Metadata Manager

Contents

Assumptions and Constraints

- The amount of memory we have for metadata is fixed for each node.
- We required variable-length data structures
  - Lists of `BufferIDs`
  - Strings for `Bucket` and `Blob` names.
- Metadata can be freed in any order, which means we can't use the same linear allocator we used for the `BufferPool`.
- Metadata must be in shared memory so that any process can access it.

- Metadata is fetched from remote nodes via RPC.

Distributed Metadata Approach

If we can represent all user primitives (`Bucket`, `VBucket`, and `Blob`) as IDs, then we can take the same approach to distributing metadata as we do for distributing `BufferID`s, which is to encode a node and an index into the ID. Then, each metadata operation looks like this:

```
if ID is on same node as calling process:
    LocalCall(ID)
else:
    RPC(ID)
```

To achieve the goal of distributing metadata in this way while still making the API easy to use, we introduce two views of user primitives.

User View

User primitives are referred to by unicode strings (names).

System View

User primitives are referred to by unsigned 64 bit integers (IDs).

Each ID encodes the data it needs to access its metadata.

BucketID

Node ID
BucketInfo array index

VBucketID

Node ID
VBucketInfo array index

BlobID

Node ID
Offset into the ID List memory where this `Blob`'s list of `BufferID`s begins.

This requires a mapping from `name -> ID`. Three such maps exist in the metadata shared memory segment

1. Bucket Map
2. VBucket Map
3. Blob Map

This trades space for speed, but we could easily combine all three maps into one if we decide that space efficiency is more important than speed.

Storage Method

Maps and ID Lists

All metadata is distributed among nodes by first hashing the key to determine the node, then hashing again to determine the slot.

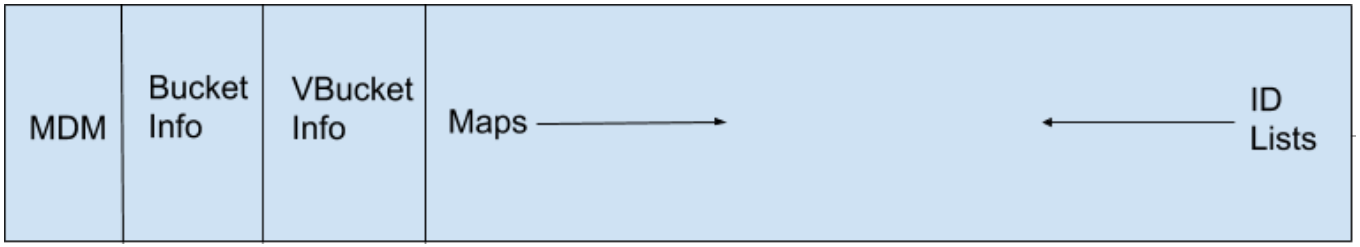
pros

- Better load balancing

cons

- May require extra RPC calls. Initial tests show that this indirection should be avoided. **TODO:** We need to revisit this.

Metadata Shared Memory Segment



MetadataManager (MDM)

Contains all the information needed to interact with the metadata shared memory segment

- offsets
- locks

BucketInfo Array

A configuration parameter `max_buckets_per_node` determines its maximum size.

- next free `Bucket`
- list of `Blobs`
- reference count
- statistics

VBucketInfo Array

A configuration parameter `max_vbuckets_per_node` determines its maximum size.

- next free `VBucket`
- list of `Blobs`
- list of `TraitIDs`
- reference count
- statistics

Maps

The **Bucket Map**, **VBucket Map**, and **Blob Map** are all stored here, and grow towards higher addresses. The `MetadataManager` stores the upper limit to ensure that it doesn't overlap with the **ID List** region. Since each ID can be treated as a `u64`, we only need one map type: `string -> u64`.

ID Lists

This section stores variable-length lists of `BlobIDs` and `BufferIDs`. It grows towards smaller address, and the `MetadataManager` ensures that it does not overlap with the **Maps** section.

Walkthrough of Bucket.Put()

1. Create a new `BlobID`. The ID's node index (top 32 bits) is created by hashing the blob name, and the ID's offset to a list of `BufferIDs` (bottom 32 bits) is allocated from the MDM shared memory segment on the target node.

2. Add the new `BlobID` to the **IdMap**. This could be local, or an RPC.

3. Add the `BlobID` to the `Bucket`'s list of blobs.

Walkthrough of Bucket.Get()

1. Hash the blob name to get the `BlobID`.

2. Get the list of `BufferIDs` from the `BlobID`.

3. Read each `BufferID`'s data into a user buffer.

Limits

- Max Buckets: 2^32 - 1
- Max Buckets Metadata Size: sizeof(BucketInfo) * (2^32 - 1) = 137.4 GiB
- Max VBuckets: 2^32 - 1
- Max VBuckets Metadata Size: sizeof(VBucketInfo) * (2^32 - 1) = 171.8 GiB
- Max Heap size for Maps and IdLists: 4 GiB

- Assuming a block size of 4 KiB, a 1-to-1 mapping of blobs to buffers, and a maximum blob name of 128 bytes, the upper limit of metadata is 4% of the total buffering hierarchy size, or 40 MiB per GiB. (hierarchy_size_in_bytes * (sizeof(BlobID) + 2*sizeof(BufferID) + max_blob_name_size + sizeof(BlobID))) / 4096.

