Examples of Mochi services

HEPnOS, FlameStore, and Mobject

Mochi Bootcamp
September 24-26, 2019

Argonne NATIONAL LABORATORY

Methodology for designing a data service

User Requirements

Data model
Access pattern
Guaranties

Service Requirements

Data organization
Metadata organization
User interface

Composition and Interfacing

Composition glue code
API implementation

Building Blocks

Runtime
Service providers

Identifying application needs

User Requirements

Which data model?

- Arrays, meshes, objects…
- Namespace, metadata

Which access pattern?

- Characteristics (e.g. access sizes)
- Collective/individual accesses

Which guarantees?

- Consistency
- Performance
- Persistence

Identifying application needs

Service Requirements

Which data model?

- Arrays, meshes, objects…
- Namespace, metadata

Which access pattern?

- Characteristics (e.g. access sizes)
- Collective/individual accesses

Which guarantees?

- Consistency
- Performance
- Persistence

- How should data be organized?
  - Sharding
  - Distribution
  - Replication
- How should metadata be organized?
  - Distribution
  - Content
  - Indexing
  -
- How do clients interface with the service?
  - Programming language
  - API

HEPnOS: A Storage Service for High Energy Physics Applications

Storing “Products”

```
class Collision {
    double energy;
    std::array<double,3> position;
    ...
};
```

User Requirements

Data Model

- Many instances of small C++ objects
- Hierarchy of datasets, runs, subruns, and events
- Products accessible by “tag”

Access Pattern

- Write-once, read-many
- Products accessed atomically
- Access by “tag” and by type
- Iterators to navigate the hierarchy

Envisioned usage

- Long-running (weeks), resizable cache based on fast, in-compute-node storage (SSDs, NVRAM, local memory)
- Accessed by multiple applications concurrently
- Backed-up by a more permanent storage system (parallel file system, archive system, object store) when undeployed

Figuring out service requirements

Interface

- C++ interface with integrated serialization
- Need for batching features and non-blocking transfers

