Buffer Organizer

The **Buffer Organizer** is the "corrector" half of our predictor/corrector model. It attempts to correct sub-optimal DPE placements by moving data among buffers.

Contents

Objectives

Operations

- Transfer
  - Who can initiate Transfer tasks?
- Evict(size_t bytes, Targets[])
  - Who can initiate Eviction tasks?
  - Who translates an Eviction into a series of Transfers?

Swap Target

- PFS Swap Target Assumptions
  - Single shared file pros
  - File per rank pros
  - File per node
  - Do we go through the BPM? (No)

Triggers

- Periodic
- Client-triggered
- System-triggered
- Requirements for Queue implementation

Design Details

- BO RPC Server
- Buffer Organizer
  - Work Queues
  - Schedulers
  - Threads

Example Flows

- Hot Put

Objectives

- Management of hierarchical buffering space
  - Data flushing
  - Read acceleration
- Manage data life cycle, or journey
  - When is the blob in equilibrium?
  - How do we eliminate unnecessary data movement?

Operations

All `BufferOrganizer` operations are implemented in terms of 3 simple operators

- MOVE(BufferID, TargetID)
- COPY(BufferID, TargetID)
- DELETE(BufferID)

With these operators, we can build more complex tasks:

Transfer

Move a `BufferID` from one set of `Target`s to another.

Who can initiate Transfer tasks?

- The System (load balancing)
- The User (producer/consumer)

Evict(size_t bytes, Targets[])

Move a set of `BufferId`s from one set of `Target`s to an unspecified location (could even be swap space).

Who can initiate Eviction tasks?

- Put (DPE)
- Get (Prefetcher)
- Thread that updates the `SystemViewState` (enforces a minimum capacity threshold passed in through the config).

Who translates an Eviction into a series of Transfers?

- DPE?
- BO?

Swap Target

When `GetBuffers` fails (because constraints can't be met or we are out of buffering space), we send blobs to **Swap Space**. We reserve a special **Buffering Target** for this purpose called the **Swap Target**. This special target is never considered by a DPE as a buffering target. It is only meant as a "dumping ground" for blobs that don't fit in our buffering space. It will usually be backed by a parallel file system, but could also be backed by AWS, or any other storage. From an API perspective, a blob in swap space is no different from a blob elsewhere in the hierarchy. You can **Get** it, ask for its metadata, **Delete** it, etc.

PFS Swap Target Assumptions

- For now we'll assume that the swap target is backed by a parallel file system.
- We'll keep one swap file per node, assuming we stick with one buffer organizer per node.

Single shared file pros

- Could theoretically reap performance benefits of collective IO operations, although I don't think we'll ever be able to capitalize on this because each rank must act independently and can't synchronize with the other ranks.
- Less stress on the PFS metadata server.

File per rank pros

- Don't have to worry about reserving size for each rank.
- Don't have to worry about locking.

File per node

- We'll go with this for the initial implementation.

- Don't have to worry about locking or reserving size with respect to the buffer organizer. However, since multiple ranks could potentially write to the same swap file, we need to either
  - Filter all swap traffic through the buffer organizer
  - Synchronize all access to the file
- Won't overload the metadata servers as bad as file per rank.

Do we go through the BPM? (No)

- + Can reuse a lot of code paths.
- - Have to decide sizes ahead of time.
- - Cuts into our RAM.
- - Might run out of buffers.

Triggers

The Buffer Organizer can be triggered in 3 ways:

Periodic

The period can be controlled by a configuration variable.

Client-triggered

- If, for any reason, a client DPE places data to the swap target, it will also trigger the buffer organizer by adding an event to the buffer organizer's queue.
- We store the blob name, the offset into the swap target (for file-based targets), and the blob size.
- When the buffer organizer processes an event, it

1. Reads the blob from the swap target into memory.
2. Calls `Put` to place the blob into the hierarchy. If the `Put` fails, it tries again, up to `num_buffer_organizer_retries` (configurable) times.

System-triggered

- Nothing is implemented yet.
- Should the BO constantly monitor the buffering hierarchy and attempt to maintain a set of rules (remaining capacity percentage, thresholds, etc.)?
- Should the BO simply carry out "orders" and not attempt to make its own decisions? If so, who gives the orders?
- Should the BO be available for other asynchronous tasks?