Performance

Objectives

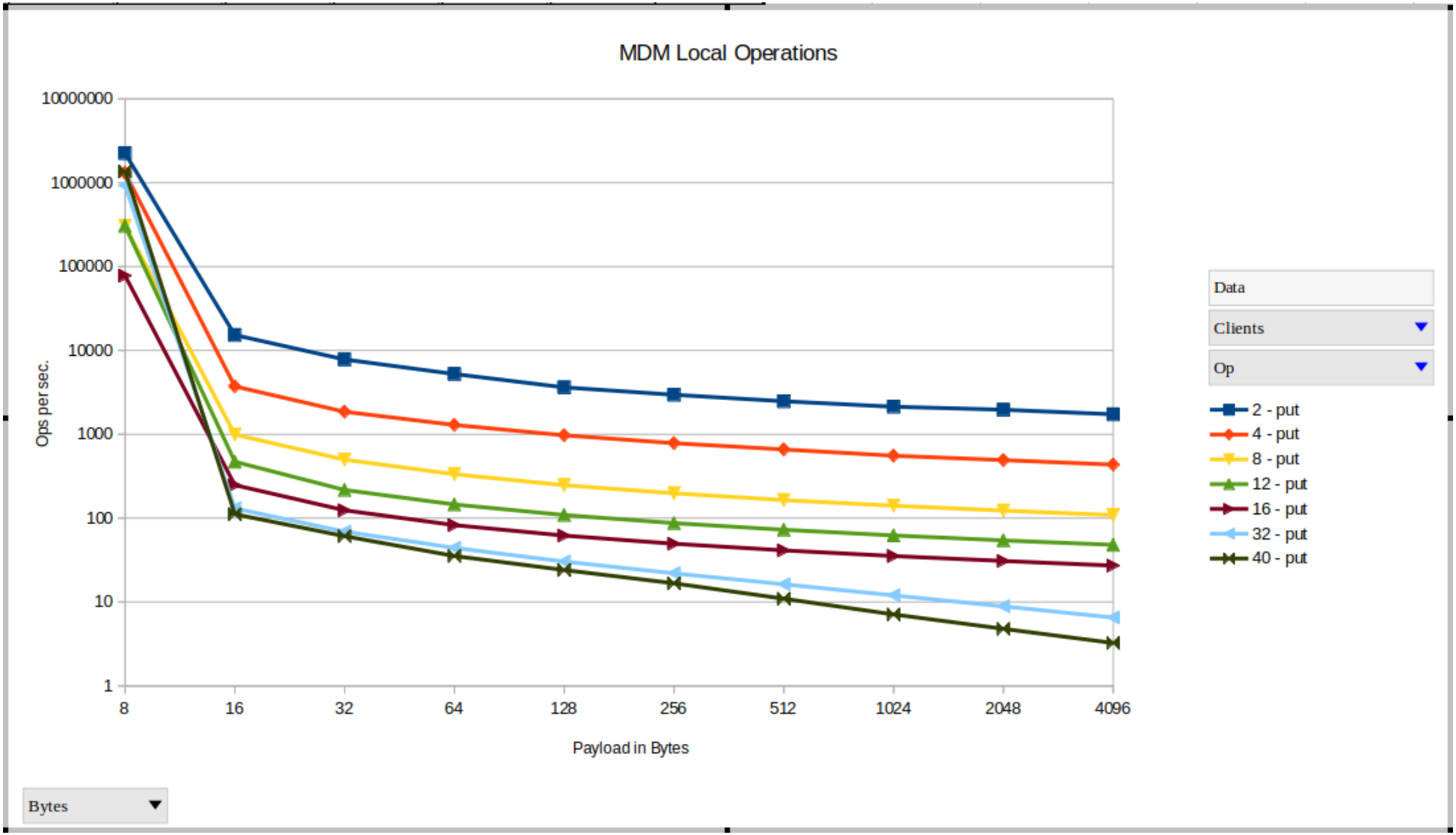
- Scalable
- Load balanced (avoid hot spots)
- Operation concurrency
- Put and Get are equally important
- Low footprint

All performance benchmarks were run on the Ares cluster compute nodes with 40 Gb/sec. ethernet.

