INFINIDAT
STORING THE FUTURE

INFINIBOX I/O PERFORMANCE MONITORING

Lat updated: 02/12/2019

Table of Contents

1 How InfiniBox reports on I/O operations

1.1 Creating collectors to monitor specific I/O

InfiniBox features an efficient data distribution architecture that uses all drives and compute resources all the time. This architecture not only delivers extremely high throughput and IOPs, but also provides visibility to key performance indicators, allowing you to locate and analyze storage performance behavior with surgical accuracy.

Data at the system level is available all the time, and more granular data can be collected on demand. To do that you have to specify what types of I/O operations are interesting, which can be done based on the datasets and clients involved in the I/O, the type of I/O operations, the results and many other parameters. This document includes detailed specification of the various I/O parameters available, and how to use them to monitor various types of activities.

1.2 Tools to monitor I/O

InfiniBox GUI provides an excellent tool to monitor current performance behavior. It allows you to view system behavior in the last 3 minutes (starting when you open the performance view), and to breakdown the system-level information to pin-point specific I/O operations. You can monitor I/O for specific hosts, volumes and filesystems, and you can look at the I/O behavior based on operation type, size or status. Additionally, you can view top resource consumers: volumes, filesystems, clients and even files.

To view long-term performance information use InfiniMetrics, which provides historical performance data at the system, pools, datasets, clusters and hosts level. InfiniMetrics allows you to easily identify performance bottlenecks and correlate them to changes in application I/O behavior. InfiniMetrics is included with InfiniBox at no additional cost.

Finally, you can use InfiniBox LiveCounters API to create custom-tailored reports. This document includes a comprehensive description of the API and available data.

1.3 Sources of I/O reports

InfiniBox software provides multiple data services:

- Access to volumes using block protocols (FC and iSCSI) also known as SAN
- Access to filesystems using file protocols (NFS) also known as NAS
- Effect of QoS limits on SAN activity (limiting IOPS and/or throughput for volumes)
- Async Replication I/O

Each type of service has different dimensions and parameters that it can apply to monitoring I/O activity (for example, NAS can monitor I/O performed by a specific user which is not relevant for SAN activity). The API for retrieving monitoring data is shared by all services, each service provides a separate set of I/O parameters and counters.

1.4 Monitoring frontend I/O

Frontend I/O, which means I/O operation initiated by clients is monitored (separately) by the SAN and NAS components. You can monitor all I/Os (unfiltered) or you can define which types of I/O operation should be considered. This is called *filtering* the I/O patterns.

Every I/O operation that is served by InfiniBox is counted if it matches the parameters required. For example, if you set a filter to monitor I/O on a specific volume, any I/O to this volume is included.

There can be multiple monitors (AKA *collectors*) running simultaneously, each one may look at different or overlapping I/O patterns, depending on the filters used. InfiniBox will include I/O operations in each relevant monitor, so each one provides accurate and consistent information.

1.5 Monitoring replication I/O

All async-replication I/O can be monitored by the dedicated component RMR. This component monitors any replication I/O both incoming or outgoing (and you can differentiate between them).

1.6 Monitoring effect of QoS

The QoS counters provide details about how QoS affects I/O operations, and why. The metrics allow users to identify when I/O is limited by a policy for a volume or a policy for a pool, and whether the limit is based on IOPS or throughput.

Any I/O operation is charged and validated against the relevant QoS policies. The decision may be one of three: the I/O is permitted to go ahead (AKA success), the I/O is delayed so that it will meet the QoS policy (AKA lag), or the I/O is denied (AKA reject). Clients are not aware when I/O is lagged (they only see an increase in latency), but they are aware when I/O is rejected.

Not all SCSI operations are counted as IOPS for QoS purpose. For example, metadata operations such as REPORT LUNS and INQUIRY are ignored by QoS. Also, UNMAP, WRITESAME and XCOPY are counted as a single IOP, but no throughput is checked. Because of these difference you should not expect the total IOPS and total throughput of QoS to be identical to those in SAN.

Note that the QoS counters take place when the I/O arrives at the InfiniBox system, but the matching front-end counters are updated when the I/O is complete. This might make a small difference when operations are delayed for a short while, say 100ms, causing them to be counted by second X for QoS and by second X+1 for SAN. Note: I/O generated by async-replication does not count for QoS purpose.

2 How Collectors and Filters work

2.1 Collectors and filters

To monitor I/O activity one has to create a *filter* and a *collector*.