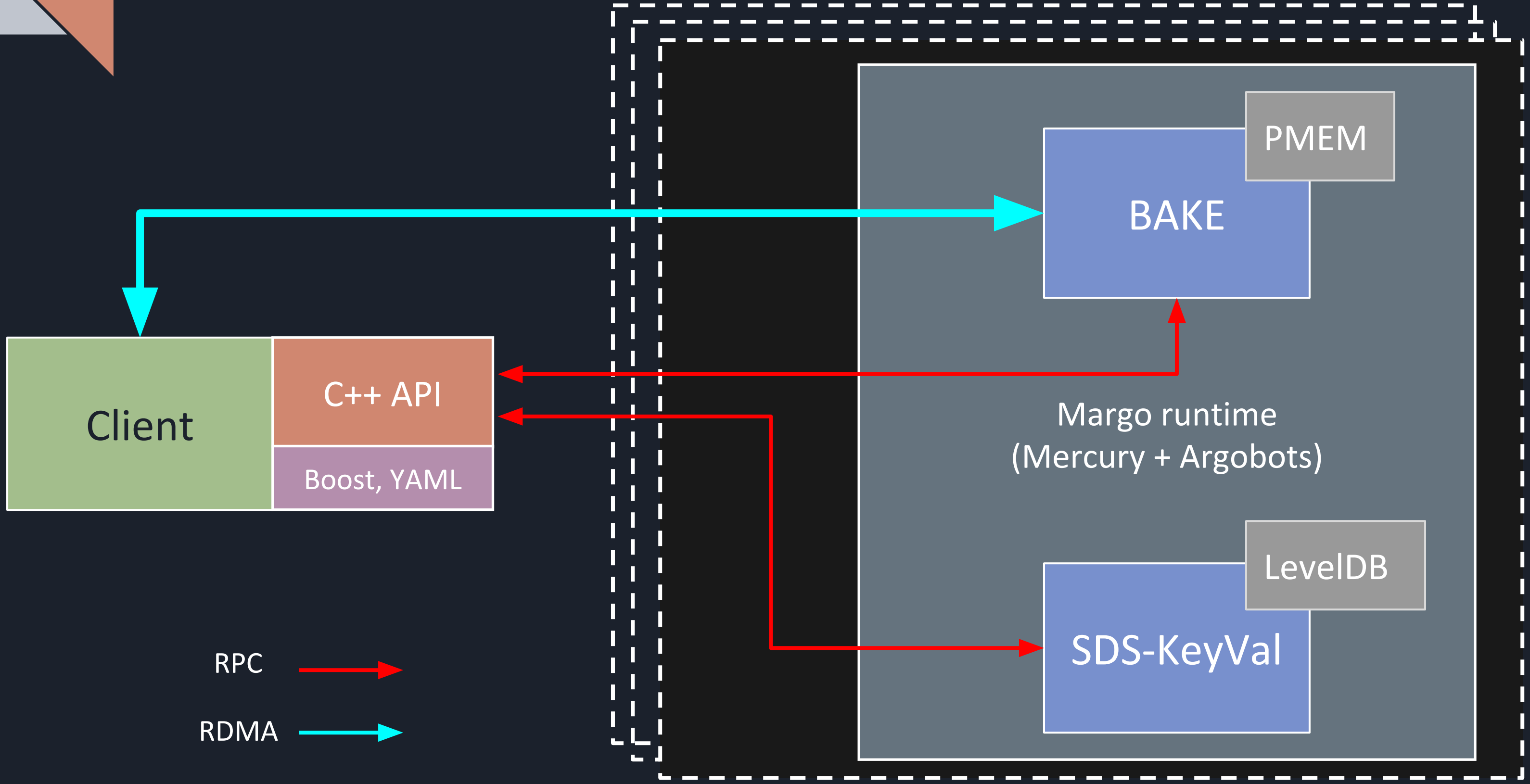
Backend

- Distributed key/value store
- Objects small enough not to be sharded
- No replication needed
- No overwrite allowed

Organization

- Path-like namespace
- Hashing function mapping paths to target

Service Requirements


PMEM
BAKE
Client
C++ API
Boost, YAML
Margo runtime
(Mercury + Argobots)
LevelDB
SDS-KeyVal
RPC
RDMA

Example of HEPnOS's interface

```
// initialize a handle to the HEPnOS datastore
hepnos::DataStore datastore( "config.yaml" );
// access a nested dataset
hepnos::DataSet ds = datastore[ "path/to/dataset" ];
hepnos::Run run = ds[43]; // access run 43 in the dataset
hepnos::SubRun subrun = run[56]; // access subrun 56
hepnos::Event ev = subrun[25]; // access event 25
// iterate over the subruns in a run
// using a C++ range-based for
for(auto& subrun : run) { ... }
```

- Map-like access to the hierarchy of datasets, runs, subruns, and events
- Iterators to navigate the hierarchy

Example of HEPnOS's interface

```
struct Particle {
      float x, y, z; // member variables
      // serialization function for boost to use
      template<typename A>
      void serialize(A& a, unsigned long version) {
            ar & x & y & z;
      }
};
...
hepnos::Event ev = subrun[25]; // access event 25
// store data (an std::vector of Particle)
st::vector<Particle> vp1 = ...;
ev.store("mylabel", vp1);
// load data
std::vector<Particle> vp2;
sv.load("mylabel", vp2);
```

- Serialization based on Boost
- Load/Store functions

FlameStore: a Storage Service for Deep Learning Workflows

Storing neural networks

Data Model

- Large weight matrices
- Neural network architecture

Access Pattern

- Not many neural networks
- Writes, updates, and reads
- Neural networks accessed atomically
- Access by name within a flat namespace
- The application is not I/O bound

User Requirements

Envisioned usage

- Workflow running for a few hours to a few days
- Storage system spanning the workflow allocation
- Backed-up by a more permanent storage system (parallel file system, archive system, object store) when undeployed

Figuring out service requirements

Interface

- Python interface easy to use with Keras
- Need for efficient access to tensors memory

