Hazelcast Python Client Documentation

Release 4.0

Hazelcast Inc. Developers

Jan 13, 2021

CONTENTS

Hazelcast is an open-source distributed in-memory data store and computation platform that provides a wide variety of distributed data structures and concurrency primitives.

Hazelcast Python client is a way to communicate to Hazelcast IMDG clusters and access the cluster data. The client provides a Future-based asynchronous API suitable for wide ranges of use cases.

CHAPTER

ONE

OVERVIEW

1.1 Usage

```
import hazelcast

# Connect to Hazelcast cluster.
client = hazelcast.HazelcastClient()

# Get or create the "distributed-map" on the cluster.
distributed_map = client.get_map("distributed-map")

# Put "key", "value" pair into the "distributed-map" and wait for
# the request to complete.
distributed_map.set("key", "value").result()

# Try to get the value associated with the given key from the cluster
# and attach a callback to be executed once the response for the
# get request is received. Note that, the set request above was
# blocking since it calls ".result()" on the returned Future, whereas
# the get request below is non-blocking.
get_future = distributed_map.get("key")
get_future.add_done_callback(lambda future: print(future.result()))

# Do other operations. The operations below won't wait for
# the get request above to complete.

print("Map size:", distributed_map.size().result())

# Shutdown the client.
client.shutdown()
```

If you are using Hazelcast IMDG and the Python client on the same machine, the default configuration should work out-of-the-box. However, you may need to configure the client to connect to cluster nodes that are running on different machines or to customize client properties.

1.2 Configuration

```
import hazelcast

client = hazelcast.HazelcastClient(
    cluster_name="cluster-name",
    cluster_members=[
        "10.90.0.2:5701",
        "10.90.0.3:5701",
    ],
    lifecycle_listeners=[
        lambda state: print("Lifecycle event >>>", state),
    ]
)

print("Connected to cluster")
client.shutdown()
```

See the API documentation of *hazelcast.client.HazelcastClient* to learn more about supported configuration options.

CHAPTER

TWO

FEATURES

- Distributed, partitioned and queryable in-memory key-value store implementation, called **Map**
- Eventually consistent cache implementation to store a subset of the Map data locally in the memory of the client, called **Near Cache**
- Additional data structures and simple messaging constructs such as **Set**, **MultiMap**, **Queue**, **Topic**
- Cluster-wide unique ID generator, called **FlakeIdGenerator**
- Distributed, CRDT based counter, called **PNCounter**
- Distributed concurrency primitives from CP Subsystem such as **FencedLock**, **Semaphore**, **AtomicLong**
- Integration with Hazelcast Cloud
- Support for serverless and traditional web service architectures with **Unisocket** and **Smart** operation modes
- Ability to listen to client lifecycle, cluster state, and distributed data structure events
- and many more

2.1 HazelcastClient API Documentation

class HazelcastClient(***kwargs*)

Bases: `object`

Hazelcast client instance to access access and manipulate distributed data structures on the Hazelcast clusters.

Keyword Arguments

- **cluster_members** (*list[str]*) – Candidate address list that client will use to establish initial connection. By default, set to `["127.0.0.1"]`.
- **cluster_name** (*str*) – Name of the cluster to connect to. The name is sent as part of the the client authentication message and may be verified on the member. By default, set to `dev`.
- **client_name** (*str*) – Name of the client instance. By default, set to `hz.client_${CLIENT_ID}`, where `CLIENT_ID` starts from `0` and it is incremented by `1` for each new client.
- **connection_timeout** (*float*) – Socket timeout value in seconds for client to connect member nodes. Setting this to `0` makes the connection blocking. By default, set to `5.0`.
- **socket_options** (*list[tuple]*) – List of socket option tuples. The tuples must contain the parameters passed into the `socket.setsockopt()` in the same order.

