - This is sufficient for initializing Mercury.
- You can specify portions of the network address at startup (e.g., domain and/or port) but we generally recommend against it.
  - Instead: start with just the protocol specifier, let Mercury assign an address, and then retrieve the string for it with addr_self().
  - Or better yet: use helper libraries like Bedrock and SSG.
- What protocols are available? We’ll talk about that more later in the tutorial.

Mercury network addressing

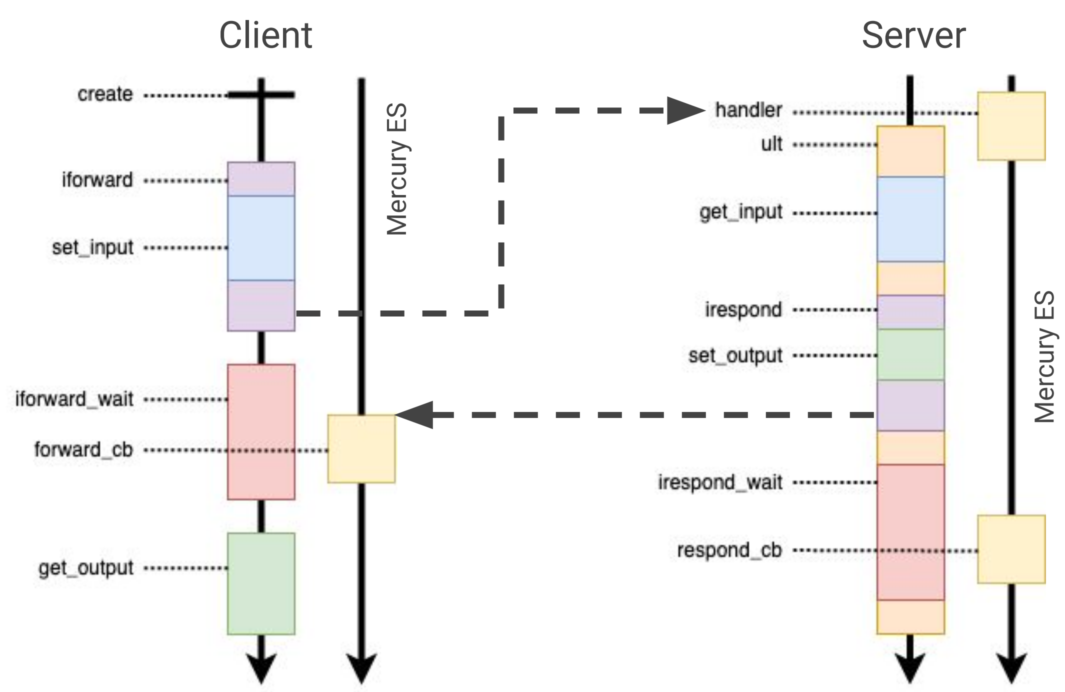
- The complete network address will be resolved to something more specific at runtime like “**ofi+tcp;ofi_rxm://192.168.122.1:37557**”
  - In this example the protocol prefix has also been expanded to include RXM.
  - The address portion is an IP address and port.
  - Most, *but not all,* networks use the hostname:port convention.
- See the server in hands-on exercise 1 for example of how to retrieve the local network address and convert it into a string to print.

Threading with Argobots



- **You don't have to use Argobots directly, but it's available if you need it!**
- User-level threads (ULTs) are very lightweight! Create as many as you need; they will be multiplexed on execution streams (ESs).
- **Argobots uses "cooperative" multithreading**: an execution stream cannot context switch to another ULT unless you yield control
  - (e.g. by calling a Mochi function or using an Argobots synchronization primitive).
- Argobots includes counterparts to all major pthread functions (ABT_thread_create(), ABT_thread_join, ABT_mutex_lock() etc.)
- Thallium includes C++ wrappers for these primitives as well.

Typical timeline of a Mochi RPC



- Mochi coordinates a lot of internal mechanisms in the RPC path.
- You can ignore details when you are starting, but they may eventually become important for performance tuning.
- Later today we'll talk about how to extract profiles and analyze them as an advanced feature.

Tip: Expect one process (and it's progress ES) to be able to saturate one NIC. You may need multiple processes to take advantage of multiple NICs.

Hands-on session 1

Using Margo and Thallium

Prerequisite

Follow the initial setup on Mochi's documentation to setup your docker image

bit.ly/3leh0yz

This Ubuntu image contains spack (already setup),
and the mochi-spack-packages repository

Instructions and objectives

Objectives

- Write your own RPC with either Margo or Thallium
- Understand argument/response serialization
- Understand and use RDMA

Instructions

- Choose your language (C/Margo or C++/Thallium)
- Follow Exercise 1 in your chosen section at bit.ly/3OgtRE8

Note

- Focus on the “insert” RPC, go back to implementing “lookup” if you have time or later
- The second part (RDMA) is a bonus, do it if you have time, or on your own later

The Mochi Methodology

Designing a Mochi service for composability

Matthieu Dorier

10 min

Services and Components

We always welcome new open-source contributions!

What is a Mochi component?

- Provides a single functionality (e.g., key/value storage)
- Is accessible via RPC/RDMA, using Margo or Thallium
- Can share its environment (Argobots and Mercury) with other components
- May have multiple backend implementations for the functionality

What is a Mochi service?

- Specific composition of Mochi components
- Specific (usually application-tailored) interface on top
- Specific data semantics and access requirements

Designing a Mochi service

User requirements

Data model
Access Pattern
Consistency guarantees
Fault tolerance

Service requirements

Data organization
Metadata organization
User interface

+

Composition, interfacing, evaluation

Bedrock or glue-code
API implementation
Evaluation and testing

Components

Existing components
New backends
New components

The Mochi methodology

The Mochi methodology revolves around a dialogue between service implementers and service users to understand their applications and use-cases, craft a relevant API, reuse existing components and develop new ones as needed, with composability, reusability, and configurability in mind

Understand your user’s requirements

Which data model?

- Arrays, meshes, objects, etc.
- Namespace, metadata, hierarchy, etc.

Which access pattern?

- Characteristics (e.g. access sizes, burstiness)
- Collective/individual accesses
- Blocking/non-blocking

Which guarantees?

- Consistency
- Performance
- Persistence

User requirements

Data model
Access Pattern
Consistency guaranties
Fault tolerance

Map user requirements to the service

How should data be organized?

- Sharding
- Distribution
- Replication

How should metadata be organized?

- Characteristics (e.g. access sizes, burstiness)
- Collective/individual accesses
- Blocking/non-blocking

How do clients interface with the service?

- Programming language
- API

Service requirements

Data organization
Metadata organization
User interface

Components: don’t reinvent the wheel!

Components

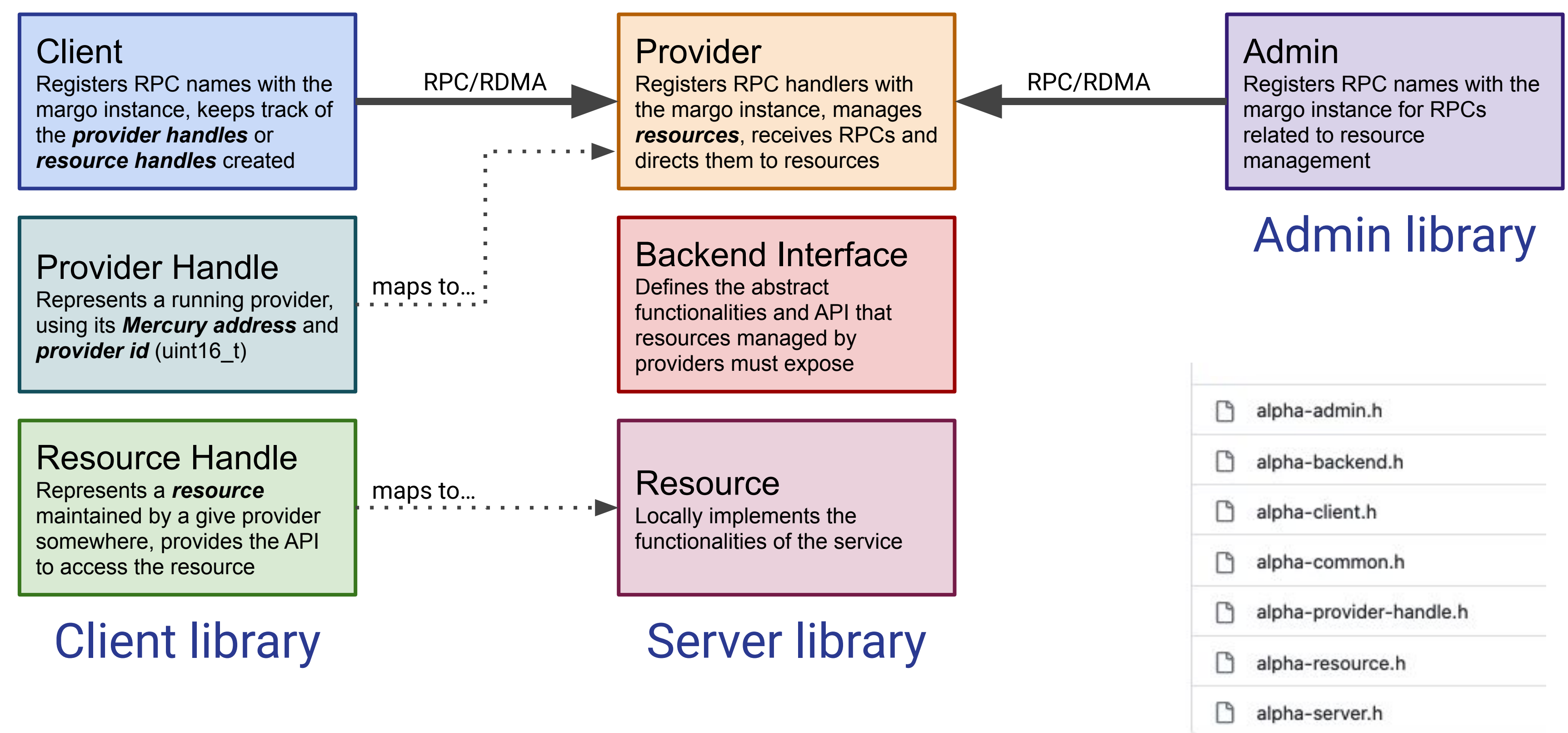
Existing components
New backends
New components