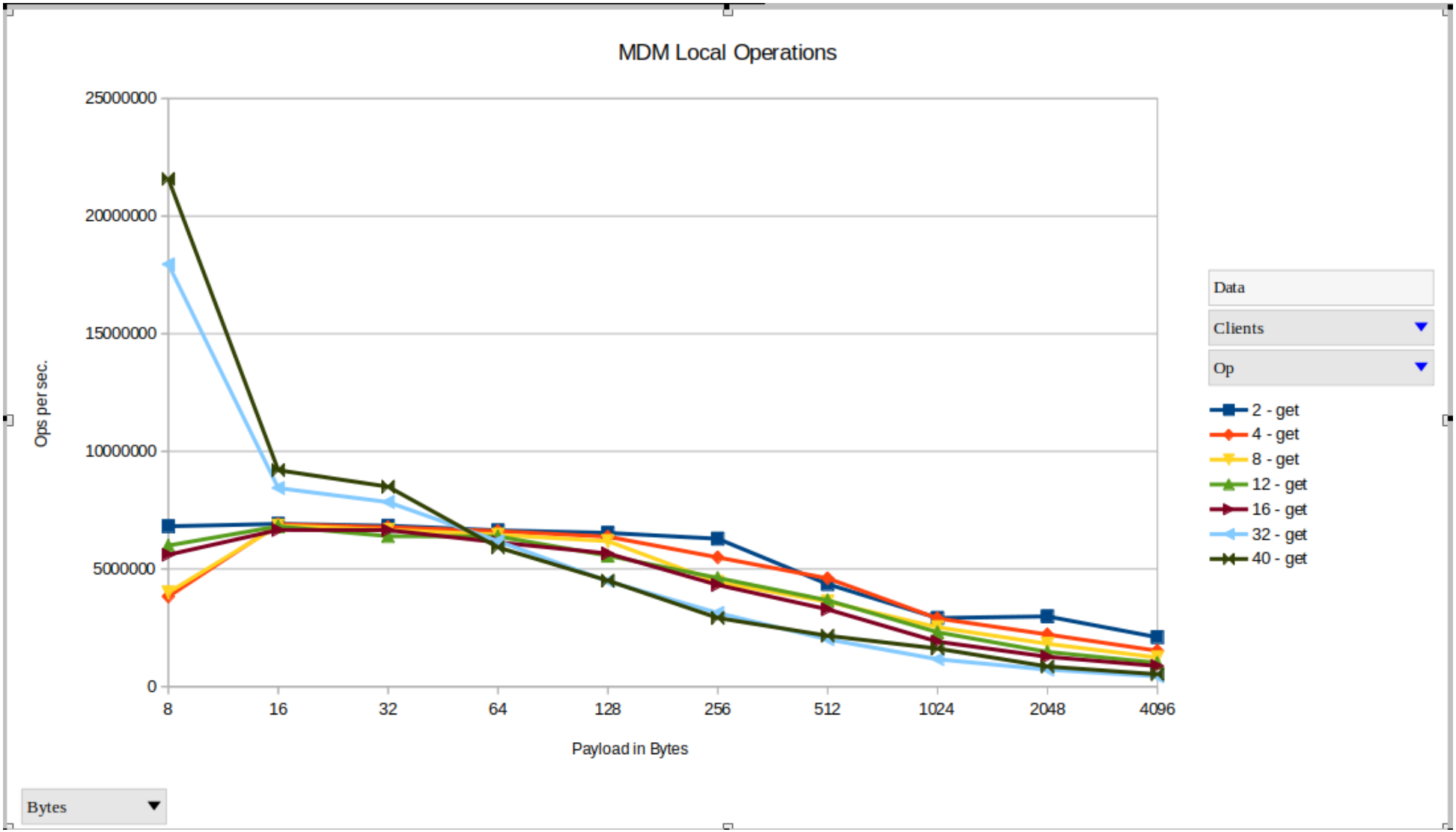
Local Performance

Overall local performance needs some optimization effort, but we are still working on full program correctness. Optimization work will come after we have a solid, tested core library running real workloads.