- **redo_operation** (*bool*) – When set to `True`, the client will redo the operations that were executing on the server in case if the client lost connection. This can happen because of network problems, or simply because the member died. However it is not clear whether the operation was performed or not. For idempotent operations this is harmless, but for non idempotent ones retrying can cause to undesirable effects. Note that the redo can be processed on any member. By default, set to `False`.
- **smart_routing** (*bool*) – Enables smart mode for the client instead of unisocket client. Smart clients send key based operations to owner of the keys. Unisocket clients send all operations to a single node. By default, set to `True`.
- **ssl_enabled** (*bool*) – If it is `True`, SSL is enabled. By default, set to `False`.
- **ssl_cafile** (*str*) – Absolute path of concatenated CA certificates used to validate server's certificates in PEM format. When SSL is enabled and cafile is not set, a set of default CA certificates from default locations will be used.
- **ssl_certfile** (*str*) – Absolute path of the client certificate in PEM format.
- **ssl_keyfile** (*str*) – Absolute path of the private key file for the client certificate in the PEM format. If this parameter is `None`, private key will be taken from the certfile.
- **ssl_password** (*str|bytes|bytearray|function*) – Password for decrypting the keyfile if it is encrypted. The password may be a function to call to get the password. It will be called with no arguments, and it should return a string, bytes, or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key. Alternatively a string, bytes, or bytearray value may be supplied directly as the password.
- **ssl_protocol** (*int|str*) – Protocol version used in SSL communication. By default, set to `TLSv1_2`. See the *hazelcast.config.SSLProtocol* for possible values.
- **ssl_ciphers** (*str*) – String in the OpenSSL cipher list format to set the available ciphers for sockets. More than one cipher can be set by separating them with a colon.
- **cloud_discovery_token** (*str*) – Discovery token of the Hazelcast Cloud cluster. When this value is set, Hazelcast Cloud discovery is enabled.
- **async_start** (*bool*) – Enables non-blocking start mode of *HazelcastClient*. When set to `True`, the client creation will not wait to connect to cluster. The client instance will throw exceptions until it connects to cluster and becomes ready. If set to `False`, *HazelcastClient* will block until a cluster connection established and it is ready to use the client instance. By default, set to `False`.
- **reconnect_mode** (*int|str*) – Defines how a client reconnects to cluster after a disconnect. By default, set to `ON`. See the *hazelcast.config.ReconnectMode* for possible values.
- **retry_initial_backoff** (*float*) – Wait period in seconds after the first failure before retrying. Must be non-negative. By default, set to `1.0`.
- **retry_max_backoff** (*float*) – Upper bound for the backoff interval in seconds. Must be non-negative. By default, set to `30.0`.
- **retry_jitter** (*float*) – Defines how much to randomize backoffs. At each iteration the calculated back-off is randomized via following method in pseudo-code `Random(-jitter * current_backoff, jitter * current_backoff)`. Must be in range `[0.0, 1.0]`. By default, set to `0.0` (no randomization).
- **retry_multiplier** (*float*) – The factor with which to multiply backoff after a failed retry. Must be greater than or equal to `1`. By default, set to `1.0`.

- **cluster_connect_timeout** (*float*) – Timeout value in seconds for the client to give up a connection attempt to the cluster. Must be non-negative. By default, set to *120.0*.
- **portable_version** (*int*) – Default value for the portable version if the class does not have the `get_portable_version()` method. Portable versions are used to differentiate two versions of the *hazelcast.serialization.api.Portable* classes that have added or removed fields, or fields with different types.
- **data_serializable_factories** (*dict[int, dict[int, class]]*) – Dictionary of factory id and corresponding *hazelcast.serialization.api.IdentifiedDataSerializable* factories. A factory is simply a dictionary with class id and callable class constructors.

```
FACTORY_ID = 1
CLASS_ID = 1

class SomeSerializable(IdentifiedDataSerializable):
    # omitting the implementation
    pass


client = HazelcastClient(data_serializable_factories={
    FACTORY_ID: {
        CLASS_ID: SomeSerializable
    }
})
```

- **portable_factories** (*dict[int, dict[int, class]]*) – Dictionary of factory id and corresponding *hazelcast.serialization.api.Portable* factories. A factory is simply a dictionary with class id and callable class constructors.

```
FACTORY_ID = 2
CLASS_ID = 2

class SomeSerializable(Portable):
    # omitting the implementation
    pass


client = HazelcastClient(portable_factories={
    FACTORY_ID: {
        CLASS_ID: SomeSerializable
    }
})
```

- **class_definitions** (*list[hazelcast.serialization.portable.classdef.ClassDefinition]*) – List of all portable class definitions.
- **check_class_definition_errors** (*bool*) – When enabled, serialization system will check for class definitions error at start and throw an `HazelcastSerializationError` with error definition. By default, set to `True`.
- **is_big_endian** (*bool*) – Defines if big-endian is used as the byte order for the serialization. By default, set to `True`.
- **default_int_type** (*int | str*) – Defines how the `int`/`long` type is represented on the cluster side. By default, it is serialized as `INT` (`32` bits). See the *hazelcast.config.IntType* for possible values.

- **global_serializer** (hazelcast.serialization.api.StreamSerializer) – Defines the global serializer. This serializer is registered as a fallback serializer to handle all other objects if a serializer cannot be located for them.
- **custom_serializers** (*dict[class, hazelcast.serialization.api.StreamSerializer]*) – Dictionary of class and the corresponding custom serializers.

```
class SomeClass(object):
    # omitting the implementation
    pass


class SomeClassSerializer(StreamSerializer):
    # omitting the implementation
    pass

client = HazelcastClient(custom_serializers={
    SomeClass: SomeClassSerializer
})
```