We already provide the following components

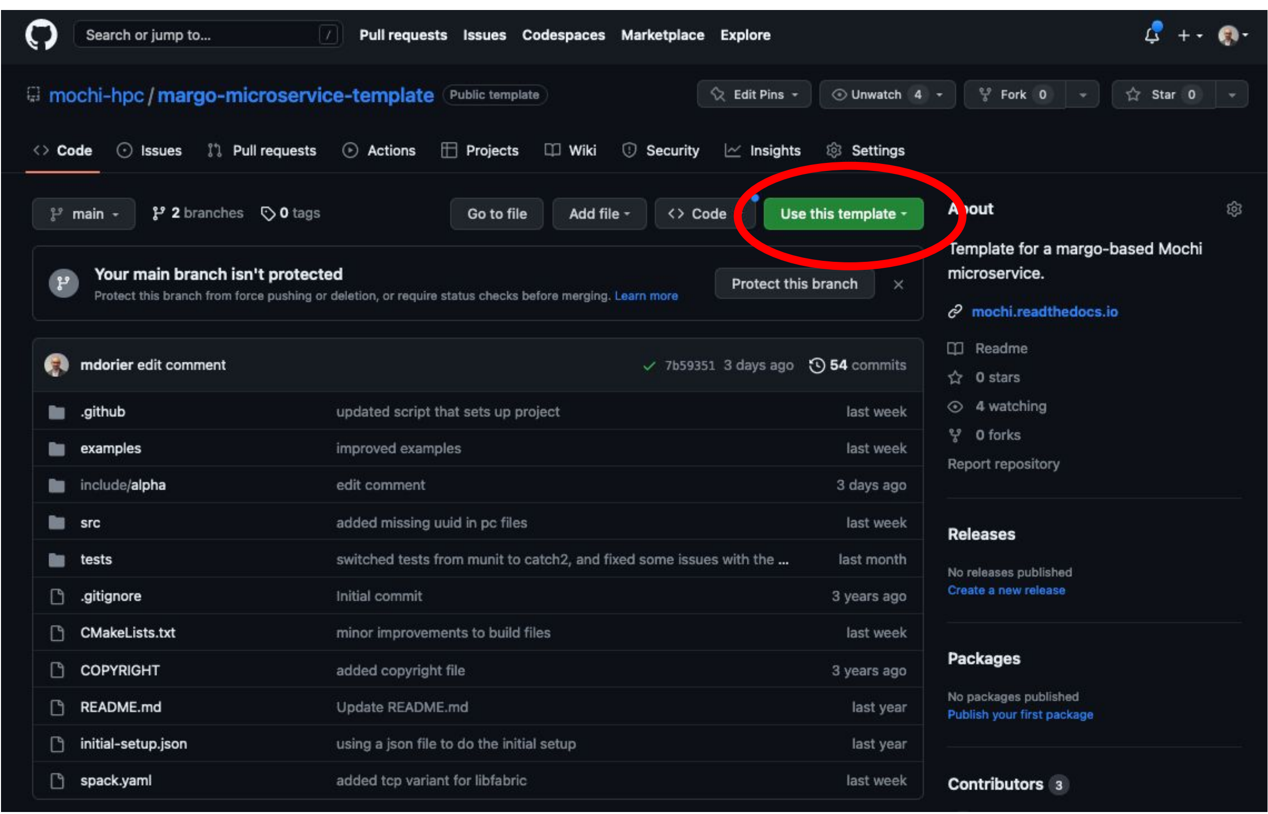
mochi-yokan	Key/value and document storage
mochi-bake	Blob storage
mochi-poesie	Embedded scripting
mochi-abt-io	Wrappers for POSIX I/O
mochi-remi	File migration
mochi-ssg	Gossip-based failure detection
mochi-raft	Replicated state machine

If some have missing features, let us know!

Architecture of a Mochi component



The Margo and Thallium templates



Provides a starting project with the annoying code already filled in so you can focus on what matters:

- The API
- The features

The templates als provide

- Unit tests (catch2)
- Github actions for automated testing and code coverage (codecov.io)

Rely on spack for dependencies (spack.yaml) and on cmake for building the code

Composition and interface

Composition and interfacing

Bedrock or glue-code API implementation

Composition

- Prefer Bedrock configuration to hand-written glue code
- Think in terms of dependency injection

Interface

- Have client-side handles representing server-side resources
- Think of non-blocking interfaces
- Think of forwarding interfaces (taking an already created bulk handle)
- Think of potential language bindings (e.g. Python)
    - Look at py-mochi-margo and at how Yokan provides its python binding (internal to the Yokan code base, built by cmake), or Bake (external, built by setuptools like normal python packages)

Hands-on Session 2

Let’s code our own microservice

Instructions and objectives

Objectives

- Use the Margo or Thallium microservice template to write your own “phonebook” component
- Understand the typical architecture of a Mochi component

Instructions

- Follow the instructions in the Exercise 2 section of your chosen language/library

Note

- You don’t need to have completed Exercise 1
- Again, focus on the “insert” function and do the “lookup” if you have time, or later

Interlude

Mochi’s role in the GekkoFS distributed file system and the ADMIRE project

Marc-André Vef
🕑 10 min

History of using Mochi

- **2017**: First GekkoFS commit January 2017 (funded by the German ADA-FS project)
  - Using Mercury since version 0.9
  - Using Margo before first version 0.1 was released
- **Since 2018**: Barcelona Supercomputing Center (BSC) GekkoFS collaboration
- **2019**: GekkoFS ranked 4th in IO500's 10-node challenge at Supercomputing
- **2020**: DelveFS – semantic file system for object stores
  - Uses Mercury and Margo in a FUSE file system
- **Since 2021**: GekkoFS funded by the ADMIRE and FIDIUM projects

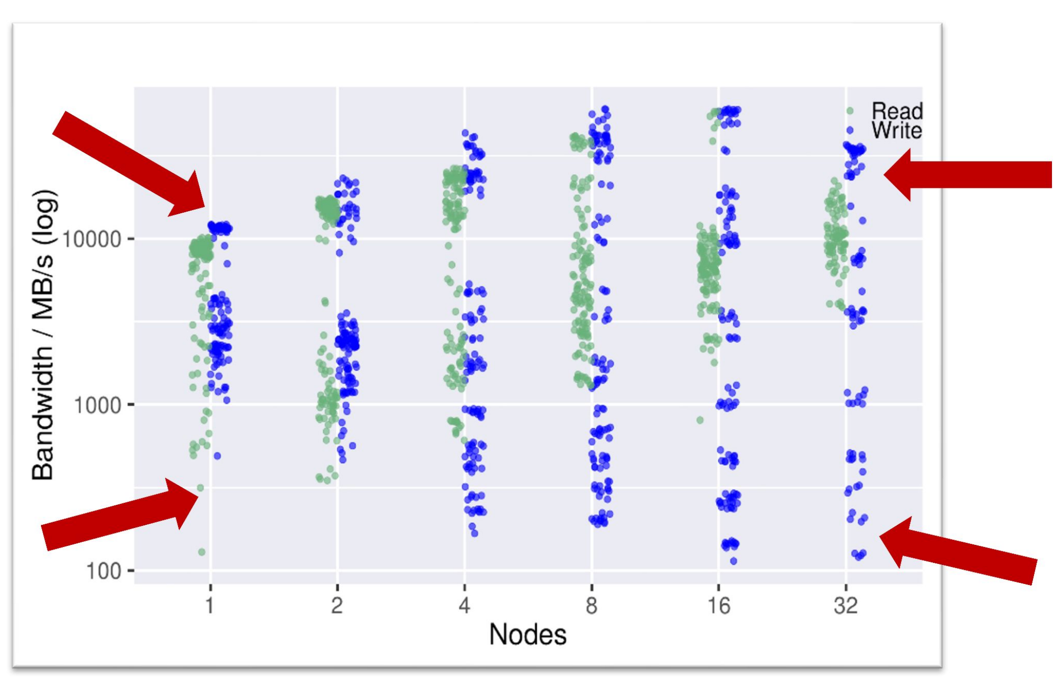
M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann.
GekkoFS - A Temporary Distributed File System for HPC Applications. In 2018 IEEE International Conference on Cluster Computing (CLUSTER)

M.-A. Vef, R. Steiner, R. Salkhordeh, J. Steinkamp, F. Vennetier, J.-F. Smigielski, and A. Brinkmann. 2020. DelveFS - An Event-Driven Semantic File System for Object Stores. In 2020 IEEE International Conference on Cluster Computing (CLUSTER)

Unpredictability of parallel file systems (PFSs)

I/O performance varies wildly <u>for identical workloads</u>

Applications suffer due to PFS load!