The filter allows you to slice-and-dice the stream of I/O operations, in order to define the subset of operations that should be counted. For example, a filter that is limited to the protocol FC matches all SAN I/O operations that arrive via FC, a filter that is limited to the protocol FC and to a specific pool matches only I/O operations on volumes which reside in that pool. See further sections in the document to view the specific filter fields available for each service.

A collector allows you to specify the statistics to monitor, such as throughput (or variants on throughput), IOPS, latency, cache hit ratios, etc. When combined with a filter (an empty filter means collect every I/O operation), collectors provide a comprehensive tool to investigate I/O patterns and their behavior.

2.2 Filters – how I/O operations can be filtered

Filters are empty by default (every I/O operation is counted), and you can add parameters to limit I/O. To indicate the power available in filters, if needed you can monitor read I/O operations of size 4KB-8KB on a particular file in a specific filesystem performed by a user on a specific NFS client... Select the relevant filter parameters to identify the activity you're interested in.

Various filter fields may receive different type of values, see Filter parameter values.

To create a new filter, issue the following REST API with the JSON data (replace "*SERVICE*" with the relevant service for the collector you want, one of "SAN", "NAS", "RMR" or SAN_QOS):

```
POST /api/rest/metrics/filters
{ "protocol_type": "SERVICE" }
```

The filter creation response returns a filter_id that will be used in further calls:

```
{ "result": { "id": 35184372088996 },
  "error": null,
  "metadata": { "ready": true }
}
```

The add conditions to the filter in order to focus on specific I/O patterns (*refine* the filter), issue the following RESI API (replace *filter_id* with the ID you received in the previous API):

```
PUT /api/rest/metrics/filters/filter_id
{ "filter_field": "value" }
For example:
PUT /api/rest/metrics/filters/35184372088996
{ "operation_category": "read" }
The response should indicate no errors:
{ "result": { "id": 35184372088996 },
  "error": null,
  "metadata": { "ready": true }
}
```

Repeat the above process to refine the filter with multiple conditions on various fields.

2.2.1 Filter parameter values

Values for filter parameter may require formatting, according to the data type of the parameter:

- Some filter fields are of type enum, which means they can have a single value out of a set of possible values.
  For example, operation_category for NAS can be either read, write or meta_data.
- Many filter fields represent objects created and managed via InfiniBox GUI and CLI, such as volumes, filesystems, pools, hosts, etc.
  When specifying an object, use the ID as reported by InfiniBox API for that object (AKA management ID).
- Some filter fields represent IP addresses, such as the initiator IP address for an iSCSI client.
  IP addresses should be converted to an integer representation, by (A) converting each octet to HEX number, (B) combining the four HEX numbers to a single word, least significant value first, and finally (C) converting the HEX word to decimal integer.
  For example, the address 172.16.1.40 (A) coverts to 0xAC.0x10.0x01.0x28, then (B) converts to 0x280110AC, and finally (C) converts to 671158444.
- Some filter fields represent individual FC ports, as WWPNs.
  WWPNs should be converted to an integer representation, by (A) concatenating the 8 octets to a single HEX word, and then (B) converting the HEX word to decimal integer.
  For example, the WWPN 10:00:00:00:C9:95:C2:B2 (A) converts to 0x10000000C995C2B2, and then (B) converts to 1152921507988882098.
- iSCSI filter fields include the initiator_iqn which can filter iSCSI hosts based on iSCSI initiator.
  Simply specify the initiator name as a string.
- NAS filter fields include uid and gid fields that allow specifying a user or group.
  Simply use the integer UID or GID as relevant.
- NAS filter fields include treeq and file_path fields, that allow specifying I/O operations for a particular file or for files in a particular TreeQ.
  These objects are only unique within a specific filesystem. In order to filter for I/O on these objects, represent them in the API as a JSON tuple of two values: the filesystem ID (management ID) and the file path or TreeQ name as a string.

2.3 Collectors – how I/O operation monitoring can be initiated and sampled

Once you have defined a filter, you can use it to create collectors that will count and sum up I/O operations statistics on any activity that matches the filter.

When you create a collector, you can specify which *collector fields* (I/O attributes) you want to sample. These can be latency-related (measure various elapsed time for I/O operations to complete), throughput-related (measure various attributes about the size of I/O operations) and IOPs-related (measure various counts of I/O operations). Each service provides different collector fields, although there is a common theme among them.

Once you create a collector, it will start to collect the statistics about the requested fields at 1 second granularity, which means you will have a set of data available for every second. Note, however, that a collector of type Aggregator (see further details later) allows you to specify the granularity.