- **near_caches** (*dict[str, dict[str, any]]*) – Dictionary of near cache names and the corresponding near cache configurations as a dictionary. The near cache configurations contains the following options. When an option is missing from the configuration, it will be set to its default value.
  - **invalidate_on_change** (bool): Enables cluster-assisted invalidate on change behavior. When set to `True`, entries are invalidated when they are changed in cluster. By default, set to `True`.
  - **in_memory_format** (int|str): Specifies in which format data will be stored in the Near Cache. By default, set to `BINARY`. See the *hazelcast.config.InMemoryFormat* for possible values.
  - **time_to_live** (float): Maximum number of seconds that an entry can stay in cache. When not set, entries won't be evicted due to expiration.
  - **max_idle** (float): Maximum number of seconds that an entry can stay in the Near Cache until it is accessed. When not set, entries won't be evicted due to inactivity.
  - **eviction_policy** (int|str): Defines eviction policy configuration. By default, set to `LRU`. See the *hazelcast.config.EvictionPolicy* for possible values.
  - **eviction_max_size** (int): Defines maximum number of entries kept in the memory before eviction kicks in. By default, set to `10000`.
  - **eviction_sampling_count** (int): Number of random entries that are evaluated to see if some of them are already expired. By default, set to `8`.
  - **eviction_sampling_pool_size** (int): Size of the pool for eviction candidates. The pool is kept sorted according to the eviction policy. By default, set to `16`.
- **load_balancer** (hazelcast.util.LoadBalancer) – Load balancer implementation for the client
- **membership_listeners** (*list[tuple[function, function]]*) – List of membership listener tuples. Tuples must be of size `2`. The first element will be the function to be called when a member is added, and the second element will be the function to be called when the member is removed with the *hazelcast.core.MemberInfo* as the only parameter. Any of the elements can be `None`, but not at the same time.

- **lifecycle_listeners** (*list[function]*) – List of lifecycle listeners. Listeners will be called with the new lifecycle state as the only parameter when the client changes lifecycle states.
- **flake_id_generators** (*dict[str, dict[str, any]]*) – Dictionary of flake id generator names and the corresponding flake id generator configurations as a dictionary. The flake id generator configurations contains the following options. When an option is missing from the configuration, it will be set to its default value.
  - **prefetch_count** (int): Defines how many IDs are pre-fetched on the background when a new flake id is requested from the cluster. Should be in the range `1..100000`. By default, set to `100`.
  - **prefetch_validity** (float): Defines for how long the pre-fetched IDs can be used. If this time elapsed, a new batch of IDs will be fetched. Time unit is in seconds. By default, set to `600` (10 minutes).

    The IDs contain timestamp component, which ensures rough global ordering of IDs. If an ID is assigned to an object that was created much later, it will be much out of order. If you don't care about ordering, set this value to `0` for unlimited ID validity.
- **labels** (*list[str]*) – Labels for the client to be sent to the cluster.
- **heartbeat_interval** (*float*) – Time interval between the heartbeats sent by the client to the member nodes in seconds. By default, set to `5.0`.
- **heartbeat_timeout** (*float*) – If there is no message passing between the client and a member within the given time via this property in seconds, the connection will be closed. By default, set to `60.0`.
- **invocation_timeout** (*float*) – When an invocation gets an exception because
  - Member throws an exception.
  - Connection between the client and member is closed.
  - Client's heartbeat requests are timed out.

  Time passed since invocation started is compared with this property. If the time is already passed, then the exception is delegated to the user. If not, the invocation is retried. Note that, if invocation gets no exception and it is a long running one, then it will not get any exception, no matter how small this timeout is set. Time unit is in seconds. By default, set to `120.0`.
- **invocation_retry_pause** (*float*) – Pause time between each retry cycle of an invocation in seconds. By default, set to `1.0`.
- **statistics_enabled** (*bool*) – When set to `True`, client statistics collection is enabled. By default, set to `False`.
- **statistics_period** (*float*) – The period in seconds the statistics run.
- **shuffle_member_list** (*bool*) – Client shuffles the given member list to prevent all clients to connect to the same node when this property is set to `True`. When it is set to `False`, the client tries to connect to the nodes in the given order. By default, set to `True`.
- **backup_ack_to_client_enabled** (*bool*) – Enables client to get backup acknowledgements directly from the member that backups are applied, which reduces number of hops and increases performance for smart clients. This option has no effect for unisocket clients. By default, set to `True` (enabled).
- **operation_backup_timeout** (*float*) – If an operation has backups, defines how long the invocation will wait for acks from the backup replicas in seconds. If acks are not

received from some backups, there won't be any rollback on other successful replicas. By default, set to `5.0`.

- **fail_on_indeterminate_operation_state** (*bool*) – When enabled, if an operation has sync backups and acks are not received from backup replicas in time, or the member which owns primary replica of the target partition leaves the cluster, then the invocation fails with *hazelcast.errors.IndeterminateOperationStateError*. However, even if the invocation fails, there will not be any rollback on other successful replicas. By default, set to `False` (do not fail).

get_executor(*name*)
Creates cluster-wide ExecutorService.

Parameters **name** (*str*) – Name of the Executor proxy.

Returns Executor proxy for the given name.