MareNostrum4 @ Barcelona Supercomputing Center (Spain)

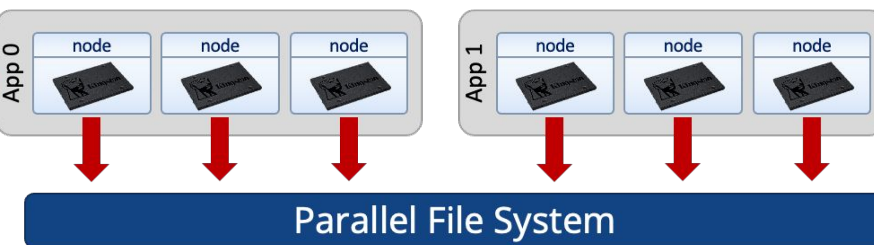
Moving from this …

Data manipulations rely on the PFS

- Uncoordinated application I/O to/from PFS
- Node-local storage typically ignored
- Increased PFS contention and performance variability

node-local storage mostly unused

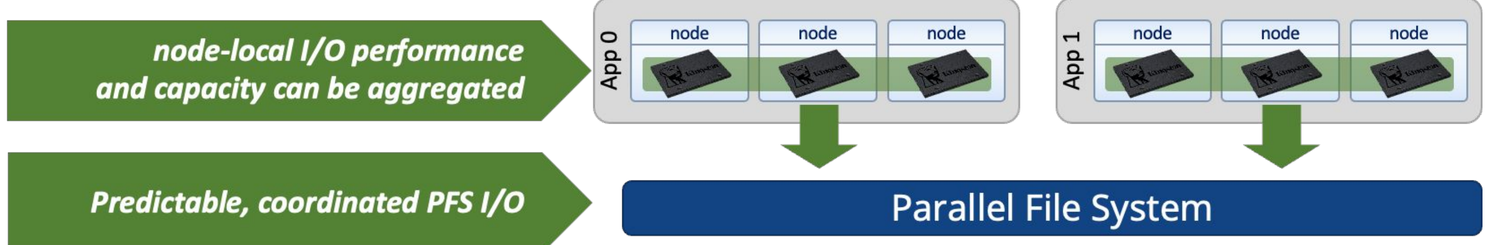
unpredictable and random PFS I/O



… to this

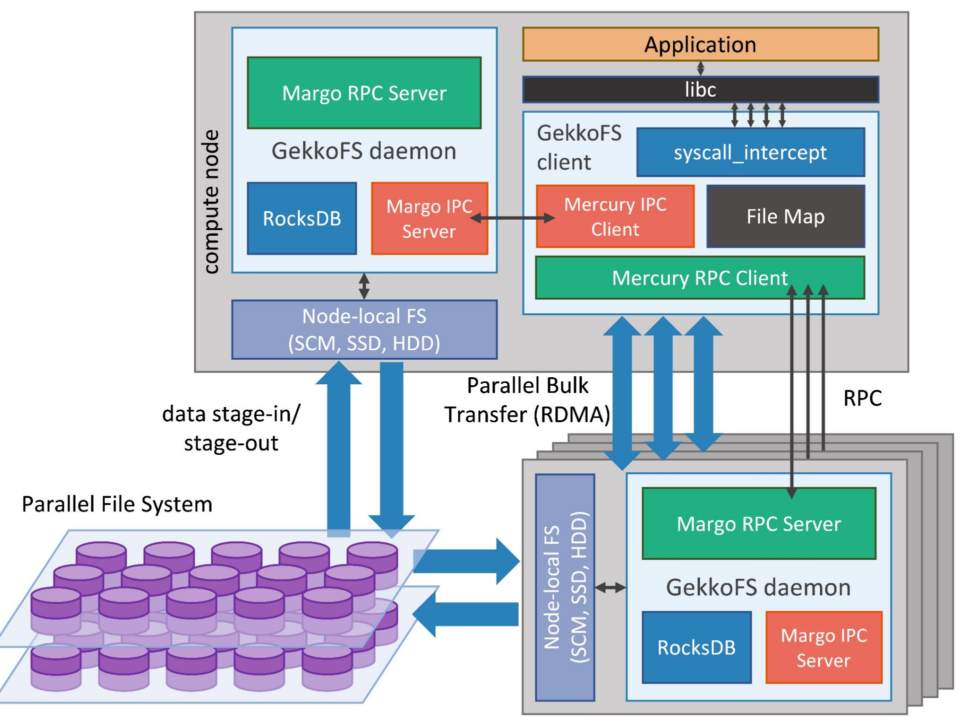
Data manipulations rely on node-local storage

- Coordinated application I/O that fits the PFS
- Harmful I/O patterns are absorbed by node-local storage
- Reduced PFS contention and performance variability



GekkoFS

1. **Scalability**
   - No central components
   - Linear scaling
2. **Fast deployment**
   - Wall time is important
   - <10 seconds for deployment
3. **User space**
   - User decides
   - No administrative support
4. **Hardware independence**
   - Use accessible storage
   - Use fast network fabrics

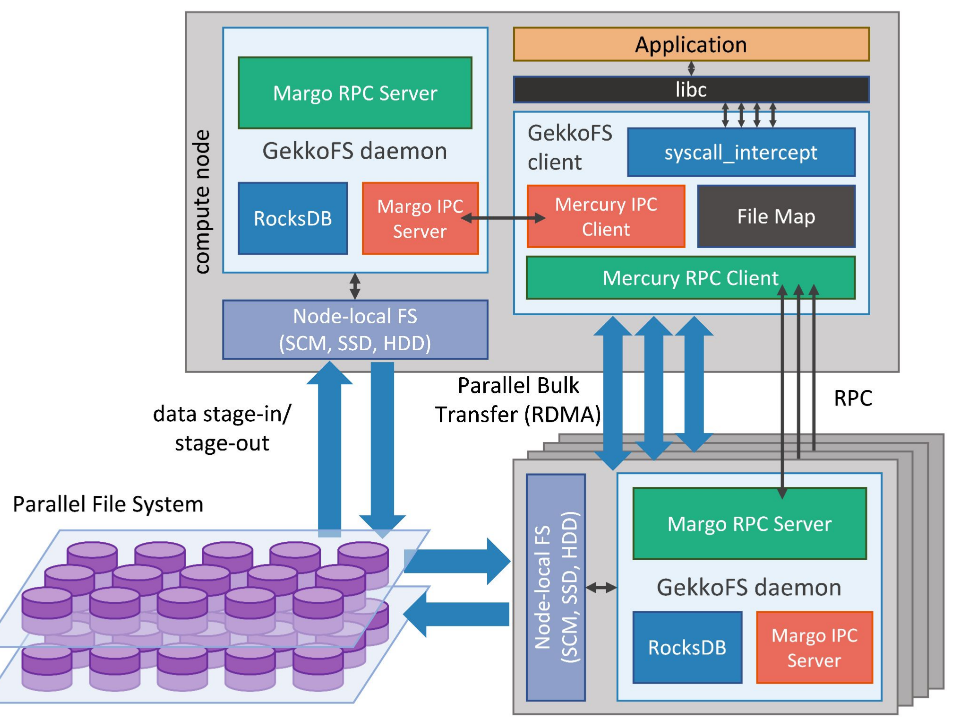


GekkoFS is open source: https://storage.bsc.es/gitlab/hpc/gekkofs/

M.-A. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann. GekkoFS - A Temporary Burst Buffer File System for HPC Applications. In 2020 Journal of Computer Science and Technology (JCST)

GekkoFS

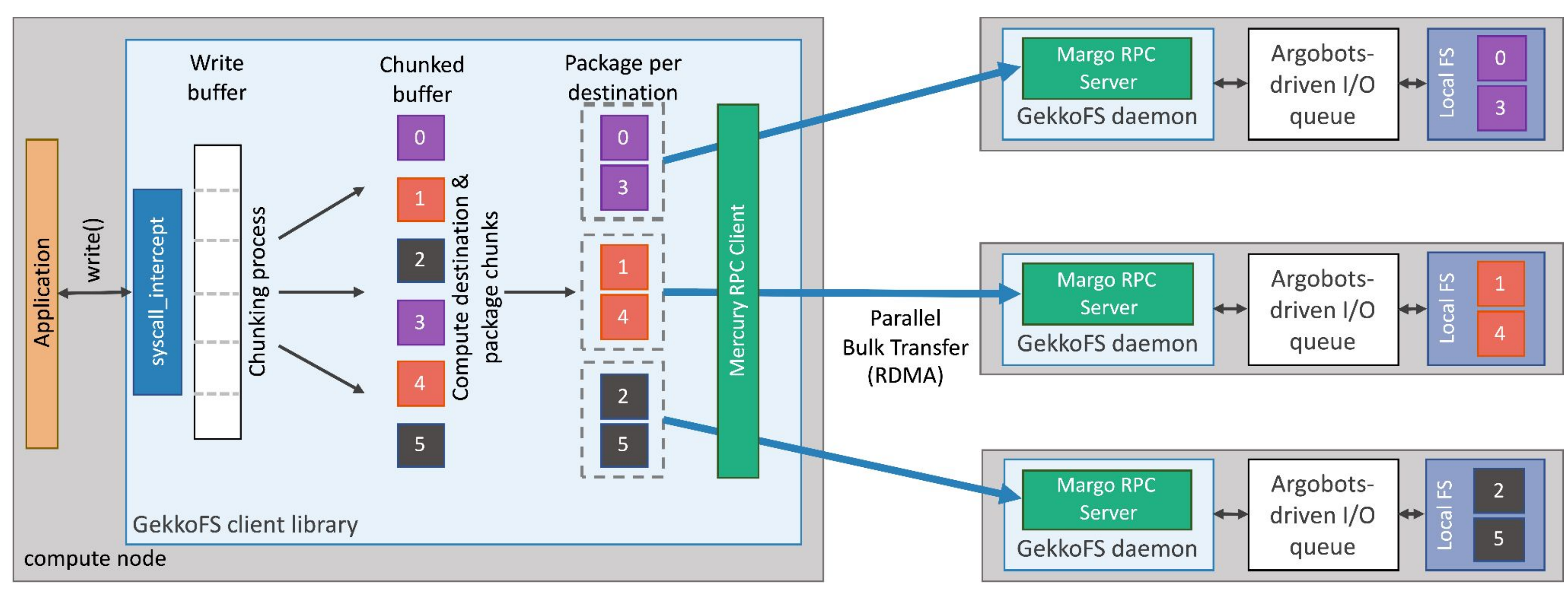
1. **Scalability**
   - No central components
   - Linear scaling
2. **Fast deployment**
   - Wall time is important
   - <10 seconds for deployment
3. **User space**
   - User decides
   - No administrative support
4. **Hardware independence**
   - Use accessible storage
   - Use fast network fabrics