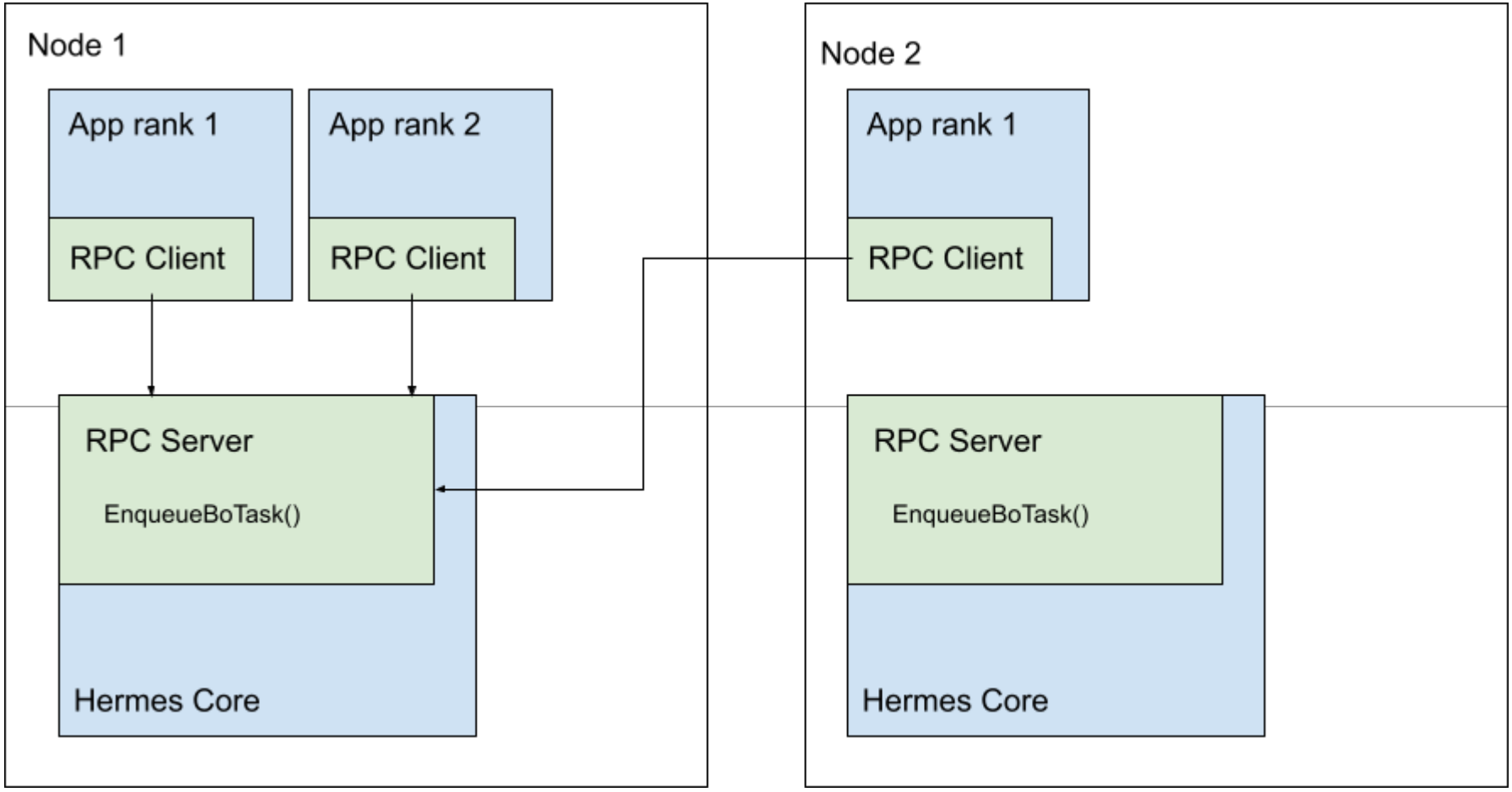
Requirements for Queue implementation

- (At least) 2 different priority lanes
- Node local and remote queues (but only for neighborhoods, not global queues).
- Need ability to restrict queue length

Design Details

BO RPC Server

- RPC is used to route `BoTasks` to the appropriate Hermes core.
- The `BO` RPC server only has one function: `bool EnqueueBoTask(BoTask task, Priority priority);`