Return type *hazelcast.proxy.executor.Executor*

get_flake_id_generator(*name*)
Creates or returns a cluster-wide FlakeIdGenerator.

Parameters **name** (*str*) – Name of the FlakeIdGenerator proxy.

Returns FlakeIdGenerator proxy for the given name

Return type *hazelcast.proxy.flake_id_generator.FlakeIdGenerator*

get_queue(*name*)
Returns the distributed queue instance with the specified name.

Parameters **name** (*str*) – Name of the distributed queue.

Returns Distributed queue instance with the specified name.

Return type *hazelcast.proxy.queue.Queue*

get_list(*name*)
Returns the distributed list instance with the specified name.

Parameters **name** (*str*) – Name of the distributed list.

Returns Distributed list instance with the specified name.

Return type *hazelcast.proxy.list.List*

get_map(*name*)
Returns the distributed map instance with the specified name.

Parameters **name** (*str*) – Name of the distributed map.

Returns Distributed map instance with the specified name.

Return type *hazelcast.proxy.map.Map*

get_multi_map(*name*)
Returns the distributed MultiMap instance with the specified name.

Parameters **name** (*str*) – Name of the distributed MultiMap.

Returns Distributed MultiMap instance with the specified name.

Return type *hazelcast.proxy.multi_map.MultiMap*

get_pn_counter(*name*)
Returns the PN Counter instance with the specified name.

Parameters **name** (*str*) – Name of the PN Counter.

Returns The PN Counter.

Return type *hazelcast.proxy.pn_counter.PNCounter*

get_reliable_topic(*name*)
Returns the ReliableTopic instance with the specified name.

Parameters **name** (*str*) – Name of the ReliableTopic.

Returns The ReliableTopic.

Return type hazelcast.proxy.reliable_topic.ReliableTopic

get_replicated_map(*name*)
Returns the distributed ReplicatedMap instance with the specified name.

Parameters **name** (*str*) – Name of the distributed ReplicatedMap.

Returns Distributed ReplicatedMap instance with the specified name.

Return type *hazelcast.proxy.replicated_map.ReplicatedMap*

get_ringbuffer(*name*)
Returns the distributed Ringbuffer instance with the specified name.

Parameters **name** (*str*) – Name of the distributed Ringbuffer.

Returns Distributed RingBuffer instance with the specified name.

Return type *hazelcast.proxy.ringbuffer.Ringbuffer*

get_set(*name*)
Returns the distributed Set instance with the specified name.

Parameters **name** (*str*) – Name of the distributed Set.

Returns Distributed Set instance with the specified name.

Return type *hazelcast.proxy.set.Set*

get_topic(*name*)
Returns the Topic instance with the specified name.

Parameters **name** (*str*) – Name of the Topic.

Returns The Topic.

Return type *hazelcast.proxy.topic.Topic*

new_transaction(*timeout=120*, *durability=1*, *type=1*)
Creates a new Transaction associated with the current thread using default or given options.

Parameters

- **timeout** (*int*) – The timeout in seconds determines the maximum lifespan of a transaction. So if a transaction is configured with a timeout of 2 minutes, then it will automatically rollback if it hasn't committed yet.
- **durability** (*int*) – The durability is the number of machines that can take over if a member fails during a transaction commit or rollback.
- **type** (*int*) – The transaction type which can be `TWO_PHASE` or `ONE_PHASE`.

Returns New Transaction associated with the current thread.

Return type *hazelcast.transaction.Transaction*

add_distributed_object_listener(*listener_func*)
Adds a listener which will be notified when a new distributed object is created or destroyed.

Parameters **listener_func** (*function*) – Function to be called when a distributed object is created or destroyed.

Returns A registration id which is used as a key to remove the listener.

Return type *hazelcast.future.Future*[str]

remove_distributed_object_listener(*registration_id*)
Removes the specified distributed object listener.

Returns silently if there is no such listener added before.

Parameters **registration_id** (*str*) – The id of registered listener.

Returns `True` if registration is removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

get_distributed_objects()
Returns all distributed objects such as; queue, map, set, list, topic, lock, multimap.

Also, as a side effect, it clears the local instances of the destroyed proxies.

Returns List of instances created by Hazelcast.

Return type list[*hazelcast.proxy.base.Proxy*]

shutdown()
Shuts down this HazelcastClient.

2.2 API Documentation

2.2.1 Hazelcast Cluster

class ClusterService(*internal_cluster_service*)
Bases: `object`

Cluster service for Hazelcast clients.

It provides access to the members in the cluster and one can register for changes in the cluster members.

add_listener(*member_added=None*, *member_removed=None*, *fire_for_existing=False*)
Adds a membership listener to listen for membership updates.

It will be notified when a member is added to cluster or removed from cluster. There is no check for duplicate registrations, so if you register the listener twice, it will get events twice.

Parameters

- **member_added** (*function*) – Function to be called when a member is added to the cluster.
- **member_removed** (*function*) – Function to be called when a member is removed from the cluster.
- **fire_for_existing** (*bool*) – Whether or not fire member_added for existing members.