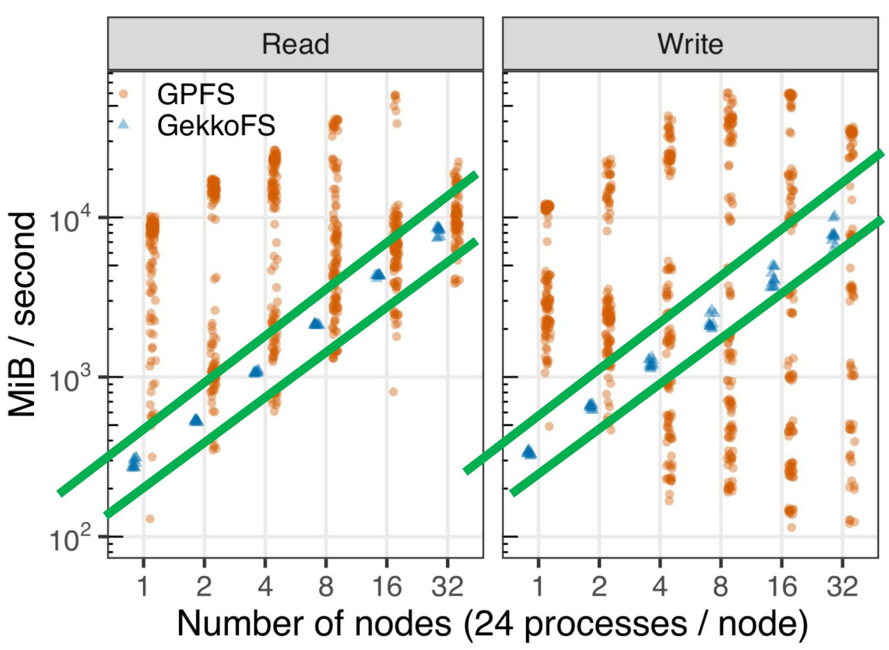
GekkoFS is open source: https://storage.bsc.es/gitlab/hpc/gekkofs/

Mochi tools: Mercury, Argobots, and Margo fundamentally contribute to GekkoFS's performance

GekkoFS with Mercury's bulk buffer transfers



Performance variability revisited

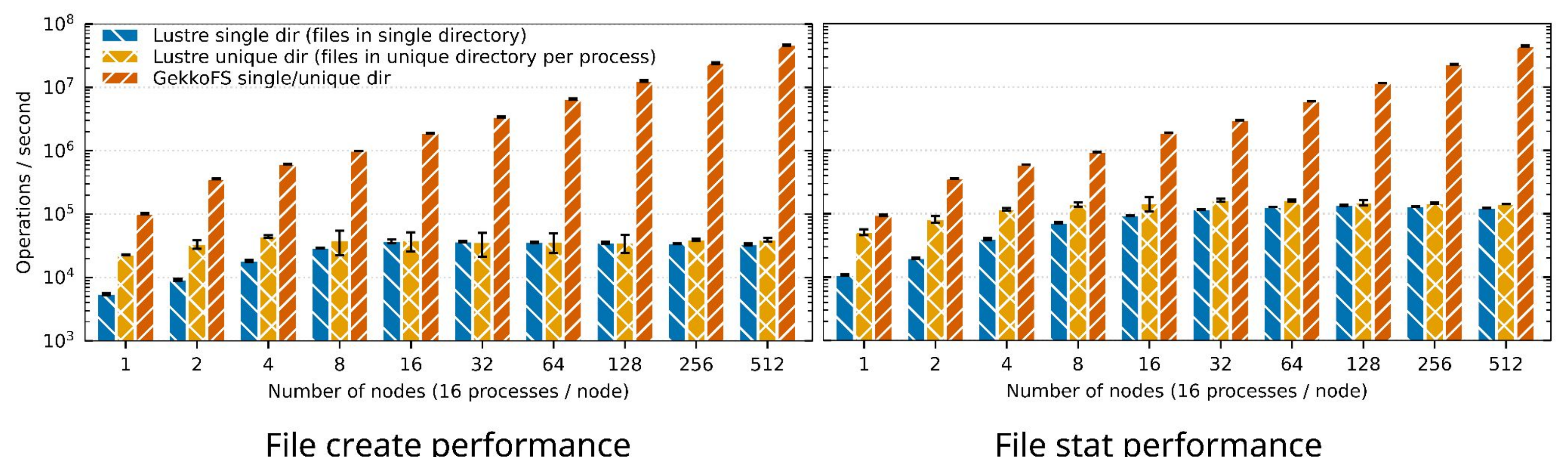


MareNostrum4 @ Barcelona Supercomputing Center (Spain)

I/O performance variability greatly reduced

Metadata performance

- GekkoFS weakly scaled (100K files per process)
  - More than 819 million files in total with 512 nodes



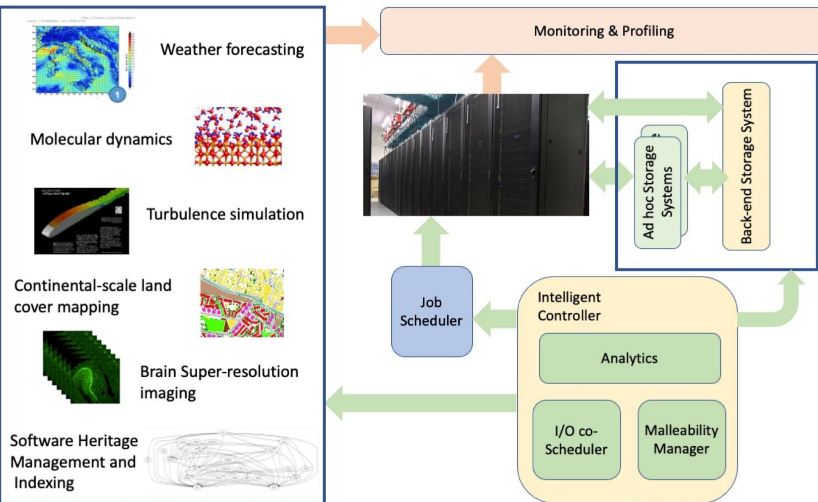
File create performance

File stat performance

MOGON II @ Johannes Gutenberg University Mainz (Germany)

The EuroHPC ADMIRE project

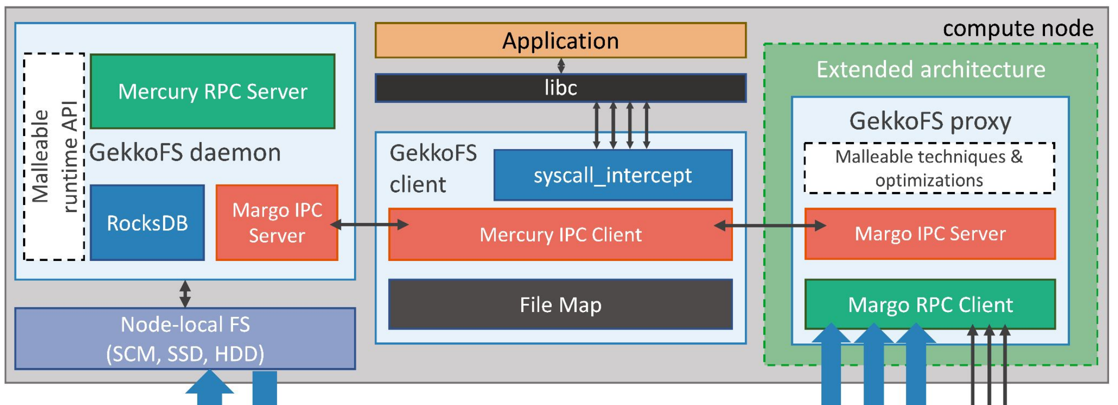
- € 7.9 million overall budget (2021-2024)
- 19 partners in 6 EU countries
- Key points:
    - Adaptive multi-tier data management
    - Computational and I/O malleability
    - Focus on ad-hoc storage systems
    - Use cases from industry and academics
- Mochi tools involvement
    - Margo as a communication framework between ADMIRE components
    - Data movement between PFS and ad-hoc storage systems



ADMIRE project architecture @ https://admire-eurohpc.eu

Recap

- Mochi tools helped us greatly in various use cases
  - Ease of use and network independence
  - Low-latency and high-throughput (GekkoFS, DelveFS)
  - Communication framework (ADMIRE)
- What's next?
  - Supporting Bedrock for file system malleability
  - Overhauling GekkoFS architecture and migrating parts to Thallium