Buffer Organizer

Work Queues

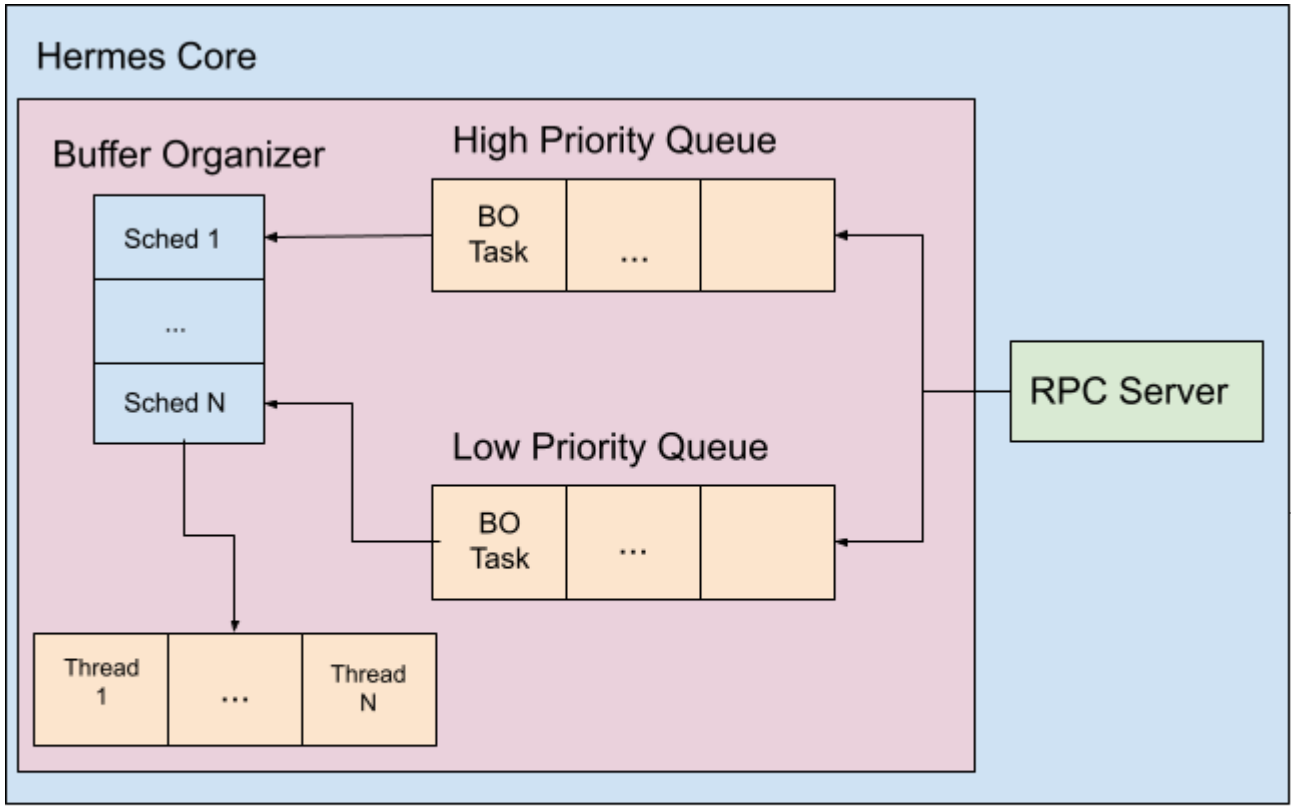
- Argobots **pool**s
- High and low priorities
- Basic FIFO queue by default
- Completely customizable (e.g., could be a priority queue, min-heap, etc.)

Schedulers

- Argobots **scheduler**s
- Takes tasks from the queues and runs them on OS threads as user level threads (basically coroutines).
- Completely customizable.
- By default, one scheduler is associated with a single execution stream (OS thread).
- Only take tasks from low priority queue if high priority queue is empty?