Returns Registration id of the listener which will be used for removing this listener.

Return type str

remove_listener(*registration_id*)
Removes the specified membership listener.

Parameters **registration_id** (*str*) – Registration id of the listener to be removed.

Returns `True` if the registration is removed, `False` otherwise.

Return type bool

get_members(*member_selector=None*)
Lists the current members in the cluster.

Every member in the cluster returns the members in the same order. To obtain the oldest member in the cluster, you can retrieve the first item in the list.

Parameters **member_selector** (*function*) – Function to filter members to return. If not provided, the returned list will contain all the available cluster members.

Returns Current members in the cluster

Return type list[*hazelcast.core.MemberInfo*]

2.2.2 Config

class IntType
Bases: `object`

Integer type options that can be used by serialization service.

VAR = 0
Integer types will be serialized as 8, 16, 32, 64 bit integers or as Java BigInteger according to their value. This option may cause problems when the Python client is used in conjunction with statically typed language clients such as Java or .NET.

BYTE = 1
Integer types will be serialized as a 8 bit integer(as Java byte)

SHORT = 2
Integer types will be serialized as a 16 bit integer(as Java short)

INT = 3
Integer types will be serialized as a 32 bit integer(as Java int)

LONG = 4
Integer types will be serialized as a 64 bit integer(as Java long)

BIG_INT = 5
Integer types will be serialized as Java BigInteger. This option can handle integer types which are less than -2^63 or greater than or equal to 2^63. However, when this option is set, serializing/de-serializing integer types is costly.

class EvictionPolicy
Bases: `object`

Near Cache eviction policy options.

NONE = 0
No eviction.

LRU = 1
Least Recently Used items will be evicted.

LFU = 2
Least frequently Used items will be evicted.

RANDOM = 3
Items will be evicted randomly.

class InMemoryFormat
Bases: object

Near Cache in memory format of the values.

BINARY = 0
As Hazelcast serialized bytearray data.

OBJECT = 1
As the actual object.

class SSLProtocol
Bases: object

SSL protocol options.

TLSv1+ requires at least Python 2.7.9 or Python 3.4 build with OpenSSL 1.0.1+ TLSv1_3 requires at least Python 2.7.15 or Python 3.7 build with OpenSSL 1.1.1+

SSLv2 = 0
SSL 2.0 Protocol. RFC 6176 prohibits SSL 2.0. Please use TLSv1+.

SSLv3 = 1
SSL 3.0 Protocol. RFC 7568 prohibits SSL 3.0. Please use TLSv1+.

TLSv1 = 2
TLS 1.0 Protocol described in RFC 2246.

TLSv1_1 = 3
TLS 1.1 Protocol described in RFC 4346.

TLSv1_2 = 4
TLS 1.2 Protocol described in RFC 5246.

TLSv1_3 = 5
TLS 1.3 Protocol described in RFC 8446.

class QueryConstants
Bases: object

Contains constants for Query.

KEY_ATTRIBUTE_NAME = '__key'
Attribute name of the key.

THIS_ATTRIBUTE_NAME = 'this'
Attribute name of the value.

class UniqueKeyTransformation
Bases: object

Defines an assortment of transformations which can be applied to unique key values.

OBJECT = 0
Extracted unique key value is interpreted as an object value. Non-negative unique ID is assigned to every distinct object value.

LONG = 1
Extracted unique key value is interpreted as a whole integer value of byte, short, int or long type. The extracted value is up casted to long (if necessary) and unique non-negative ID is assigned to every distinct value.

RAW = 2
Extracted unique key value is interpreted as a whole integer value of byte, short, int or long type. The extracted value is up casted to long (if necessary) and the resulting value is used directly as an ID.

class IndexType
Bases: `object`

Type of the index.

SORTED = 0
Sorted index. Can be used with equality and range predicates.

HASH = 1
Hash index. Can be used with equality predicates.

BITMAP = 2
Bitmap index. Can be used with equality predicates.

class ReconnectMode
Bases: `object`

Reconnect options.

OFF = 0
Prevent reconnect to cluster after a disconnect.

ON = 1
Reconnect to cluster by blocking invocations.

ASYNC = 2
Reconnect to cluster without blocking invocations. Invocations will receive ClientOfflineError

2.2.3 Core

Hazelcast Core objects and constants.

class MemberInfo(*address*, *uuid*, *attributes*, *lite_member*, *version*, **_*)
Bases: `object`

Represents a member in the cluster with its address, uuid, lite member status, attributes and version.

address
Address of the member.

Type *hazelcast.core.Address*

uuid
UUID of the member.

Type uuid.UUID

attributes
Configured attributes of the member.

Type dict[str, str]

lite_member
`True` if the member is a lite member, `False` otherwise. Lite members do not own any partition.

Type bool

version
Hazelcast codebase version of the member.