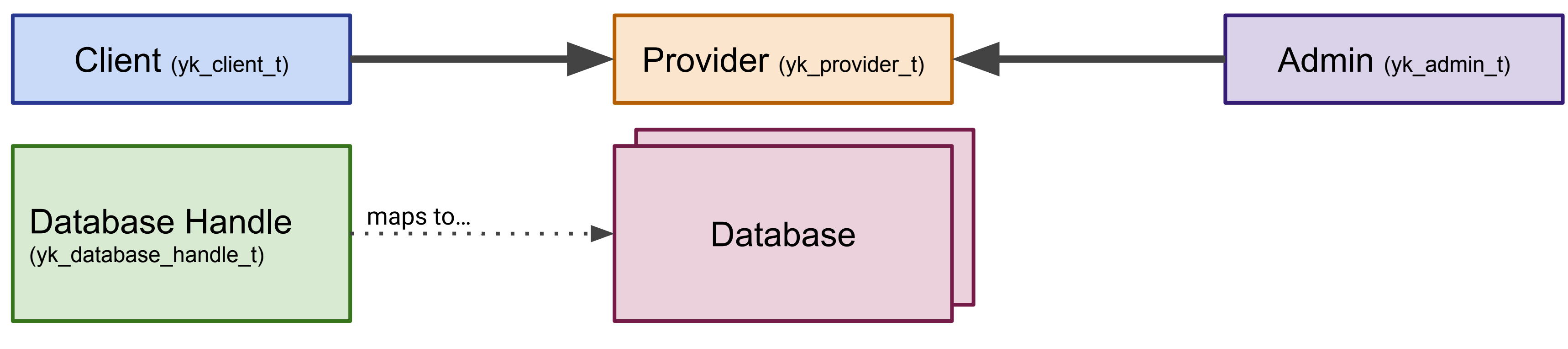
Some Mochi components

Reuse, compose, and contribute

Yokan

Key/value and document storage

Yokan in one picture



- Client library for databases accesses (put/get/erase/…)
- Database identified by a UUID

- Many available database backends (rocksb, leveldb, etc.)

- Admin library to create and destroy databases

- Key/value storage: caller chooses the keys and values
    - Database can be sorted (sort function may be specified) or unsorted
- Document storage: a database stores collections, a collection contains multiple documents identified by a numerical ID
    - User choose the target collection and document content, ID is returned by the database

Provider and Admin interface

Provider

```c
yk_return_t ret = yk_provider_register(mid, 42, NULL, YOKAN_PROVIDER_IGNORE);
```

Admin

```c
yk_admin_t admin = YOKAN_ADMIN_NULL;
yk_database_id_t db_id;

ret = yk_admin_init(mid, &admin);

ret = yk_open_database(admin, server_addr, provider_id, NULL, "map", "{}", &db_id);

char db_id_str[37];
yk_database_id_to_string(db_id, db_id_str);
printf("Database id is %s (take note of it!)\n", db_id_str);

ret = yk_admin_finalize(admin);
```

Client interface

Client

```
yk_client_t client = YOKAN_CLIENT_NULL;
yk_database_id_t db_id;

ret = yk_client_init(mid, &client);

yk_database_id_from_string(db_id_str, &db_id);

yk_database_handle_t db_handle = YOKAN_DATABASE_HANDLE_NULL;
ret = yk_database_handle_create(client, server_addr, provider_id, db_id, &db_handle);

...

ret = yk_database_handle_release(db_handle);
ret = yk_client_finalize(client);
```

Storing and retrieving key/value pairs

```
const char* key = "matthieu";
const char* value_in = "dorier";

ret = yk_put(db_handle, YOKAN_MODE_DEFAULT, key, strlen(key), value_in, strlen(value_in));

char value_out[128];
size_t value_out_size = 128;
ret = yk_get(db_handle, YOKAN_MODE_DEFAULT, key, strlen(key), value_out, &value_out_size);
```

Other functionalities

- Put/get multiple key/value pairs (`put_multi`, `get_multi`, `put_packed`, `get_packed`)
- Erase, check existence, get value length (`erase`, `exists`, `length`), with `_multi` variants
- List key/value pairs (`list_keys(_packed)`, `list_keyvals(_packed)`)
- Use filters to return only some of the key/value pairs (simple filters like key prefix, or more complex using Lua scripting)

Document storage

Collections

- List of documents, identified by an ID starting from 0 and increasing as new documents are added
- A database may contain multiple collections, identified by a name

```
ret = yk_collection_create(db_handle, "my_collection", YOKAN_MODE_DEFAULT);

const char* document = "This is a document";
size_t doc_size = strlen(document);

yk_id_t id;
ret = yk_doc_store(db_handle, "my_collection", YOKAN_MODE_DEFAULT, document, doc_size, &id);
printf("Document has id %lu\n", id);

char buffer[128] = {0};
size_t buf_size = 128;
ret = yk_doc_load(db_handle, "my_collection", YOKAN_MODE_DEFAULT, id, buffer, &buf_size);
```

Document storage

Collections

- List of documents, identified by an ID starting from 0 and increasing as new documents are added
- A database may contain multiple collections, identified by a name

Document storage functionalities

- Storing/loading one or multiple documents (`doc_store/load_(multi/packed)`)
- Listing documents, possibly using filters (e.g. Lua) to select relevant ones
- Updating documents (`update_(multi/packed)`)
- Erasing document (ID is not reused, loading again will return an error)
- Getting the length of a document (`length(_multi)`)

⚠️ Document storage is built on top of key/value storage, so if you use a database as a document store, don't use the key/value store interface or you might mess up your collections.

Change the semantics with modes

- Most functions have a “mode” parameter, usually set to `YOKAN_MODE_DEFAULT` (0)
- Modes (found in yokan/common.h) will alter the semantics of these functions

Some examples

- `YOKAN_MODE_APPEND` The put functions will append the values to existing ones instead of overwriting them
- `YOKAN_MODE_NEW_ONLY` The put or store functions will add the key/value or document only if it does not exist
- `YOKAN_MODE_CONSUME` The get functions will also erase the key/value pair
- `YOKAN_MODE_NO_RDMA` The function will not rely on RDMA for data transfers, passing data as RPC arguments instead
- `YOKAN_MODE_INCLUSIVE` The list functions will include the provided starting key if found
- …

⚠️ Not all database backends support each mode. Look at the list on mochi.readthedocs.io to select a backend that provides the mode you want.

Backends

Yokan provides many database backends

- In-memory backends: map, unordered_map, set, unordered_set
- Storage-backed backends: rocksdb, leveldb, lmdb, gdbm, tkrzw, unqlite, berkeleydb
- Some backends have sub-backends (e.g. berkeleydb and tkrzw have a sorted and a non-sorted backend, and some backends have options for being in-memory)
- All these backends are highly configurable

You can implement your own backend easily (see mochi.readthedocs.io for instructions)

⚠️ Some backends are not sorted (e.g. gdbm, unordered_map,...), some backends do not store values (e.g. set, unordered_set)

⚠️ Document storage is only possible on top of backends that store values (obviously)

⚠️ Document storage on top of backends that are not sorted risk a performance penalty when calling `doc_list` functions.

Last note on Yokan

Key/value storage is the most common data service you may want, and we have seen many Mochi users implement their own (before Yokan was implemented). If you need a key/value store, consider using Yokan before jumping to re-implementing the wheel. If you think that Yokan does not provide a specific feature that you need, talk to us!

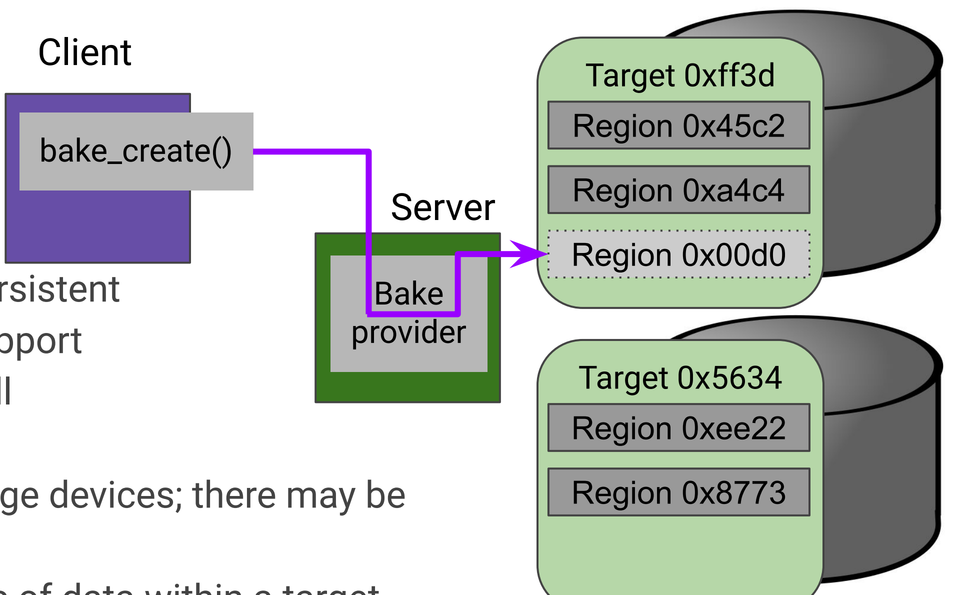
Bake

Blob storage

Phil Carns

10 min

What is Bake?



Bake is a “blob store” microservice: it provides bare-bones access to large blobs of data.