Once the collector started running, you can read samples by querying the collector API. InfiniBox maintains a set of samples (at 1 second granularity), so even if you missed a sample you will receive the information at the next time you query the API. Each time you query, the list of samples is returned to the API client and erased from InfiniBox cache, so you never receive the same data twice. Note: InfiniBox has a limited number of samples it can store, so if not sampled for 500 seconds it will erase the cache and delete the collector and filter.

Each time you query the collector API, the data returned contains the timestamp for the first data set in the response. The next data set in the response, if it exists, is 1 seconds later, and so on.

You can specify multiple collectors in a single query API request. The number of collectors that can be sampled in a single request depends on their type and is explained in the next sections.

The total number of active collectors per service (whether they are queried or not) is limited to 400. Note: InfiniBox GUI and InfiniMetrics consume several collectors.

When you create a collector, you must include the filter_id to identify the filter for the collector.

Once you created collectors, they begin to accumulate statistics at 1 second intervals. You can sample these statistics at a larger interval, in which case you will receive a data set for each 1 second since the last sample taken.

Repeat the sampling until you no longer need the data. Note that if a collector is not sampled for some time (depending on the collector type), InfiniBox will delete it and automatically remove the statistics it contained. In such a case the following error is returned:

```
{
  "result": null,
  "error": {
    "code": "LC_INVALID_COLLECTOR_ID",
    "message": "Collector id 105553116266580 is invalid",
    "reasons": [],
    "severity": "ERROR",
    "is_remote": false,
    "data": null
  },
  "metadata": {
    "ready": true
  }
}
```

2.3.1 Counter collectors

This is the simplest type of collector: it simply counts or sums up the operations that match the filter. Every sample will contain a single set of values representing the I/O for a specific 1 second interval (the values depend on the collector fields you specified).

Counter collectors are very simple to use and consume very few resources.

If the collector is not sampled for 28 seconds, the older samples are discarded. If the collector is not sampled for 500 seconds, the collector will be automatically deleted.

To create a new Counter collector, issue the following REST API with the JSON data (replace *filter_id* with the ID of the filter you created earlier, and replace the *fieldX* with the collector fields specifying which statistics you want):

```
POST /api/res/metrics/collectors
{ "filter_id": filter_id,
  "type": "COUNTER",
  "collected_fields": ["field1", "field2" ...]
}
```

For example:

```
POST /api/res/metrics/collectors
{ "filter_id": 35184372088996,
  "type": "COUNTER",
  "collected_fields": ["ops", "throughput"]
}
```

The collector creation response returns a collector_id that will be used in further calls, and also the specific details describing the collector:

```
{ "result":
  { "id": 35184372089045,
    "filter_id": 35184372088996,
    "collected_fields": ["ops", "throughput"],
    "type": "COUNTER" },
    "error": null,
    "metadata": { "ready": true }
}
```

To collect the statistics from one (or more) collectors, issue the following REST API call:

```
GET /api/rest/metrics/collectors/data?collector_id=in:<id>[,...]
```

The following example response shows the results for a query on a single Counter collector:

```
{ "result":
  { "collectors": [
      { "id": 35184372089043,
        "fields": [ "ops", "throughput" ],
        "data": [
          [14451, 467937729],
          [12736, 413005671] ],
        "collector_type": "COUNTER",
        "interval_milliseconds": 1000,
        "end_timestamp_milliseconds": 1496922832312
      } ]
  } ,
  "error": null,
  "metadata": { "ready": true }
}
```

Note:

- The data entity shows two samples, one sample per second.
- Use the end_timestamp_millisecond and interval_milliseconds values to calculate the timing of each sample.
- Each sample contains two values, one for the ops field, and the second for throughput.
- The order of the values inside each sample is defined as the fields appear in the fields entity.

2.3.2 Counter collector – a shortcut

There is a shortcut to creating a Collector counter, where the filter parameters are described together with the collector, i.e. a filter is created automatically.

The API format is:

```
POST /api/res/metrics/collectors
{   "filters": {
      "protocol_type": "SAN",
      "operation_category": "read" },
  "type": "COUNTER",
  "collected_fields": ["field1", "field2" ...]
}
```

2.3.3 Histogram collectors

Histograms collectors provide a simple way to analyze I/O behavior by breaking the summary data (such as returned by a Counter collector) down to smaller buckets.

A Histogram collector allows you to select a single filter field of type enum (i.e. a filter field that has a predefined set of possible values) and retrieve a data set for each possible enum value. Use this to quickly identify an I/O attribute that is common to most I/O operations.

For example, say you want to identify what I/O operations are most prevalent: read, write, unmap, etc. You *could* create multiple filter and collectors, each one sampling different operation category... however, a single Histogram collector will provide you with all that information.