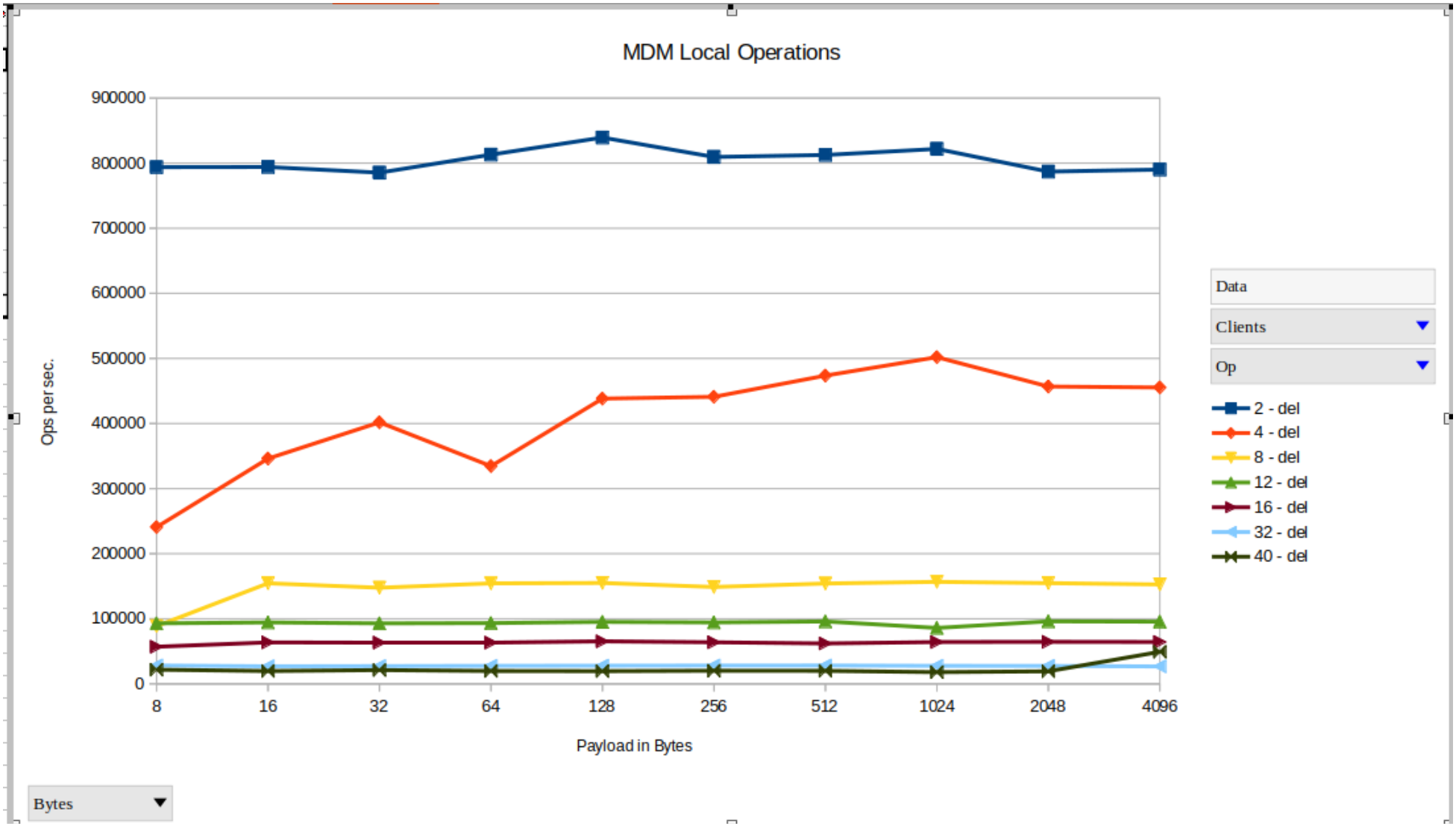
Put performance measures the number of times per second we can insert variable sized lists of 64-bit integers (from 8 bytes to 4 KiB) into the local MDM. Scalability is pretty bad at the moment (notice the log scale in the y axis) because we have not yet optimized the ID list allocator. It's doing a simple linear search to find the first free heap slot that can accommodate the requested size. As more and more blocks are freed, the heap becomes fragmented, which makes the search even worse. We plan to improve the algorithm and perform occasional heap defragmentation by merging adjacent free blocks.



Get performance is much more scalable, since the allocator isn't involved. Larger requests take longer to copy, of course.