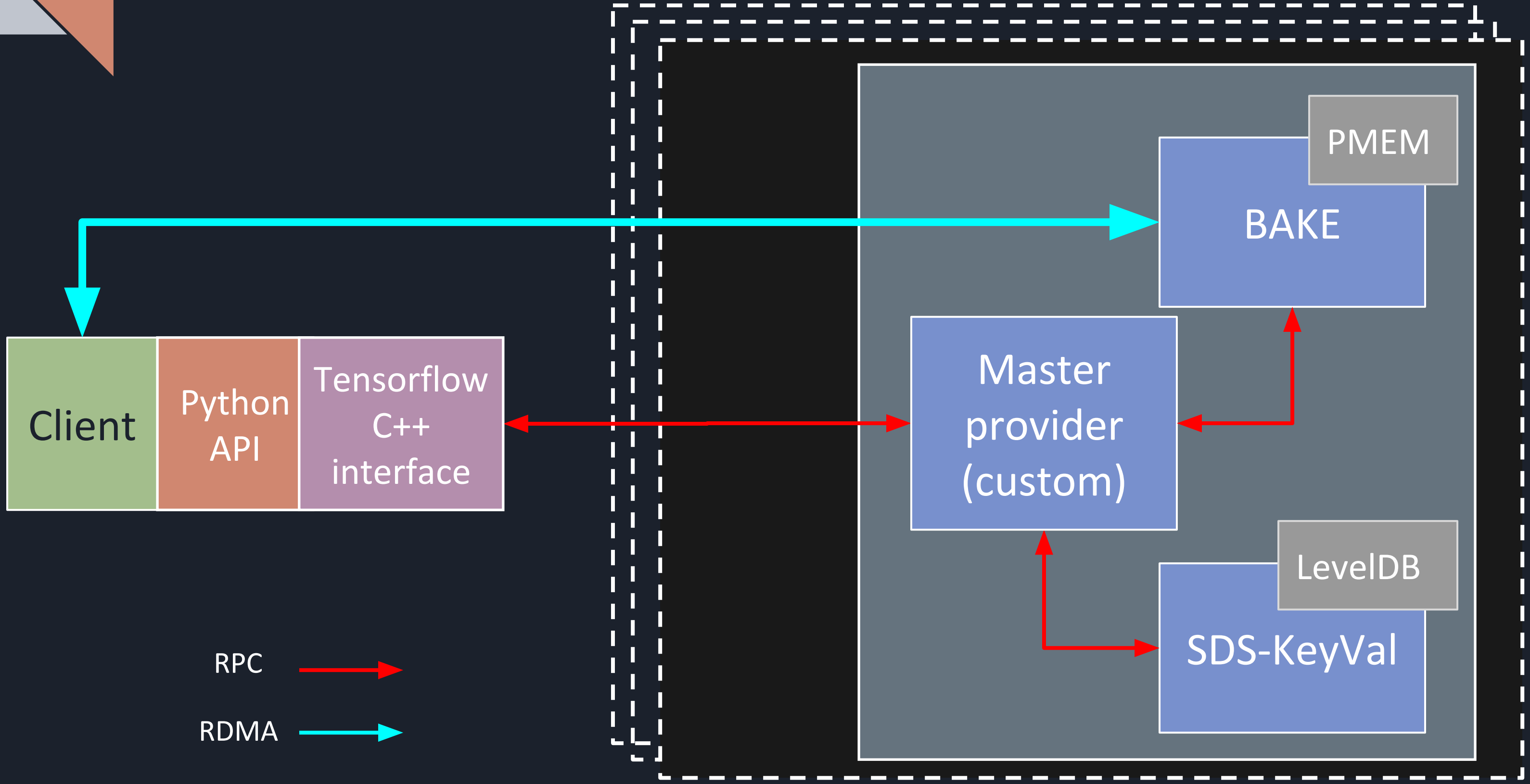
Backend

- Single key/value store for metadata
- Distributed blob storage w/ efficient bulk transfers
- No replication needed
- Overwrite allowed

Organization

- Flat namespace
- Hashing function mapping name to target

Service Requirements



PMEM
BAKE
Client
Python API
Tensorflow C++ interface
Master provider (custom)
LevelDB
SDS-KeyVal
RPC
RDMA

Example of FlameStore's interface

```
with Engine('tcp', use_progress_thread=True, mode=pymargo.client) as engine:
    client = Client(engine)
    provider = client.lookup(provider_addr, provider_id)
    ...
    callback = RemoteCheckpointCallback(model_name='MyModel',
                  client=client, provider_addr=provider_addr,
                   provider_id=provider_id))
    model = ...
    model.fit(... callbacks=[callback])
    ...
    model = client.load_model(provider, 'MyModel')
    optimizer = client.load_optimizer(provider, 'MyModel')
```

- Integrates with Keras code through the callback interface
- Enables checkpointing the optimizer in addition to weights
- Records the model's architecture using JSON

Mobject: an Object Storage Service tailored for HPC

Storing objects for HPC

User Requirements

Data Model

- Mimic the RADOS data model
- Objects are byte arrays identified by unique names
- Objects can be written incrementally by many processes
- Associate key/val attributes with objects

Access Pattern

- Objects will be accessed/updated concurrently, by potentially many processes
- Object accesses tend to be large & aligned, but small strided accesses critical to performance
- Accessed directly by apps and by middleware systems
- Expect a mix of read and write

Envisioned usage

- In-system object store deployed alongside application(s) as primary I/O provider
    - Dedicated nodes or co-located with app nodes

- Typically backed by persistent memory devices, but can also offer in-memory object storage
    - Similar to prior examples, mechanisms exist to migrate to more appropriate levels in the storage hierarchy

Figuring out service requirements

Interface

- C interface implementing a subset of the RADOS API
- Simple POSIX-like API for accessing object extents