- Originally designed for efficient access to persistent memory but has now been generalized to support conventional local file system devices as well
- Terminology:
  - **Targets** are roughly analogous to storage devices; there may be multiple per provider
  - **Regions** are uniquely addressable units of data within a target
- Semantics:
  - Bake generates all identifiers (represented with hex numbers above)
  - Meant to be paired with an indexing service (e.g. Yokan)
  - There is no “append”; region sizes are defined at create time
  - Read and write concurrently as much as you like within a region
  - There is an explicit “persist” operation to make data durable and immutable

Backend types

PMEM backend

- Uses the PMDK library (https://pmem.io/pmdk/) for access
- Intended for use on byte addressable memory devices
    - Could be an NVDIMM, dynamic memory, or even a memory-mapped conventional device
- Target is pre-allocated to a specific total size (as if you are using an NVDIMM)
- Most efficient for true, RDMA-capable memory devices with byte-granular access

File backend

- Stores data in local POSIX files and directories
- Files are accessed using the abt-io Mochi component to allow for high concurrency
- Default mode uses a log structure with block alignment; data is buffered for RDMA
- Future versions will leverage liburing and a multilog format
- Most efficient for block-based storage devices (hard drive, NVMe, etc.)

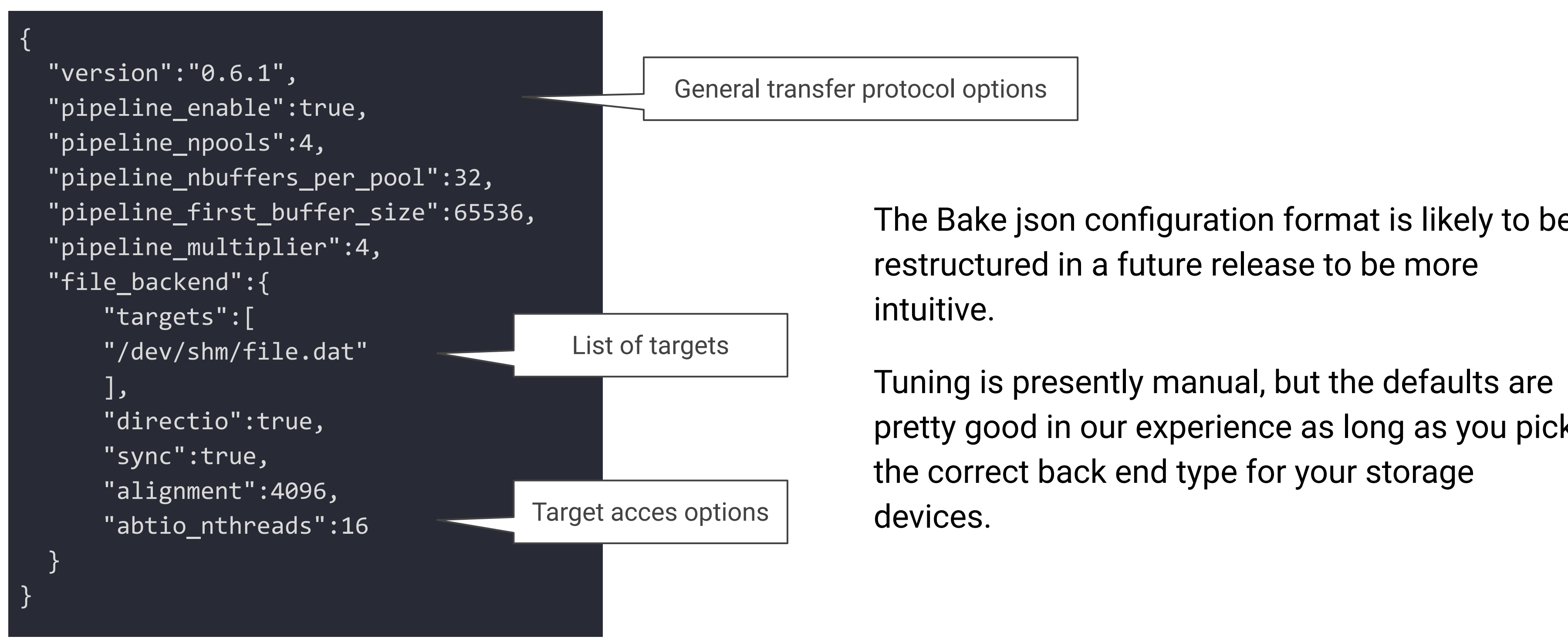
Provider interface

```
#include <bake-server.h>

bake_provider_t provider = NULL;
struct bake_provider_init_info init_info =
             BAKE_PROVIDER_INIT_INFO_INITIALIZER;

int ret = bake_provider_register(mid, 42, &init_info, &provider);

bake_target_id_t tid;

ret = bake_provider_create_target(
        provider, "pmem:/dev/shm/mytarget.dat",
        8388608, &tid);

char tid_str[37];
ret = bake_target_id_to_string(tid, tid_str, 37);
margo_info(mid, "Target ID is %s", tid_str);
```

Backends include pmem and file

Target IDs are UUID

Configuration



Client interface

```c
#include <bake-client.h>

const char* target_id_str = ...;

bake_client_t client;
ret = bake_client_init(mid, &client);

bake_provider_handle_t ph;
ret = bake_provider_handle_create(
        client, server_address, 42, &ph);


…

ret = bake_provider_handle_release(ph);
ret = bake_client_finalize(client);
```

```c
bake_target_id_t tid;
if(target_id_str) {
    ret = bake_target_id_from_string(
        target_id_str, &tid);
} else {
    uint64_t num_targets;
    ret = bake_probe(ph, 1, &tid, &num_targets);
}

bake_region_id_t rid;
ret = bake_create(ph, tid, 10, &rid);

char in_buffer[10] = ...;
ret = bake_write(ph, tid, rid, 0, in_buffer, 10);

ret = bake_persist(ph, tid, rid, 0, 10);

char out_buffer[10];
uint64_t bytes_read;
ret = bake_read(
    ph, tid, rid, 0, out_buffer, 10, &bytes_read);
```

Client interface

Core functions

- bake_create(): create region
- bake_write/bake_read(): access data in a region
- `bake_get_size`(): get the size of a region
- `bake_remove`(): remove a region
- `bake_create_write_persist` (): create, write, and persist a region in one call
  - This is a compound operation to minimize round trips for a common pattern

Proxy functions: What if one provider wants to relay data transfers to Bake? You could buffer data and then call bake_write(), but that introduces an extra memory transfer. Alternatively you can use "proxy" functions that let you delegate a bulk handle to Bake so that it can perform the RDMA transfer itself.

- `bake_proxy_write`(): write from a bulk handle instead of a local buffer
- `bake_proxy_read`(): read into a bulk handle instead of a local buffer
- `bake_create_write_persist_proxy` (): same as `bake_create_write_persist` but with a bulk handle

Bedrock

Configuration and bootstrapping

Matthieu Dorier

10 min

Why Bedrock?

- Server code often looks the same
  - Read input configuration
  - Start providers
  - Call `margo_wait_for_finalize` or `engine::wait_for_finalize`
- Bedrock provides
  - A unified JSON-based configuration format to describe your service
  - A daemon that reads such configuration and spins up providers
- Bonus
  - Write/generate your configuration using Python
  - Query the configuration remotely at any time
  - Change the configuration (add/remove providers, Argobots pools and ES, etc.)

Getting started with Bedrock

Installing

```
spack install mochi-bedrock
```

Once installed

```
bedrock <protocol> -c <configuration.json>
```

Can be launched on multiple nodes with mpirun or any launcher you have on your machine

Example configuration

see https://mochi.readthedocs.io/en/latest/bedrock/02_json.html

Example configuration

```
{
  "margo" : {
    "mercury" : { },
    "argobots" : {
      "abt_mem_max_num_stacks" : 8,
      "abt_thread_stacksize" : 2097152,
      "pools" : [
        {
          "name" : "my_rpc_pool",
          "kind" : "fifo_wait",
          "access" : "mpmc"
        }
      ],
      "xstreams" : [
        {
          "name" : "my_rpc_xstream",
          "cpubind" : 2,
          "affinity" : [ 2, 3, 4, 5 ],
          "scheduler" : {
            "type" : "basic_wait",
            "pools" : [ "my_rpc_pool" ]
          }
        }
      ]
    },
    "progress_pool" : "__primary__",
    "rpc_pool" : "my_rpc_pool"
  },
```

```
  "abt_io" : [
    {
      "name" : "my_abt_io",
      "pool" : "__primary__"
    }
  ],
  "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"
      }
    }
  ]
}
```

How do I enable Bedrock for my service?

In C, write a dynamic library (.so) containing the following structure

```
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)
```

See https://mochi.readthedocs.io/en/latest/bedrock/04_c_module.html for more information (in particular on how to declare dependencies and retrieve them)

How do I enable Bedrock for my service?