Type *hazelcast.core.MemberVersion*

class Address(*host*, *port*)
Bases: `object`

Represents an address of a member in the cluster.

host
Host of the address.

Type str

port
Port of the address.

Type int

class DistributedObjectEventType
Bases: `object`

Type of the distributed object event.

CREATED = 'CREATED'
DistributedObject is created.

DESTROYED = 'DESTROYED'
DistributedObject is destroyed.

class DistributedObjectEvent(*name*, *service_name*, *event_type*, *source*)
Bases: `object`

Distributed Object Event

name
Name of the distributed object.

Type str

service_name
Service name of the distributed object.

Type str

event_type
Event type. Either `CREATED` or `DESTROYED`.

Type str

source
UUID of the member that fired the event.

Type uuid.UUID

class SimpleEntryView(*key*, *value*, *cost*, *creation_time*, *expiration_time*, *hits*, *last_access_time*, *last_stored_time*, *last_update_time*, *version*, *ttl*, *max_idle*)
Bases: `object`

EntryView represents a readonly view of a map entry.

key
The key of the entry.

value
The value of the entry.

cost
The cost in bytes of the entry.

Type int

creation_time
The creation time of the entry.

Type int

expiration_time
The expiration time of the entry.

Type int

hits
Number of hits of the entry.

Type int

last_access_time
The last access time for the entry.

Type int

last_stored_time
The last store time for the value.

Type int

last_update_time
The last time the value was updated.

Type int

version
The version of the entry.

Type int

ttl
The last set time to live milliseconds.

Type int

max_idle
The last set max idle time in milliseconds.

Type int

class HazelcastJsonValue(*value*)
Bases: `object`

HazelcastJsonValue is a wrapper for JSON formatted strings.

It is preferred to store HazelcastJsonValue instead of Strings for JSON formatted strings. Users can run predicates and use indexes on the attributes of the underlying JSON strings.

HazelcastJsonValue is queried using Hazelcast's querying language. See Distributed Query section.

In terms of querying, numbers in JSON strings are treated as either Long or Double in the Java side. str, bool and None are treated as String, boolean and null respectively.

HazelcastJsonValue keeps given string as it is. Strings are not checked for being valid. Ill-formatted JSON strings may cause false positive or false negative results in queries.

HazelcastJsonValue can also be constructed from JSON serializable objects. In that case, objects are converted to JSON strings and stored as such. If an error occurs during the conversion, it is raised directly.

None values are not allowed.

to_string()
Returns unaltered string that was used to create this object.

Returns The original string.

Return type str

loads()
Deserializes the string that was used to create this object and returns as Python object.

Returns The Python object represented by the original string.

Return type any

class MemberVersion(*major*, *minor*, *patch*)
Bases: `object`

Determines the Hazelcast codebase version in terms of major.minor.patch version.

2.2.4 CP Subsystem

class CPSubsystem(*context*)
Bases: `object`

CP Subsystem is a component of Hazelcast that builds a strongly consistent layer for a set of distributed data structures.

Its APIs can be used for implementing distributed coordination use cases, such as leader election, distributed locking, synchronization, and metadata management.

Its data structures are CP with respect to the CAP principle, i.e., they always maintain linearizability and prefer consistency over availability during network partitions. Besides network partitions, CP Subsystem withstands server and client failures.

Data structures in CP Subsystem run in CP groups. Each CP group elects its own Raft leader and runs the Raft consensus algorithm independently.

The CP data structures differ from the other Hazelcast data structures in two aspects. First, an internal commit is performed on the METADATA CP group every time you fetch a proxy from this interface. Hence, callers should cache returned proxy objects. Second, if you call `destroy()` on a CP data structure proxy, that data structure is terminated on the underlying CP group and cannot be reinitialized until the CP group is force-destroyed. For this reason, please make sure that you are completely done with a CP data structure before destroying its proxy.

get_atomic_long(*name*)
Returns the distributed AtomicLong instance with given name.

The instance is created on CP Subsystem.

If no group name is given within the `name` argument, then the AtomicLong instance will be created on the default CP group. If a group name is given, like `.get_atomic_long("myLong@group1")`, the given group will be initialized first, if not initialized already, and then the instance will be created on this group.

Parameters **name** (*str*) – Name of the AtomicLong.

Returns The AtomicLong proxy for the given name.

Return type *hazelcast.proxy.cp.atomic_long.AtomicLong*

get_atomic_reference(*name*)
Returns the distributed AtomicReference instance with given name.

The instance is created on CP Subsystem.

If no group name is given within the `name` argument, then the AtomicLong instance will be created on the DEFAULT CP group. If a group name is given, like `.get_atomic_reference("myRef@group1")`, the given group will be initialized first, if not initialized already, and then the instance will be created on this group.

Parameters **name** (*str*) – Name of the AtomicReference.

Returns The AtomicReference proxy for the given name.

Return type *hazelcast.proxy.cp.atomic_reference.AtomicReference*

get_count_down_latch(*name*)
Returns the distributed CountDownLatch instance with given name.