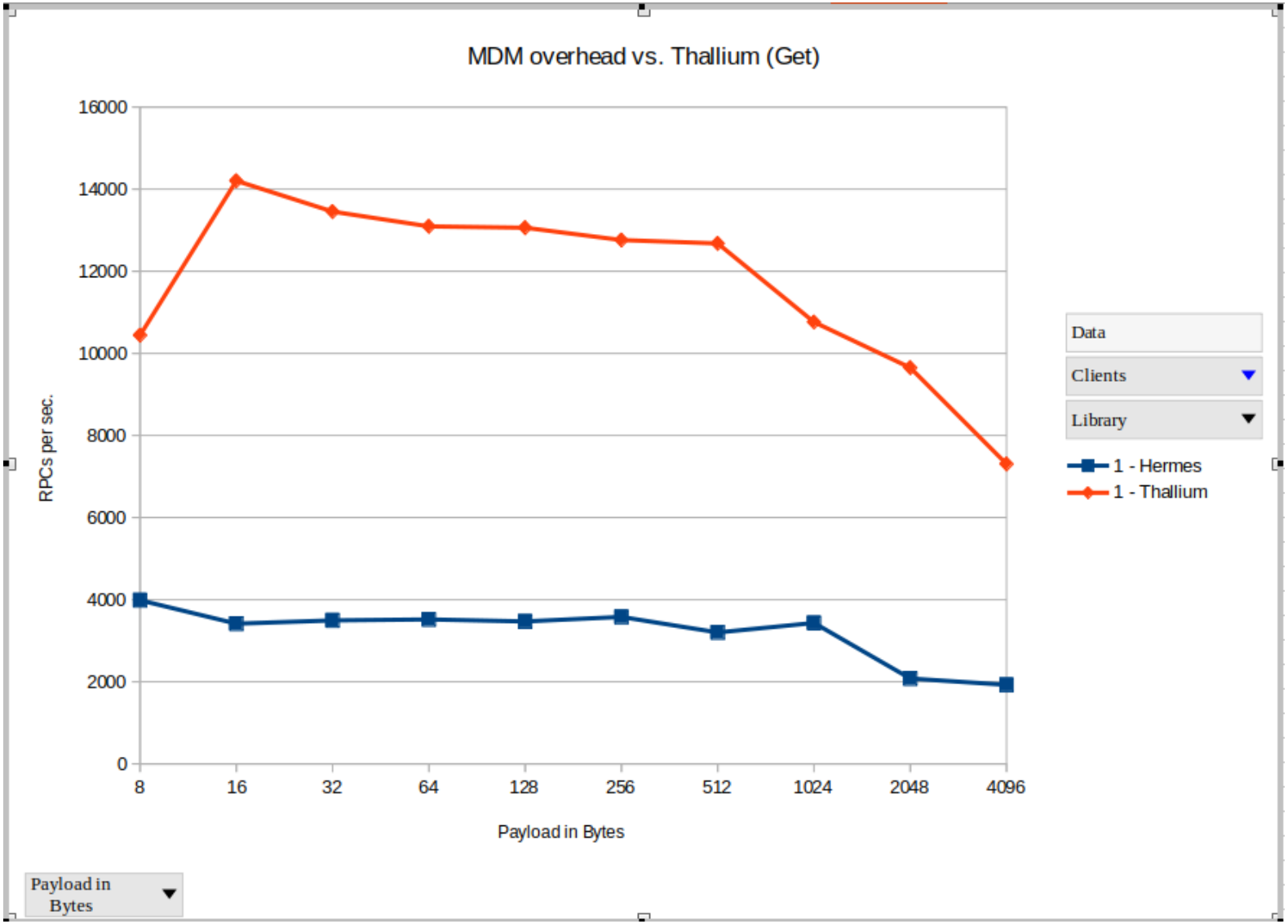
Delete requests don't depend on the payload size, since we just need to invalidate the `BlobID` and free its list of `BufferIDs`. Everything is serialized through a lock though, so some work must be done to improve the scalability.