In C++, write a dynamic library (.so) inheriting from
`bedrock::AbstractServiceFactory`

```
#include <bedrock/AbstractServiceFactory.hpp>

class ServiceBFactory : public bedrock::AbstractServiceFactory {
    /* ... */
};

BEDROCK_REGISTER_MODULE_FACTORY(module_b, ServiceBFactory)
```

See https://mochi.readthedocs.io/en/latest/bedrock/05_cpp_module.html for more information

Querying the configuration of a server

```
bedrock-query <protocol> -a <address>
```

```
bedrock-query <protocol> -a <address1> -a <address2> -a <address3>
```

```
bedrock-query <protocol> -s <filename>
```

```
bedrock-query <protocol> -a <address> -j query.jx9
```

```
$providers = $__config__.providers;
$names = [];
foreach($providers as $p) {
    array_push($names, $p.name);
}
return $names;
```

- You can query multiple processes at the same time, the resulting JSON will map addresses to their configuration
- You can use an SSG file to get the addresses
- You can use the Jx9 language to process the configuration before it is returned

Initializing Bedrock using Jx9

```
$config = {
   margo : {
     argobots : {
       pools : [ { name : "my_pool" } ]
     }
  }
};

for ($i = 0; $i < $num_extra_pools; $i++) {
  $pool_name = sprintf("extra_pool_$i");
  array_push($config.margo.argobots.pools, { name : $pool_name } );
}

return $config;
```

```
bedrock ofi+tcp --jx9 -c example.jx9 --jx9-context num_extra_pools=4
```

Write a Jx9 script that creates the configuration

Provide parameters when launching Bedrock

Can be useful for parameterized configurations

Creating your configuration programmatically using Python

```python
my_process = bedrock.spec.ProcSpec(margo='na+sm')

# Create the new pool, it will be added automatically to the process
my_rpc_pool = my_process.margo.argobots.pools.add(name='my_rpc_pool',
kind='fifo', access='mpmc')
# Create two execution streams using that pool
for i in range(0,4):
    sched = bedrock.spec.SchedulerSpec(type='basic', pools=[my_rpc_pool])
    my_process.margo.argobots.xstreams.add(name=f'my_xstream_{i}',
scheduler=sched)
# Now let's set the pools we want for handling RPCs
my_process.margo.rpc_pool = my_rpc_pool

# Print the resulting configuration
print(my_process.to_json(indent=4))
```

- This method validates that the configuration is correct
- Can be used as part of an ML-driven workflow, when Python is also used for other tasks
- See https://github.com/mochi-hpc/py-mochi-bedrock/ for more information

Other components

SSG, ABT-IO, REMI, POESIE

Phil Carns

10 min

SSG: Group membership and fault detection

What is SSG?

- Group membership component
- Bootstrap from MPI, PMIx, or a list of address
- Allows processes to join an leave
- Membership changes (including crashes) propagated via the SWIM protocol

Why should I use it?

- Great way to setup a Mochi service on multiple processes/nodes
- Great way to provide service information to a client (by reading an SSG group file)
- Allow groups to change over time

```
#include <ssg.h>

static void my_membership_update_cb(void* uargs,
      ssg_member_id_t member_id,
      ssg_member_update_type_t update_type)
{...}
...
ssg_group_id_t gid;
ret = ssg_group_create(
    mid, "mygroup", array_of_addresses,
    num_addresses, &config,
    my_membership_update_cb, NULL, &gid);
...
int size;
ret = ssg_get_group_size(gid, &size);
...
ret = ssg_group_leave(gid);
```

ABT-IO: POSIX I/O for Mochi components

What is ABT-IO

- Argobots wrappers for POSIX I/O function (pwrite, pread, open, close, etc.) including non-blocking versions
- Only depends on Argobots

Why should I use it?

- POSIX I/O functions block the whole ES they run on, they don't yield to other ULTs
- ABT-IO offloads I/O operations to dedicated ES
- For recent kernel versions, ABT-IO will take advantage of I/O uring, hiding its complexity behind a POSIX-like interface

```
// abt_io_init takes the number of
// dedicated ES to create
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
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 is
finalized
abt_io_finalize(abtio);
```

REMI: File migrations

What is REMI

- Component to migrate file from a local storage on one node to the local storage on another
- Works with the notion of “fileset” to be migrated, with some associated key/value metadata

Why should I use it?

- Many microservices have resources that are backed by files
    - e.g. a database file, a storage target, etc.
- REMI is a generic solution to migrating a resource from one provider/process to another

```
// initialize REMI client
remi_client_t remi_clt;
ret = remi_client_init(mid, abtio, &remi_clt);
// create REMI provider handle
ret = remi_provider_handle_create(
    remi_clt, svr_addr, 1, &remi_ph);
// create a fileset
ret = remi_fileset_create(
    "my_migration_class", local_root, &fileset);
ret = remi_fileset_register_file(
    fileset, "/path/to/my/file");
 // fill the fileset with some metadata
ret = remi_fileset_register_metadata(
    fileset, "ABC", "DEF");
// check if we can compute the size of the fileset
size_t size = 0;
ret = remi_fileset_compute_size(fileset, 1, &size);
remi_fileset_set_xfer_size(fileset, 4);
int status = 0;
ret =  remi_fileset_migrate(
    remi_ph, fileset, remote_root,
    REMI_KEEP_SOURCE, REMI_USE_ABTIO, &status);
```

POESIE: executing scripts

What is POESIE

- Embed a Python or Lua interpreter in a Mochi service, and send code to it via RPC

Why should I use it?

- Can be a good way to enable in-service processing capabilities

Note: several existing Mochi components, such as Yokan and Bedrock, already have scripting language capabilities (using Lua and Jx9 respectively), without relying on POESIE.

```
/* create a POESIE client */
poesie_client_t pcl;
poesie_client_init(mid, &pcl);
/* create a POESIE provider handle */
poesie_provider_handle_t pph;
ret = poesie_provider_handle_create(
    pcl, svr_addr, provider_id, &pph);
/* get the id of the vm */
poesie_vm_id_t vm_id;
poesie_lang_t  lang;
ret = poesie_get_vm_info(
    pph, vm_name, &vm_id, &lang);
/* executing something */
const char* code =
    "print(\"Hello World from Lua VM \" .. "
    "__name__); return \"Bonjour\"";
char* output = NULL;
ret = poesie_execute(
   pph, vm_id, POESIE_LANG_DEFAULT, code, &output);
```

Hands-on Session 3

Using Bedrock and composing microservices

Instructions and objectives

Objectives

- Write a bedrock module to bootstrap your “phonebook” microservice
- Compose your microservice with the Yokan key/value microservice

Instructions

- Follow the instructions in the Exercise 3 section of you chosen language/library

Note

- You will reuse the code from Exercise 2
- You don’t need to have completed Exercise 2, but you need the code to build correctly

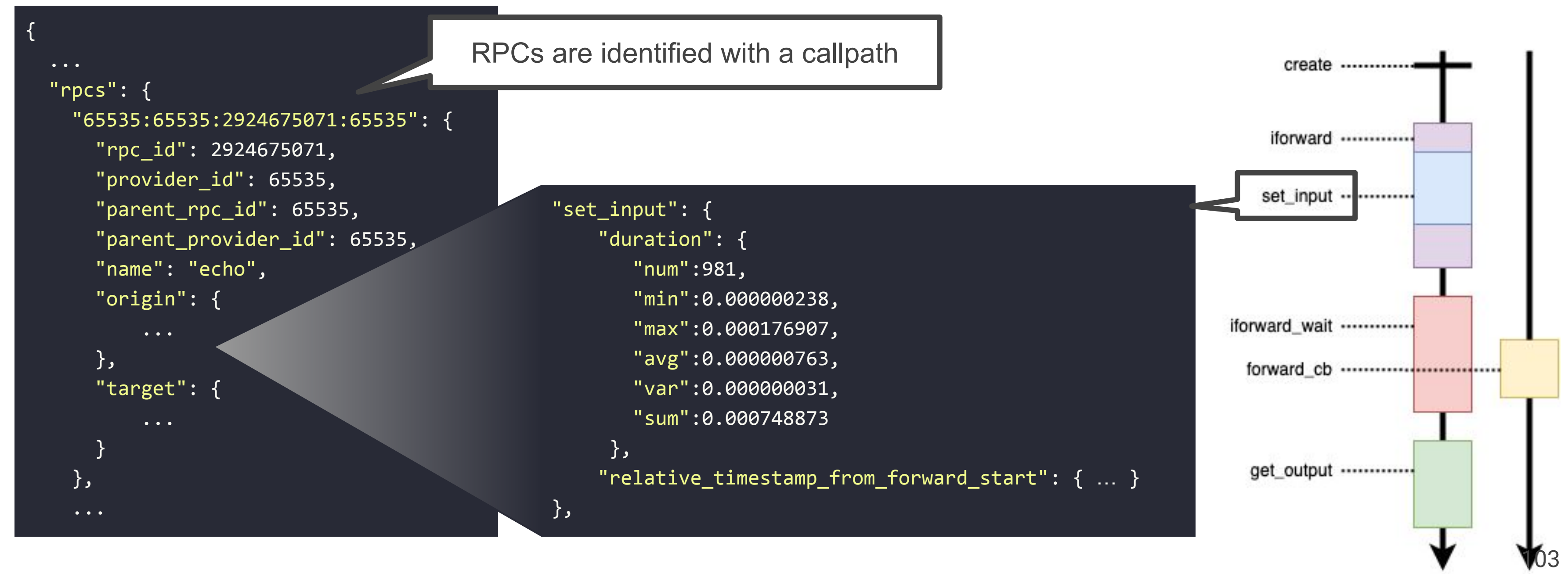
Advanced features

You aren't tired, are you?