The instance is created on CP Subsystem.

If no group name is given within the `name` argument, then the CountDownLatch instance will be created on the DEFAULT CP group. If a group name is given, like `.get_count_down_latch("myLatch@group1")`, the given group will be initialized first, if not initialized already, and then the instance will be created on this group.

Parameters **name** (*str*) – Name of the CountDownLatch.

Returns The CountDownLatch proxy for the given name.

Return type *hazelcast.proxy.cp.count_down_latch.CountDownLatch*

get_lock(*name*)
Returns the distributed FencedLock instance instance with given name.

The instance is created on CP Subsystem.

If no group name is given within the `name` argument, then the FencedLock instance will be created on the DEFAULT CP group. If a group name is given, like `.get_lock("myLock@group1")`, the given group will be initialized first, if not initialized already, and then the instance will be created on this group.

Parameters **name** (*str*) – Name of the FencedLock

Returns The FencedLock proxy for the given name.

Return type *hazelcast.proxy.cp.fenced_lock.FencedLock*

get_semaphore(*name*)
Returns the distributed Semaphore instance instance with given name.

The instance is created on CP Subsystem.

If no group name is given within the `name` argument, then the Semaphore instance will be created on the DEFAULT CP group. If a group name is given, like `.get_semaphore("mySemaphore@group1")`, the given group will be initialized first, if not initialized already, and then the instance will be created on this group.

Parameters **name** (*str*) – Name of the Semaphore

Returns The Semaphore proxy for the given name.

Return type *hazelcast.proxy.cp.semaphore.Semaphore*

2.2.5 Errors

exception HazelcastError(*message=None*, *cause=None*)
Bases: `Exception`

General HazelcastError class.

exception ArrayIndexOutOfBoundsError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ArrayStoreError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception AuthenticationError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception CacheNotExistsError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception CallerNotMemberError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception CancellationError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ClassCastError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ClassNotFoundError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ConcurrentModificationError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ConfigMismatchError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ConfigurationError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception DistributedObjectDestroyedError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception DuplicateInstanceNameError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastEOFError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ExecutionError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastInstanceNotActiveError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastOverloadError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastSerializationError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastIOError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception IllegalArgumentError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception IllegalAccessException(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception IllegalAccessError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception IllegalMonitorStateError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception IllegalStateError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception IllegalThreadStateError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception IndexOutOfBoundsError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastInterruptedError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception InvalidAddressError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception InvalidConfigurationError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception MemberLeftError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NegativeArraySizeError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NoSuchElementError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NotSerializableError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NullPointerError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception OperationTimeoutError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception PartitionMigratingError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception QueryError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception QueryResultSizeExceededError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception SplitBrainProtectionError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ReachedMaxSizeError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception RejectedExecutionError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ResponseAlreadySentError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception RetryableHazelcastError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception RetryableIOError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastRuntimeError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception SecurityError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception SocketError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception StaleSequenceError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception TargetDisconnectedError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception TargetNotMemberError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastTimeoutError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception TopicOverloadError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception TransactionError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception TransactionNotActiveError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception TransactionTimedOutError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception URISyntaxError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception UTFDataFormatError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception UnsupportedOperationError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception WrongTargetError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception XAError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception AccessControlError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception LoginError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception UnsupportedCallbackError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NoDataMemberInClusterError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ReplicatedMapCantBeCreatedOnLiteMemberError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception MaxMessageSizeExceededError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception WANReplicationQueueFullError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastAssertionError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception OutOfMemoryError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception StackOverflowError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NativeOutOfMemoryError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ServiceNotFoundError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception StaleTaskIdError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception DuplicateTaskError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception StaleTaskError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception LocalMemberResetError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception IndeterminateOperationStateError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NodeIdOutOfRangeError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception TargetNotReplicaError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception MutationDisallowedError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ConsistencyLostError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastClientNotActiveError(*message='Client is not active'*)
Bases: *hazelcast.errors.HazelcastError*

exception HazelcastCertificationError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception ClientOfflineError
Bases: *hazelcast.errors.HazelcastError*

exception ClientNotAllowedInClusterError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception VersionMismatchError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NoSuchMethodError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NoSuchMethodException(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NoSuchFieldError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NoSuchFieldException(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NoClassDefFoundError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception UndefinedErrorCodeError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception SessionExpiredError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception WaitKeyCancelledError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception LockAcquireLimitReachedError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception LockOwnershipLostError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception CPGroupDestroyedError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception CannotReplicateError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception LeaderDemotedError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception StaleAppendRequestError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

exception NotLeaderError(*message=None*, *cause=None*)
Bases: *hazelcast.errors.HazelcastError*

2.2.6 Future

class Future
Bases: object

Future is used for representing an asynchronous computation result.

set_result(*result*)
Sets the result of the Future.