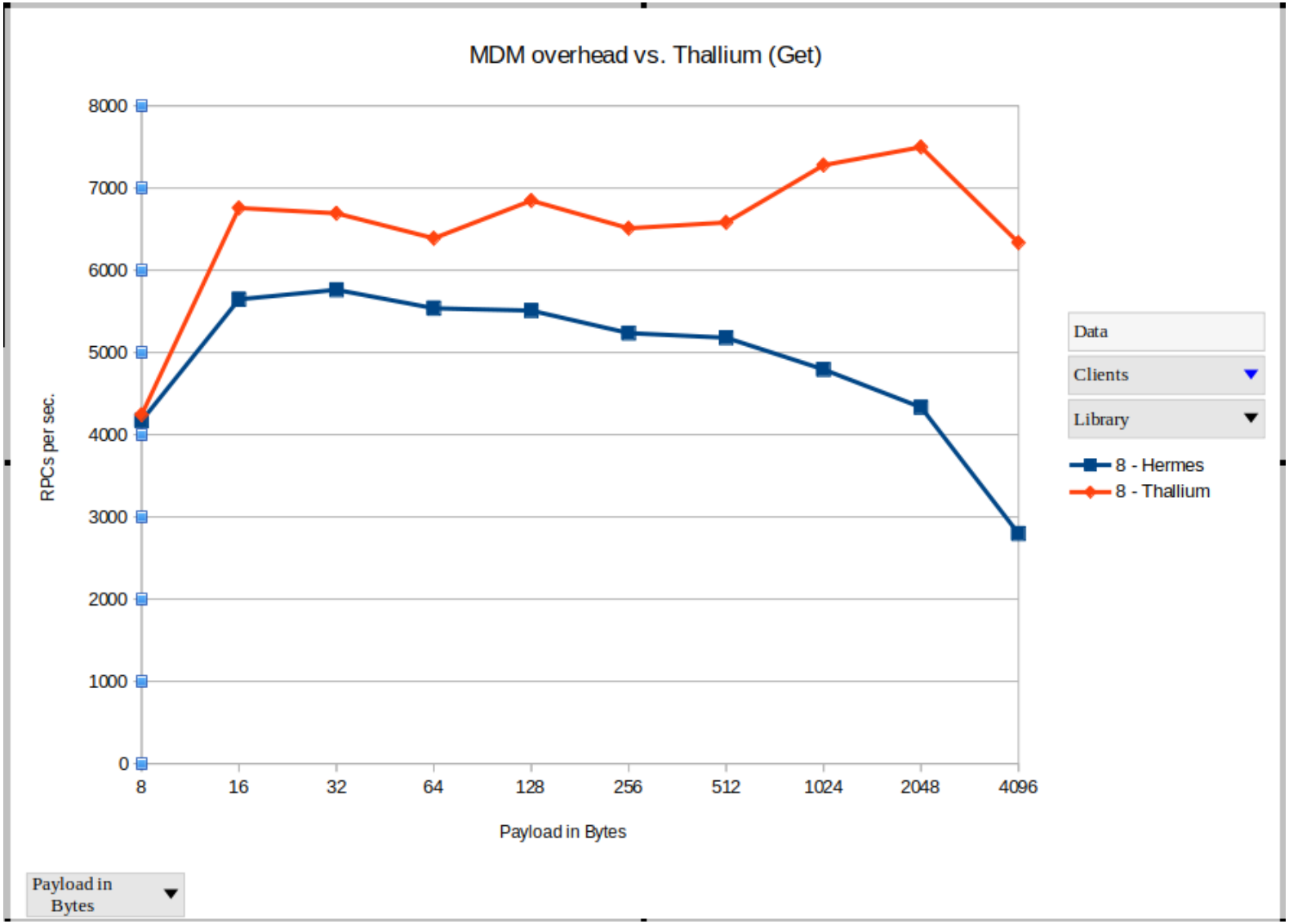
Single Server Remote Performance

We're currently using Thallium (https://mochi.readthedocs.io/en/latest/thallium.html) for the RPC layer (largely for RDMA support). To understand the overhead of the MDM, we run similar workloads in pure Thallium and Hermes, fetching a variable sized vector of 64-bit integers via RPC. For single client single server workloads, the overhead of the MDM over pure Thallium is roughly 300%. This is because the 4 RPC handler threads are able to operate independently in Thallium, but require synchronization in Hermes. They must take a lock when

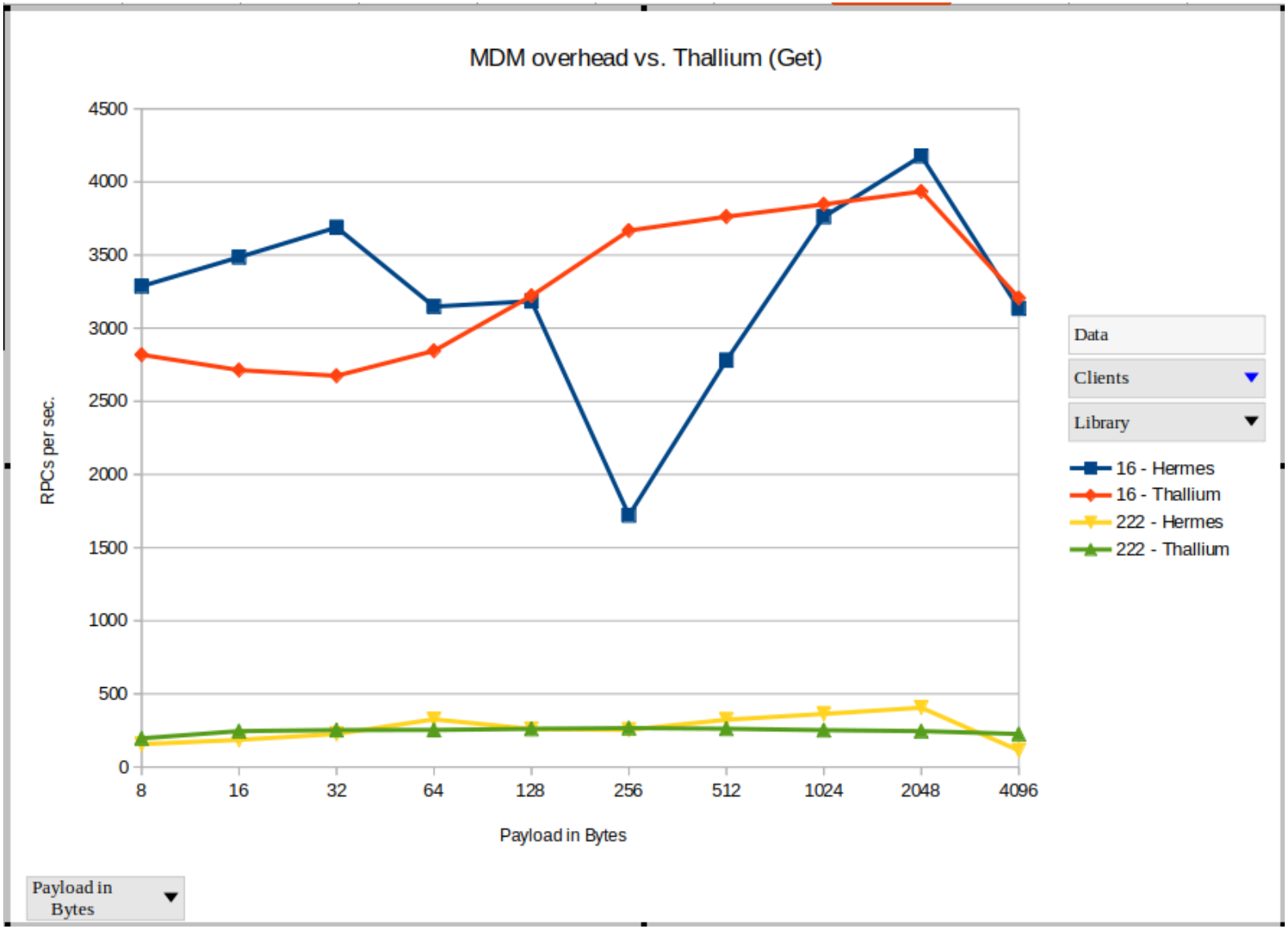
copying from the MDM shared memory.