Phil Carns

Performance statistics

- Set the `MARGO_ENABLE_MONITORING` environment variable before running your code
- At exit, your applications/services will produce JSON files with statistics
- See https://github.com/mochi-hpc/mochi-performance-analysis to process them with Pandas



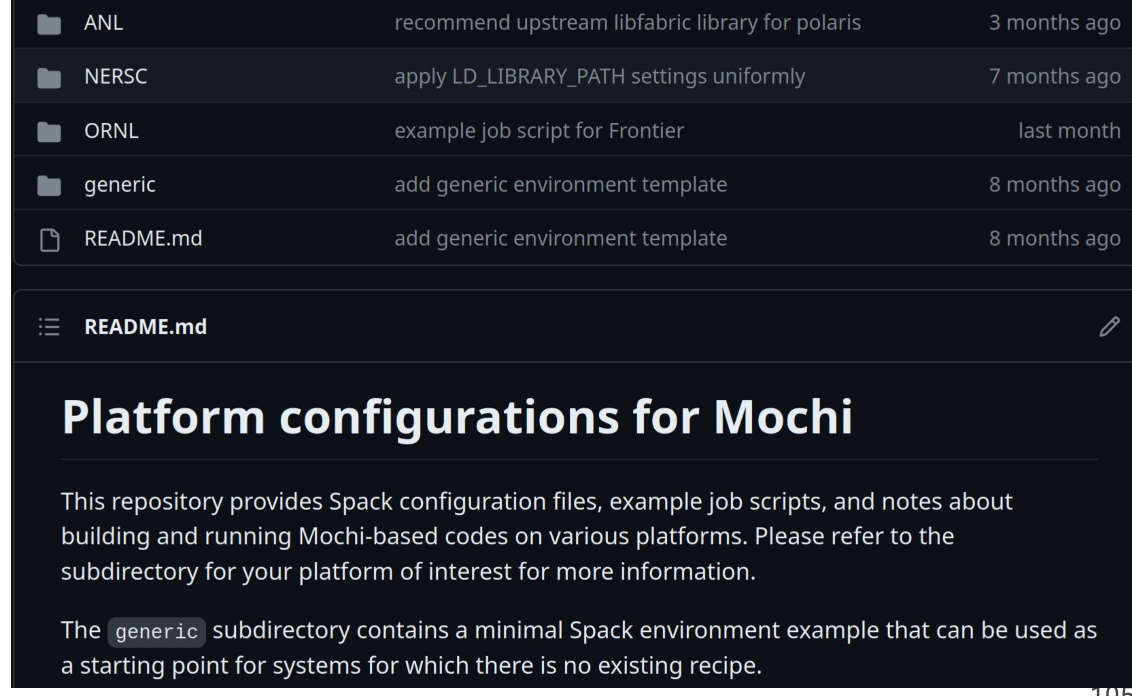
Translating tutorial skills to real systems

- The Mochi tutorial docker container was preconfigured with the following so that you could focus on learning about data service development with Mochi:
  - Development tools
  - Spack
  - Network configuration
- There is a little bit of preliminary configuration work needed to deploy Mochi on a new system
- Spack isn't strictly required, but it is highly recommended for Mochi
- Spack and network configuration go hand-in-hand
  - Spack is able to build nearly anything you need from scratch, but on many platforms it is crucial to use network libraries that are already in place. Examples:
    - Proprietary network stacks (like HPE's Slingshot)
    - Customized network stacks (like Nvidia's OFED Infiniband library with GPU support)
    - Interoperability with MPI (avoiding conflicts/mismatches when MPI and Mochi are used at the same time)
- "**Hello Mochi**" is a step-by-step resource for the process of configuring Mochi on a new machine, but we will highly some elements in the next few slides: https://mochi.readthedocs.io/en/latest/hello-mochi.html

Mochi Spack recipes

https://github.com/mochi-hpc-experiments/platform-configurations

- Check our platform-configurations repository for example configurations.
- We would love to have some new contributions!
- The “generic” recipe has a skeleton Spack configuration if you can’t find a good match.
- Just uncomment the parts you need.
- Each system subdirectory also includes an example job script.



Margo-info

```
####################################################################
# Available Margo (Mercury) network transports on host frontier10465
# - GREEN indicates that it can be initialized successfully.
# - RED indicates that it cannot.
####################################################################

# <address> <transport> <protocol> <results> <example runtime address>

### libfabric tcp provider (TCP/IP) ###
ofi+tcp://    ofi    tcp    YES    ofi+tcp;ofi_rxm://10.128.163.152:37543
### libfabric CXI provider (HPE Cassini/Slingshot 11) ###
ofi+cxi://    ofi    cxi    YES    ofi+cxi://0x24fce400
### integrated sm plugin (shared memory) ###
na+sm://    na    sm    YES    na+sm://30022-0
### TCP/IP protocol, transport not specified ###
tcp://    <any>    tcp    YES    ofi+tcp;ofi_rxm://10.128.163.152:42773
### shared memory protocol, transport not specified ###
sm://    <any>    sm    YES    na+sm://30022-1
### libfabric Verbs provider (InfiniBand or RoCE) ###
ofi+verbs://    ofi    verbs    NO    N/A
### libfabric shm provider (shared memory) ###
ofi+shm://    ofi    shm    NO    N/A
...
```

You've followed a recipe, built everything, but your service won't start. What's wrong?

- The "margo-info" command line utility is installed as part of mochi-margo.
- You can try it in your container now!
- It will probe common network address types and see which our actually working
- The output is extensive, share it with us for help interpreting if you aren't sure why something isn't working.
- **Resist the temptation of tcp even though it (almost) always works; the biggest optimization you can make is to use the fastest native transport for your system.**

Margo-info (continued)

```
######################################################################
# Suggested transport-level diagnostic tools:
# - libfabric:    `fi_info -t FI_EP_RDM`
# - UCX:    `ucx_info -d`
# - verbs:    `ibstat`
# - TCP/IP:    `ifconfig`
# - CXI:    `cxi_stat`
#
######################################################################
# Verbose margo-info information:
# - debug log output:
#   /tmp/margo-info-stderr-0j222h
# - results in JSON format:
#   /tmp/margo-info-json-JsYRbi
#
######################################################################
# List of dynamic libraries used by the margo-info utility:
# - mochi-margo-0.13.1-hcjlhribc345pvtnq5bl2tdwc6xpzqmn/lib/libmargo.so.0
# - mercury-master-r2f7brn7fv3ftnkmlfk65qiug3avynhx/lib/libmercury.so.2
# - /opt/cray/libfabric/1.15.2.0/lib64/libfabric.so.1
# - /usr/lib64/libcxi.so.1
#
# Note: the above list was filtered display only those libraries likely related
#      to communication. Run margo-info with -l to display all libraries.
#
######################################################################
```

Excerpts from margo-info output show some other possible clues:

- Other command line diagnostic tools that are available depending on your desired transport
- Verbose debug log output
  - Most useful if you run margo-info with a specific protocol, like “verbs://” as an argument to limit the volume
- A list of libraries being used
  - A large portion of setup problems result from Mochi using the wrong library for a given network.

Getting Mochi software updates

- We’ve tried to make the Mochi software bug free ;-)
- But occasionally we overlook something. More frequently, we add new features!
- The best way to get updates is to refresh your copy of the mochi-spack-packages git repository.
- You can try it in your docker container, using your preferred git update method:

```
mochi@mt1:~$ cd mochi-spack-packages
mochi@mt1:~/mochi-spack-packages$ git fetch --all
Fetching origin
mochi@mt1:~/mochi-spack-packages$ git rebase
origin/main
Current branch main is up to date.
```

- That will not alter any existing environments.
- If you really want a new version of something, you will need to uninstall the relevant packages and install them again.
    - “spack uninstall –dependents <package>” is often helpful, but you may need to remove it from an environment (or remove the environment and recreate) first.

Conclusion

Phil Carns

10 min

Thank you for your participation!

We appreciate everyone spending time with us today, and we hope that you learned something new about building custom data services with Mochi!

- Were there things that you especially liked or disliked about the tutorial?
- **Please send email us or contact us on Slack to let us know!**
- We would like to improve the content for future tutorials.

Staying in touch

- Feel free to continue working on tutorial examples after this event and *continue to ask questions*.
    - We will keep the #mochi-tutorial-isc-2023 channel open on the Mochi slack space (see tutorial materials or Mochi web site for link)
    - You are also welcome to post more general questions on #general or on the mochi-devel mailing list.
- If you continue to work with Mochi, then we encourage you to attend the Mochi quarterly meetings, announced on the mailing list and slack, e.g.
    - https://www.mcs.anl.gov/research/projects/mochi/2023/04/21/quarterly-meeting-and-newsletter-april-2023/
    - Propose topics, questions, or things you would like to share for the agenda!
- We are happy to list publications and projects using Mochi on our web site.

Notable related events at ISC 2023

- Panel: The Future of Open-Source Filesystems for HPC – Competition, Cooperation or Consolidation?
  - Monday, May 22, 2023 5:20 PM to 6:35 PM · 1 hr. 15 min. (Europe/Berlin) - Hall Z - 3rd Floor
- BoF: A European I/O Trace Repository to Build Better I/O Systems & HPC Applications
  - Tuesday, May 23, 2023 4:15 PM to 5:15 PM · 1 hr. (Europe/Berlin) - Hall G1 - 2nd Floor
- Workshop: HPC I/O in the Data Center
  - Thursday, May 25, 2023 9:00 AM to 6:00 PM · 9 hr. (Europe/Berlin) - Hall Y2 - 2nd Floor
- Panel: 7th International Workshop on In Situ Visualization (WOIV)
  - Thursday, May 25, 2023, 9:00 AM to 1:00 PM (Europe/Berlin) - Hall Y3 - 2nd Floor
- Workshop: 2nd International Workshop on Malleability Techniques Applications in High-Performance Computing
  - Thursday, May 25, 2023 2:00 PM to 6:00 PM · 4 hr. (Europe/Berlin) - Hall Y5 - 2nd Floor

THIS WORK WAS SUPPORTED BY THE U.S. DEPARTMENT OF ENERGY, OFFICE OF SCIENCE, ADVANCED SCIENTIFIC COMPUTING RESEARCH, UNDER CONTRACT DE-AC02-06CH11357.