Every possible enum value is collected separately and is returned with every sample separately. This means each 1 second data set contains multiple tuples of samples, one per possible enum value.

Note: once you identified an interesting I/O attribute using the Histogram collector, you can create a new filter with additional filter fields on that attribute value, and further analyze other aspects of these I/O operations.

If the collector is not sampled for 28 seconds, the older samples are discarded. If the collector is not sampled for 500 seconds, the collector will be automatically deleted.

To create a new Histogram collector, issue the following REST API with the JSON data (replace *filter_id* with the ID of the filter you created earlier, replace the *fieldX* with the collector fields specifying which statistics you want, and replace *enum_filter_field* with the name of the enum field you want to investigate):

```
POST /api/res/metrics/collectors
{ "filter_id": filter_id,
  "type": "HISTOGRAM",
  "collected_fields": ["field1", "field2" ...],
  "histogram_field": "enum_filter_field"
}
```

For example:

```
POST /api/res/metrics/collectors
{ "filter_id": 35184372088996,
  "type": "HISTOGRAM",
  "collected_fields": ["ops", "throughput"]
  "histogram_field": "operation_category"
}
```

The collector creation response returns a collector_id that will be used in further calls, and also the specific details describing the collector:

```
{ "result":
  { "id": 35184372089045,
    "filter_id": 35184372088996,
    "collected_fields": ["ops", "throughput"],
    "histogram_field": "operation_category",
    "type": "HISTOGRAM" },
    "error": null,
    "metadata": { "ready": true }
}
```

To collect the statistics from one (or more) collectors, issue the following REST API call:

```
GET /api/rest/metrics/collectors/data?collector_id=in:<id>[,...]
```

The following example response shows the results for a query on a single Histogram collector:

```
{ "result":
  { "collectors": [
      { "id": 35184372089043,
        "fields": [ "ops", "throughput" ],
        "data": [
          [ [641, 30521856],  [3751, 111668224],  [0, 0] ],
          [ [1837, 87529598], [10901, 323443980], [0, 0] ] ],
        "collector_type": "HISTOGRAM",
        "histogram_field": "operation_category",
        "ranges": [ "read", "write", "xcopy" ],
        "interval_milliseconds": 1000,
        "end_timestamp_milliseconds": 1496922831312
      } ]
  } ,
  "error": null,
  "metadata": { "ready": true }
}
```

Note:

- The data entity shows two samples, one sample per second.
- Use the end_timestamp_millisecond and interval_milliseconds values to calculate the timing of each sample.
- Each sample contains three data sets, one for each of the enum values (read, write and xcopy).
- The order of the values inside each sample is defined as the fields appear in the fields entity. In the sample above one for the ops field, and the second for throughput.

Therefore, the example result JSON document above translates to the following:

Timestamp	operation_category	ops	throughput
1496922831312 (2017-06-08 11:53:51.312 UTC)	read	641	30521856
	write	3751	111668224
	xcopy	0	0
1496922831312 (2017-06-08 11:53:51.312 UTC)	read	1837	87529598
	write	10901	323443980

Timestamp	operation_category	ops	throughput
	xcopy	0	0

2.3.4 Top collectors

Histogram collectors are useful when the possible values for a filter field are predefined, like in enum fields. Top collectors provide similar capability for fields that represent I/O operations attributes whose values are not known in advance and could be extremely numerous. For example, if you want to identify the hosts that are performing the most IOPS, the list of hosts could be quite large.

Typically, every filter field (that is not of type enum) can be a grouping field, which specifies how to breakdown the I/O operations. The collector will maintain data set for each actual field value separately and will return these with every sample separately.

For example, say you want to identify an NFS client that is consuming a lot of bandwidth. You can create a Top collector to breakdown the relevant I/O operation by NFS client identity, such as the client IP address. At every second, InfiniBox will maintain a list of all active NFS clients and their throughput.

This means each 1 second data set contains multiple tuples of samples, one per actual value of the grouping that caused the I/O. The Top collector doesn't maintain the full list of values: instead it will keep in cache only the Top N values (N can be up to 100) based on some collector field (e.g. throughput). Note that because every second contains different I/O pattern, the list of returned entities might be different at each second.

Each data set that represents a single second of I/O may contain up to N items. Items that did not perform any I/O will not appear in the data set.

Note: as with Histogram collectors, once you identified an interesting I/O attribute using the Top collector, you can create a new filter with additional filter fields on that attribute value, and further analyze other aspects of these I/O operations.