However, with a sufficient amount of clients this gap narrows considerably. With 8 clients the overhead shrinks to roughly 18%.

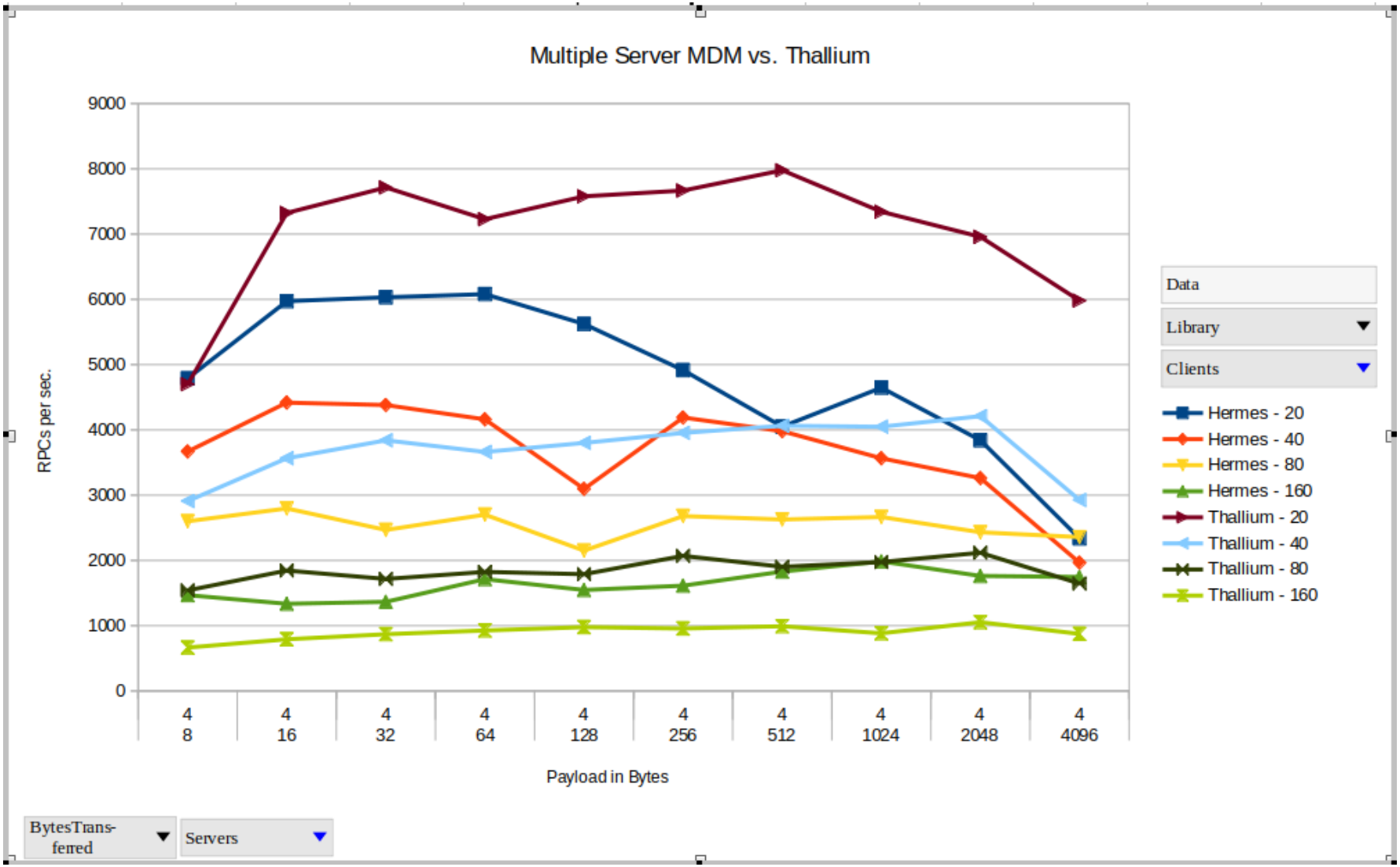


And with anything over 16 clients the gap shrinks even further. With 222 clients the overhead is practically nothing, but notice that a single server is overloaded with so many clients, and the RPCs per second drops to around 200. Therefore, we must also scale the number of servers.