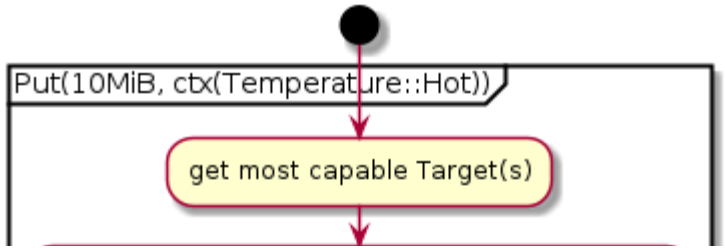
Threads

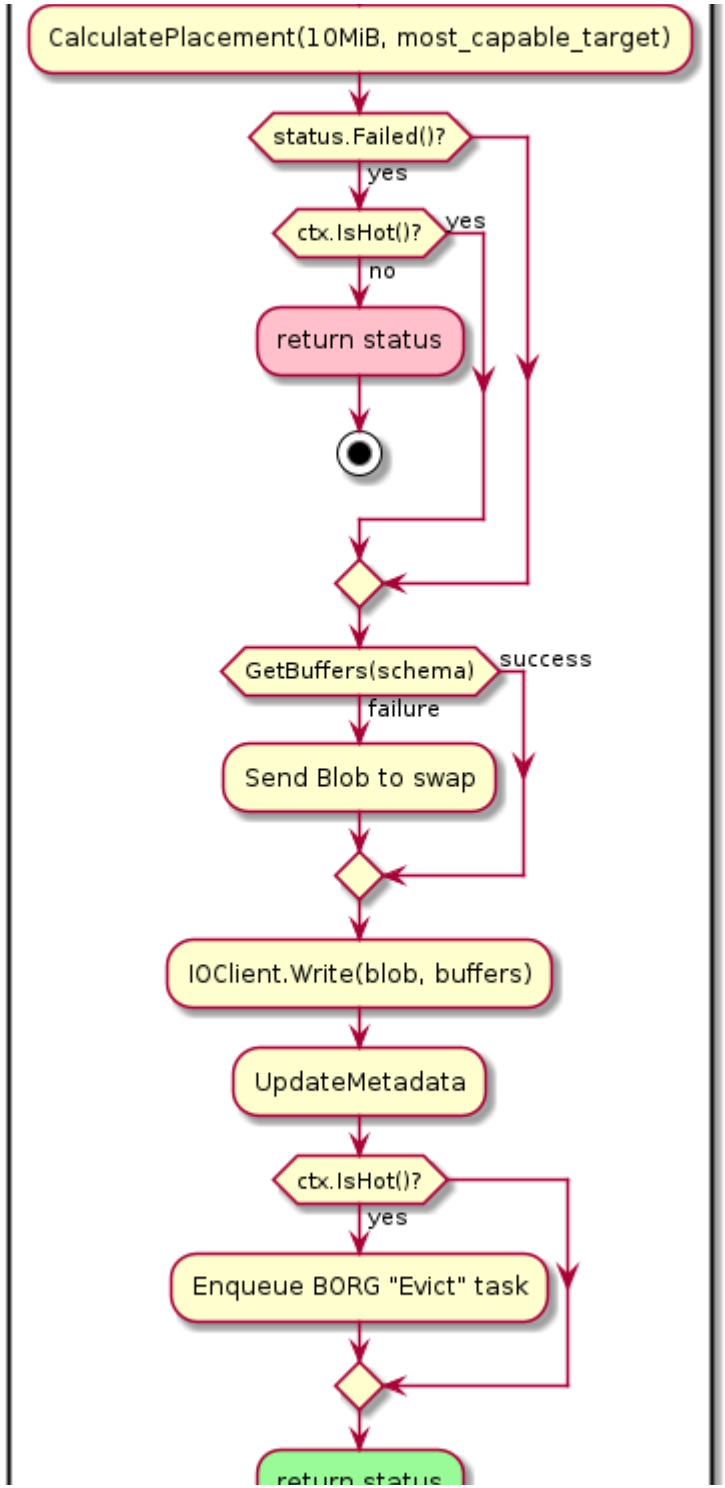
- Argobots **execution stream**s
- Bound to a **processing element** (CPU core or hyperthread), and shouldn't be oversubscribed.

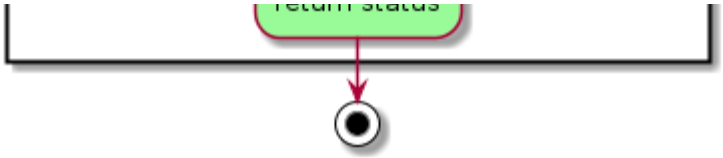


Example Flows

Hot Put




CalculatePlacement(10MiB, most_capable_target)
status.Failed()?
yes
ctx.IsHot()?
yes
no
return status
GetBuffers(schema)
success
failure
Send Blob to swap
IOClient.Write(blob, buffers)
UpdateMetadata
ctx.IsHot()?
yes
Enqueue BORG "Evict" task
return status



BO Eviction Flow