If the collector is not sampled for 28 seconds, the older samples are discarded. If the collector is not sampled for 500 seconds, the collector will be automatically deleted.

To create a new Top collector, issue the following REST API with the JSON data (replace *filter_id* with the ID of the filter you created earlier, replace the *fieldX* with the collector fields specifying which statistics you want, replace *filter_field* with the name of the field you want to investigate, and replace *sort_field* with the name of the collector field for cutting the top entities, and set *sort_ascending* to true for ascending sort or false for descending sort):

```
POST /api/res/metrics/collectors
{ "filter_id": filter_id,
  "type": "TOP",
  "collected_fields": ["field1", "field2" ...],
  "grouping_field": "filter_field",
  "sorting_field": "sort_field",
  "sort_ascending": use_ascending,
  "max_results": max_results
}
```

For example:

```
POST /api/res/metrics/collectors
{ "filter_id": 35184372088996,
  "type": "TOP",
  "collected_fields": ["ops", "throughput"]
  "grouping_field": "vol_id",
  "sorting_field": "ops",
  "sort_ascending": false,
  "max_results": 10
}
```

The collector creation response returns a collector_id that will be used in further calls, and also the specific details describing the collector:

```
{ "result":
  { "id": 35184372089045,
    "filter_id": 35184372088996,
    "collected_fields": ["ops", "throughput"],
    "grouping_field": "vol_id",
    "sorting_field": "ops",
    "sort_ascending": false,
    "max_results": 10,
    "type": "TOP" },
    "error": null,
    "metadata": { "ready": true }
}
```

To collect the statistics from one (or more) collectors, issue the following REST API call:

```
GET /api/rest/metrics/collectors/data?collector_id=in:<id>[,...]
```

The following example response shows the results for a query on a single Top collector:

```
{ "result":
  { "collectors": [
      { "id": 35184372089043,
        "fields": ["vol_id", "ops", "throughput"],
        "data": [
          [ ["61321", 2794, 67253941 ],
            ["61317", 2352, 101169969],
            ["61319", 2324, 77666658 ] ],
          [ ["61321", 4636, 111570820],
            ["61317", 3471, 149286622] ] ],
        "collector_type": "TOP",
        "grouping_field": "vol_id",
        "sorting_field": "ops"
        "interval_milliseconds": 1000,
        "end_timestamp_milliseconds": 1497259383017,
      } ]
  } ,
  "error": null,
  "metadata": { "ready": true }
}
```


Note:

- The data entity shows two samples, one sample per second.
- Use the end_timestamp_millisecond and interval_milliseconds values to calculate the timing of each sample.
- Each sample contains several data sets, up to the max_results defined in the collector, each data set contains the value of the group_field.
- The order of the values inside each sample is defined as the fields appear in the fields entity.

2.3.5 Aggregator collectors

Aggregator collectors are similar to Top collectors and differ in two ways:

Aggregator collectors provide the FULL list of items that were samples for every second. It does not filter the top N items. Note: items that did not perform any I/O will not appear in the data set.

Additionally, the Aggregator allows you to set the data collection period, between 1 second and 5 minutes, specifically the interval can be 1 sec, 10 sec, 30 sec, 1 min, 2 min or 5 min. Whichever the chosen interval is, InfiniBox will maintain in cache 5 samples of statistics, so choosing a sampling interval also affects the rate in which you need to collect the samples: if the collector is not sampled for 5 x interval, the older samples are discarded. Note the interval is defined in milli-seconds.

If the collector is not sampled for 8 x interval or 80 seconds (whichever is larger), the collector will be automatically deleted.

While Aggregator collectors are very simple to use, they potentially consume much more resources that Counter collectors or Histogram collectors. Therefore, the number of active Aggregator collectors per service is limited to 20. Note: InfiniMetrics consumes several Aggregator collectors. INFINIDAT recommends you consult with INFINIDAT professional services before using Aggregator collectors.

To create a new **Aggregator** collector, issue the following REST API with the JSON data (replace *filter_id* with the ID of the filter you created earlier, replace the *fieldX* with the collector fields

specifying which statistics you want, replace *filter_field* with the name of the field you want to investigate):

```
POST /api/res/metrics/collectors
{ "filter_id": filter_id,
  "type": "AGGREGATOR",
  "collected_fields": ["field1", "field2" ...],
  "grouping_field": "filter_field",
  "interval_milliseconds": interval_ms
}
```

For example:

```
POST /api/res/metrics/collectors
{ "filter_id": 35184372088996,
  "type": "AGGREGATOR",
  "collected_fields": ["ops", "throughput"]
  "grouping_field": "vol_id",
  "interval_milliseconds": 10000
}
```