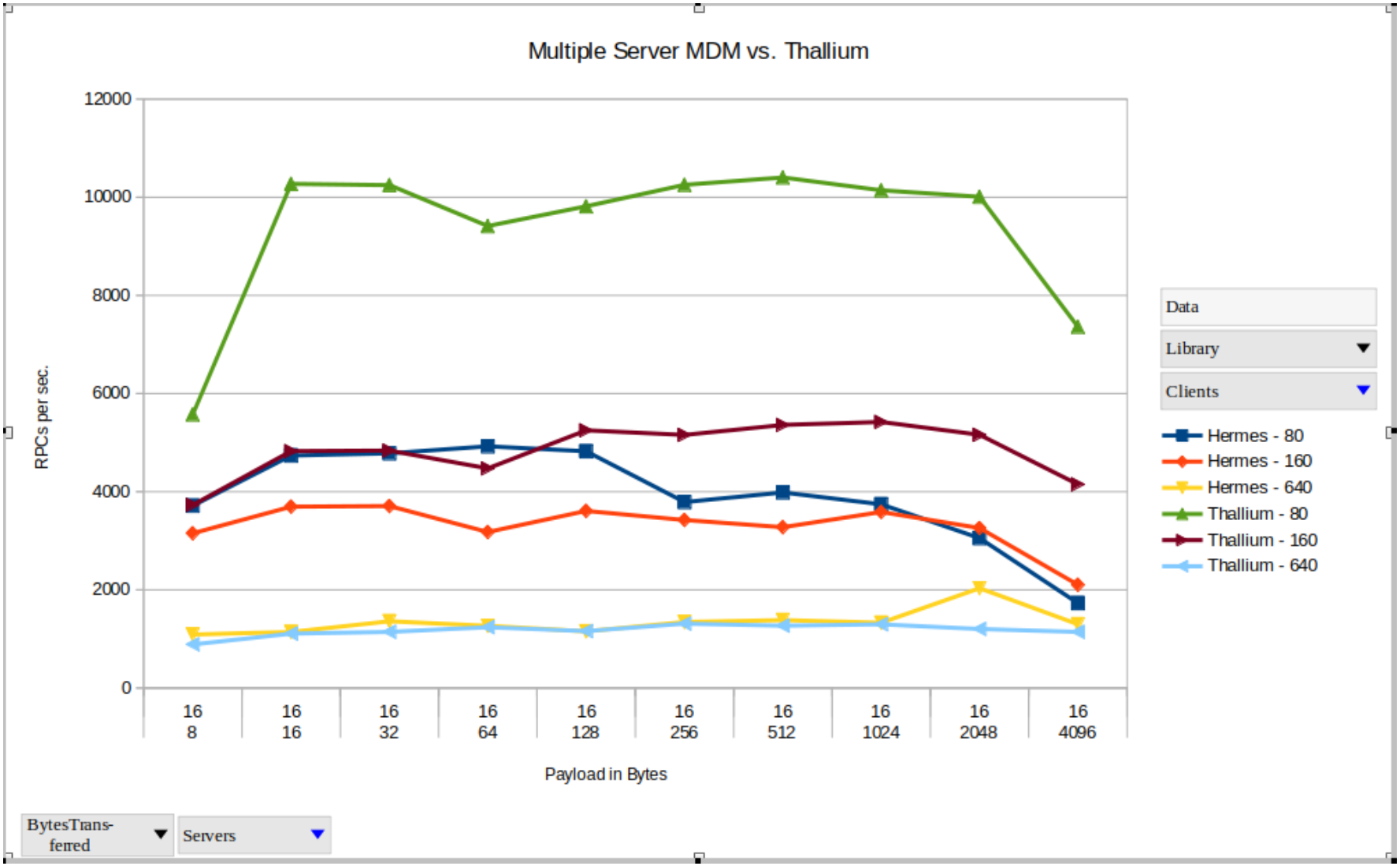


Multiple Server Remote Scalability

To measure the overhead of multiple MDM servers over pure Thallium, we again run similar workloads, this time scaling the number os servers in addition to the clients. This time each node is running the same number of clients, so 1 / N clients will be doing local operations as opposed to RPCs. With 4 servers and 20 clients (5 clients per server) we see roughly 25% overhead, but 40 clients the overhead becomes negligible. Hermes is actually appears faster than pure Thallium with over 80 clients, but this requires some investigation as it does not seem possible.



Scaling the servers up to 16, we again see a wide gap with relatively few clients (5 per node), with the gap closing as we approach full saturation (40 clients on each node).



System View State (SVS)

- Each node has a local SVS, and one node has a global SVS.
- Whenever the BPM acquires or releases `Buffers`, it tracks the change in the node's local SVS's "capacities" field.
- Every X (configurable) milliseconds, a thread on each node wakes up and applies all changes in the local SVS to the global SVS.
- When the DPE runs, it might grab the global SVS and make decisions based on that snapshot, or it may decide it can make a decision without the global snapshot.