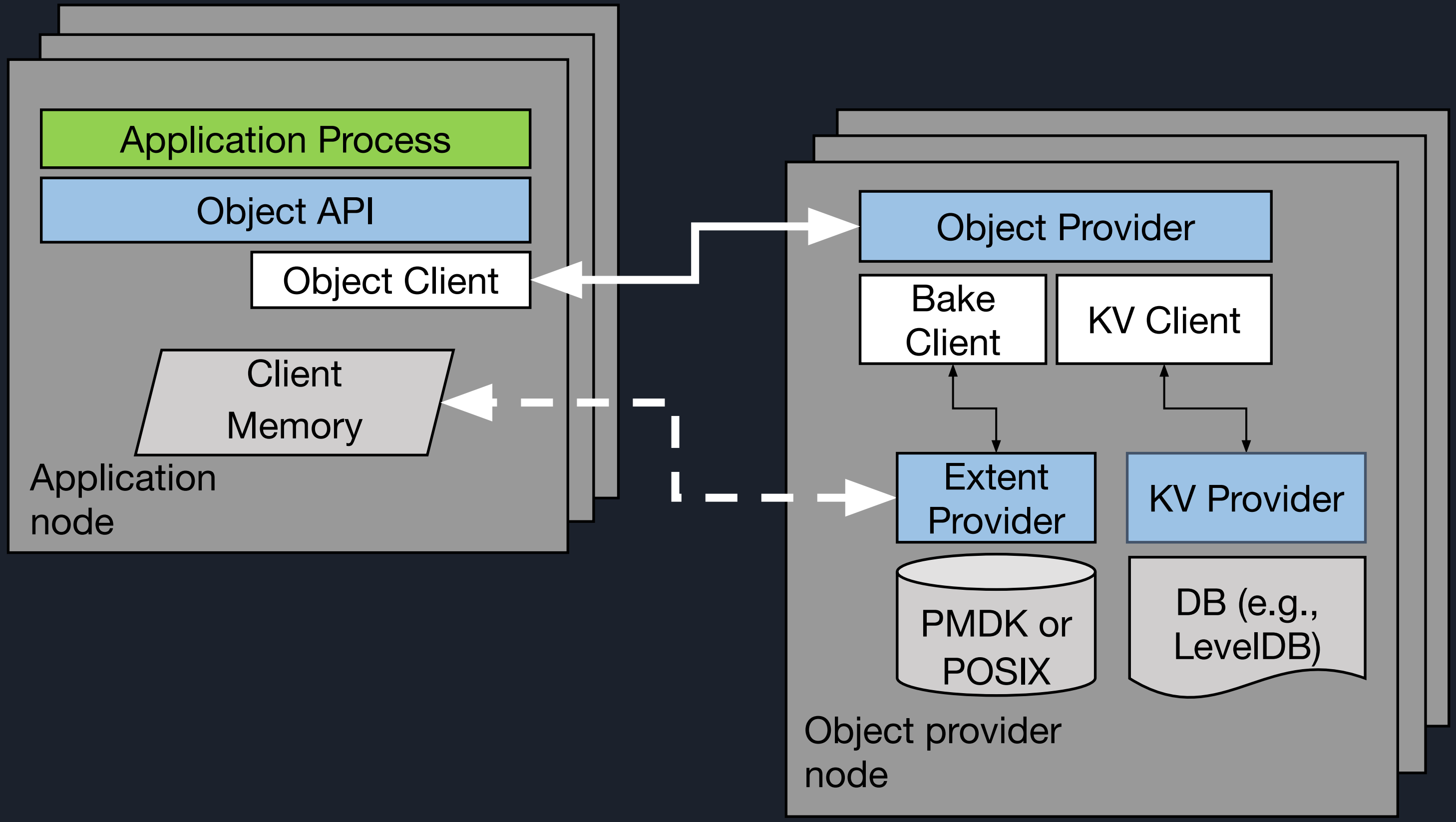
Backend

- Distributed blob storage w/ efficient bulk transfers
- Distributed metadata w/ a kv per blob store
- Log-structured object storage abstraction
- No replication

Organization

- Flat namespace
- Hashing function mapping name to target

Service Requirements


Application Process
Object API
Object Client
Client Memory
Application node
Object Provider
Bake Client
KV Client
Extent Provider
KV Provider
PMDK or POSIX
DB (e.g., LevelDB)
Object provider node

Example of Mobject's interface

```
{
rados_write_op_t create_op;

rados_op = rados_create_write_op();
rados_write_op_create(rados_op, LIBRADOS_CREATE_IDEMPOTENT, NULL);
rados_write_op_write(write_op, (const char *)buffer, length, offset);
... /* more operations */
ret = rados_write_op_operate(rados_op, rados_ioctx, object_name, NULL, 0);
rados_release_write_op(rados_op);
if (ret)
        fprintf(stderr, "unable to create RADOS object", ret);
}
```

- Simple, POSIX-like create/read/write interface for accessing object data
- Implements the RADOS write_op and read_op interfaces, allowing clients to submit lists of I/O operations for a given object

Mobject performance results

- Using IOR (with RADOS backend) as a driver, compare a couple of different Mobject deployments against GPFS on Cooley system @ Argonne
    - Kove devices are network-attached persistent memory devices
    - tmpfs deployment is directly to RAM (not persistent)