The collector creation response returns a collector_id that will be used in further calls, and also the specific details describing the collector:

```
{ "result":
  { "id": 35184372089045,
    "filter_id": 35184372088996,
    "collected_fields": ["ops", "throughput"],
    "grouping_field": "vol_id",
    "type": "AGGREGATOR" },
    "error": null,
    "metadata": { "ready": true }
}
```

To collect the statistics from one (or more) collectors, issue the following REST API call:

```
GET /api/rest/metrics/collectors/data?collector_id=in:<id>[,...]
```

The following example response shows the results for a query on a single Aggregator collector:

```
{ "result":
  { "collectors": [
      { "id": 35184372089043,
        "fields": ["vol_id", "ops", "throughput"],
        "data": [
          [ ["61321", 2794, 67253941 ],
            ["61317", 2352, 101169969],
            ["61319", 2324, 77666658 ] ],
          [ ["61321", 4636, 111570820],
            ["61317", 3471, 149286622] ] ],
        "collector_type": "AGGREGATOR",
        "grouping_field": "vol_id",
        "interval_milliseconds": 1000,
        "end_timestamp_milliseconds": 1497259383017,
      } ]
  } ,
  "error": null,
  "metadata": { "ready": true }
}
```


Note:

- The data entity shows two samples, one sample per second.
- Use the end_timestamp_millisecond and interval_milliseconds values to calculate the timing of each sample.
- Each sample contains an unbounded number of data sets, each data set contains the value of the group_field.
- The order of the values inside each sample is defined as the fields appear in the fields entity.

3 Filter & Collector Fields

3.1 SAN FC/iSCSI frontend I/O

See Monitoring frontend I/O for details on the data reported by this service

3.1.1 Filter/grouping fields

API Field Name	GUI Field Name	Values	Meaning
protocol	Protocol	Enum: FC, iSCSI	The SAN protocol
host_id	Host	ID in InfiniBox mgmt DB	The ID of the host
initiator_port_key	N/A	ID in InfiniBox mgmt DB	The ID (key) of the host port
initiator_IQN	Initiator (iSCSI)	String	The host iSCSI initiation IQN
source_ip	Host IP (iSCSI)	Integer for IP address	The host iSCSI IP address
initiator_WWPN	Initiator (FC)	Integer for FC WWPN	The host FC port WWPN
destination_ip	Target IP (iSCSI)	Integer for IP address	The iSCSI IP address on InfiniBox
network_space_id	Network Space	ID in InfiniBox mgmt DB	The ID of the iSCSI NetSpace
target_WWPN	Target WWPN	Integer for FC WWPN	The FC port WWPN on InfiniBox
pool_id	Pool	ID in InfiniBox mgmt DB	The ID of the pool containing vol

API Field Name	GUI Field Name	Values	Meaning
vol_id	Volume	ID in InfiniBox mgmt DB	The ID of the volume
operation_type	Operation Type	Enum: caw, format_unit, inquiry, log_sense, login, mode_select, mode_sense, persistent_reserve_in, persistent_reserve_out, read, read_buf, read_capacity, receive_copy_result, release, report_devid, report_id, report_luns, request_sense, reserve, rtpg, send_diagnostic, start_stop_unit, sync_cache, write_same, xcopy, task_management_functions, test_unit_ready, unmap, verify, write, write_buf	The exact SCSI command
operation_category	Operation Category	Enum: login, other, read, unmap, write, writesame, xcopy	A category for SCSI commands
operation_size_histogram	Operation Size	Enum: lt_4k, ge_4k_lt_8k, ge_8k_lt_32k, ge_32k_lt_64k, ge_64k_lt_512k, ge_512k	The size bucket of the I/O
SCSI Status		Enum: busy, error, good	The I/O SCSI result
io_sync_replicated	IO Sync Replicated	Enum: false, true	True if the I/O operation was mirrored to a sync-replica

3.1.2 Collector fields

Field Name	Values
throughput	The total size (in bytes/sec) of **successful** SCSI operations
operation_size	The total size (in bytes/sec) of **all** SCSI operations
average_operation_size	The average size (in bytes) of all SCSI operations

Field Name	Values
external_latency	The total time (in µs) the I/O operation took, from the time it arrived (started to arrive) to InfiniBox until the time the final response was sent to the host. External latency is affected by internal latency but may increase due to host / fabric delays and due to QoS.
internal_latency	The total time (in µs) InfiniBox spent handling the I/O operation (i.e. without waiting for more data from/to the host), until the time the final response was sent to the host. Internal latency is affected by sync-replication but is neither affected by host / fabric delays nor by QoS.
external_latency_wout_err	Same as external latency, but including incomplete SCSI operations, i.e. timeouts. These value skew latency largely because they measure in seconds rather than milli-seconds.
internal_latency_wout_err	Same as internal latency, but including incomplete SCSI operations, i.e. timeouts. These value skew latency largely because they measure in seconds rather than milli-seconds.
sections_read	The total number of 64KB sections InfiniBox accessed
sections_read_from_cache	The total number of 64KB sections InfiniBox accessed and found the data in DRAM cache
sections_read_from_ssd	The total number of 64KB sections InfiniBox accessed and found the data in SSD read-only cache
sections_read_from_disk	The total number of 64KB sections InfiniBox accessed and performed read from HDD to provide the response
ops	The total number of SCSI operations
aborted_task	Number of tasks aborted by the InfiniBox backend
errors	Number of errors detected in SCSI operations

3.2 SAN QoS effect

See Monitoring effect of QoS for more details on the data reported by this service.

3.2.1 Filter/grouping fields

API Field Name	GUI Field Name	Values	Meaning
qos_entity_id	N/A	ID in InfiniBox mgmt DB	The ID of the dataset with QoS
qos_pool_id	N/A	ID in InfiniBox mgmt DB	The ID of the pool
qos_status	N/A	Enum: lag, reject, success	The effect of the QoS rule: success – no effect, lag – I/O is postponed for a while, reject – I/O was rejected
resource_type	N/A	Enum: entity, pool	Type of rule involved: a rule for a volume or a rule for a pool
resource_unit	N/A	Enum: bw, io	Type of limit the caused the QoS: a limit of bandwidth or for IOPS

3.2.2 Collector fields

Field Name	Values
qos_latency	The average time (in µs) I/O operations were delayed, due to a decision to add a lag to the operation.
approved_ios	The number of operations per second
approved_throughput	Total size (in bytes) of operations per second

3.3 NAS NFS frontend I/O

See Monitoring frontend I/O for more details on the data reported by this service.

3.3.1 Filter/grouping fields

API Field Name	GUI Field Name	Values	Meaning
source_ip	Client IP	Integer for IP address	The host IP address
user_id	User ID (UID)	Integer	The UID submitting the I/O
group_id	Group ID (GID)	Integer	The GID submitting the I/O
destination_ip	Server IP	Integer for IP address	The NFS IP address on InfiniBox
pool_id	Pool	ID in InfiniBox mgmt DB	The ID of the pool containing FS
fs_entity_id	Filesystem	ID in InfiniBox mgmt DB	The ID of the FS accessed
export_id	Export	ID in InfiniBox mgmt DB	The ID of the NFS export accessed
Treeq	TreeQ	A tuple with FS ID and treeq name	The FS and TreeQ accessed
file_path	File	A tuple with FS ID and file path	The FS and File accessed
operation_type	Operation Type	Enum: access, commit, create, dump, export, fs_information, fs_statistics, get_attributes, link, lookup, make_directory, make_node, mount, null_mnt, null_nfs, path_configuration, read, read_directory, read_directory_plus, read_link, remove, remove_directory, rename, set_attributes, symbolic_link, unmount, unmount_all, write	The exact NFS verb
operation_category	Operation Category	Enum: meta_data, read, write	A category for NFS verbs
operation_size_histogram	Operation Size	Enum: lt_4k, ge_4k_lt_8k, ge_8k_lt_32k, ge_32k_lt_64k, ge_64k_lt_512k, ge_512k	The size bucket of the I/O

3.3.2 Collector fields

Field Name	Values
average_operation_size	The average size (in bytes) of all read and/or write NFS operations
throughput	The total size (in bytes/sec) of all read and/or write NFS operations
latency	The total time (in µs) the I/O operation took, from the time it arrived (started to arrive) to InfiniBox until the time the response was sent to the host
sections_read	The total number of 64KB sections InfiniBox accessed
sections_read_from_cache	The total number of 64KB sections InfiniBox accessed and found the data in DRAM cache
sections_read_from_ssd	The total number of 64KB sections InfiniBox accessed and found the data in SSD read-only cache
sections_read_from_disk	The total number of 64KB sections InfiniBox accessed and performed read from HDD to provide the response
ops	The total number of NFS operations

3.4 RMR I/O

See Monitoring replication I/O for more details on the data reported by this service.

3.4.1 Filter/grouping fields

API Field Name	GUI Field Name	Values	Meaning
operation_type	Operation Type	Enum: read, write	The direction for the replication: read – replication is outgoing write – replication is incoming
replica_id	Replica	ID in InfiniBox mgmt DB	The ID of the replica object

API Field Name	GUI Field Name	Values	Meaning
dataset_id	Volume	ID in InfiniBox mgmt DB	The ID of the local dataset
pair_dataset_id	N/A	ID in InfiniBox mgmt DB	The ID of the remote dataset
node_id	Node	Enum: 1, 2 or 3	The ID of the local InfiniBox node performing the replication
replication_type	N/A	Enum: async, sync	The reason for the async replication: async – a replica with async policy; sync – a replica with sync policy is in the process of closing the gap

3.4.2 Collector fields

Field Name	Values
ops	The total number of replication operations
average_operation_size	The average size (in bytes) of all replication operations
throughput	The total size (in bytes) of all data read or written by replication
latency	The total time (in µs) replication operations took
sections_read	The total number of 64KB sections InfiniBox accessed (only relevant for outgoing replications)
sections_read_from_cache	The total number of 64KB sections InfiniBox accessed and found the data in DRAM cache
sections_read_from_ssd	The total number of 64KB sections InfiniBox accessed and found the data in SSD read-only cache
sections_read_from_disk	The total number of 64KB sections InfiniBox accessed and performed read from HDD to provide the response

4 How to use the self-documenting API

When creating a filter we must first specify the protocol or protocol type this filter will be applied to. The API supports discovery of all available filter and collector fields, so we can use this request to get a list of available protocols and types.

Send the following REST API request:

```
GET /api/rest/metrics/available_fields
```

And the response is:

```
{ "result": {
    "available_filter_fields": [
      { "name": "protocol",
        "type": "enum",
        "values":["NAS","NAS_QOS","RMR",
                  "SAN","SAN_QOS","other"] },
      { "name": "protocol_type",
        "type": "enum",
        "values": ["RMR", "SAN", "SAN_QOS" ]
    }]
  },
"error": null,
  "metadata": { "ready": true }
}
```

Once you create a filter, you can query the filter and collectors fields. Send the following REST API request (replace *filter_id* with the ID you received in the previous API):

```
GET /api/rest/metrics/filters/filter_id/available_fields?level=ADVANCED
```

And the response is:

```
{ "result": {
    "id": 211106232542843,
    "available_filter_fields": [
      { "name": "replica_id",
        "type": "uint" },
      { "name": "replication_type",
        "type": "enum",
        "values": [ "async", "sync" ] }
    ],
    "available_collector_fields": [
      { "name": "ops",
        "unit": "N/A",
        "description": "The number of Ops per second" },
      { "name": "average_operation_size",
        "unit": "B",
        "description": "Average operations size" }
    ] },
  "error": null,
  "metadata": { "ready": true }
}
```

5 Live-counters errors codes

Error code / Description
LC_CANNOT_MIX_COLLECTORS Mixing collectors of different types within a single request is forbidden
LC_COLLECTOR_COUNTERS_LIMIT_EXCEEDED The request contains {collectors} collectors, but the number of collectors of type {collector_type} is limited to {colletors_limit}
LC_INVALID_FILTER_ID Filter id {handle_id} is invalid
LC_INVALID_COLLECTOR_ID Collector id {handle_id} is invalid
LC_INVALID_FIELD_NAME Field '{field_name}' is invalid for filter id {handle_id}
LC_INVALID_FIELD_VALUE Value '{field_value}' is invalid for field '{field_name}' and filter id {handle_id}
LC_FIELD_TYPE_MISMATCH Field '{field_name}' is not supported as a {field_type}
LC_INVALID_INTERVAL_MS Interval milliseconds must be in steps of 1000
LC_SORT_FIELD_MISSING_FROM_COLLECTOR Sorted by field must be one of the collector fields
LC_FIELD_IS_UNAVAILABLE Field '{field_name}' is unavailable for filter id {handle_id}, use get_available_fields to list all available fields for this specific filter
LC_CANNOT_UPDATE_FILTER_WITH_COLLECTOR Filter id {handle_id} already has collectors defined and can't be updated. Please create a new filter
LC_FILTER_HANDLES_LIMIT_EXCEEDS Number of filters exceeds the maximum allowed per service({limit})

LC_COLLECTOR_HANDLES_LIMIT_EXCEEDS Number of collectors exceeds the maximum allowed per service({limit})
LC_AGGREGATOR_COUNTERS_LIMIT_EXCEEDED Number of collector aggregators exceeds the maximum allowed per service ({max_allow_aggregators})
LC_DISABLED Performance Analytics service disabled for '{module}' service disabled