Parameters **result** – Result of the Future.

set_exception(*exception*, *traceback=None*)
Sets the exception for this Future in case of errors.

Parameters

- **exception** (*Exception*) – Exception to be threw in case of error.
- **traceback** (*function*) – Function to be called on traceback.

result()
Returns the result of the Future, which makes the call synchronous if the result has not been computed yet.

Returns Result of the Future.

is_success()
Determines whether the result can be successfully computed or not.

done()
Determines whether the result is computed or not.

Returns True if the result is computed, False otherwise.

Return type bool

running()
Determines whether the asynchronous call, the computation is still running or not.

Returns True if the result is being computed, False otherwise.

Return type bool

exception()
Returns the exceptional result, if any.

Returns Exception of this Future.

Return type Exception

traceback()
Traceback function for the Future.

add_done_callback(*callback*)

continue_with(*continuation_func*, **args*)
Create a continuation that executes when the Future is completed.

Parameters

- **continuation_func** (*function*) – A function which takes the Future as the only parameter. Return value of the function will be set as the result of the continuation future. If the return value of the function is another Future, it will be chained to the returned Future.
- ***args** – Arguments to be passed into continuation_function.

Returns A new Future which will be completed when the continuation is done.

Return type *Future*

combine_futures(*futures*)
Combines set of Futures.

Parameters **futures** (*list[Future]*) – List of Futures to be combined.

Returns Result of the combination.

Return type *Future*

2.2.7 Lifecycle

class LifecycleState
Bases: object

Lifecycle states.

STARTING = 'STARTING'
The client is starting.

STARTED = 'STARTED'
The client has started.

CONNECTED = 'CONNECTED'
The client connected to a member.

SHUTTING_DOWN = 'SHUTTING_DOWN'
The client is shutting down.

DISCONNECTED = 'DISCONNECTED'
The client disconnected from a member.

SHUTDOWN = 'SHUTDOWN'
The client has shutdown.

class LifecycleService(*internal_lifecycle_service*)
Bases: object

Lifecycle service for the Hazelcast client. Allows to determine state of the client and add or remove lifecycle listeners.

is_running()
Checks whether or not the instance is running.

Returns True if the client is active and running, False otherwise.

Return type bool

add_listener(*on_state_change*)
Adds a listener to listen for lifecycle events.

Parameters **on_state_change** (*function*) – Function to be called when lifecycle state is changed.

Returns Registration id of the listener

Return type str

remove_listener(*registration_id*)
Removes a lifecycle listener.

Parameters **registration_id** (*str*) – The id of the listener to be removed.

Returns `True` if the listener is removed successfully, `False` otherwise.

Return type bool

2.2.8 Partition

class PartitionService(*internal_partition_service*, *serialization_service*)
Bases: `object`

Allows to retrieve information about the partition count, the partition owner or the partitionId of a key.

get_partition_owner(*partition_id*)
Returns the owner of the partition if it's set, `None` otherwise.

Parameters **partition_id** (*int*) – The partition id.

Returns Owner of the partition

Return type uuid.UUID

get_partition_id(*key*)
Returns the partition id for a key data.

Parameters **key** – The given key.

Returns The partition id.

Return type int

get_partition_count()
Returns partition count of the connected cluster.

If partition table is not fetched yet, this method returns `0`.

Returns The partition count

Return type int

2.2.9 Predicate

class Predicate
Bases: `object`

Represents a map entry predicate. Implementations of this class are basic building blocks for performing queries on map entries.

Special Attributes

The predicates that accept an attribute name support two special attributes:

- `__key` - instructs the predicate to act on the key associated with an item.
- `this` - instructs the predicate to act on the value associated with an item.

Attribute Paths

Dot notation may be used for attribute name to instruct the predicate to act on the attribute located at deeper level of an item. Given `"full_name.first_name"` path the predicate will act on `first_name` attribute of the value fetched by `full_name` attribute from the item itself. If any of the attributes along the path can't be resolved, `IllegalArgumentError` will be thrown. Reading of any attribute from `None` will produce `None` value.

Square brackets notation may be used to instruct the predicate to act on the list element at the specified index. Given `"names[0]"` path the predicate will act on the first item of the list fetched by `names` attribute from the item. The index must be non-negative, otherwise `IllegalArgumentError` will be thrown. Reading from the index pointing beyond the end of the list will produce `None` value.

Special `any` keyword may be used to act on every list element. Given `"names[any].full_name.first_name"` path the predicate will act on `first_name` attribute of the value fetched by `full_name` attribute from every list element stored in the item itself under `names` attribute.

Handling of None

The predicates that accept `None` as a value to compare with or a pattern to match against if and only if that is explicitly stated in the method documentation. In this case, the usual equality logic applies: if `None` is provided, the predicate passes an item if and only if the value stored under the item attribute in question is also `None`.

Special care must be taken while comparing with `None` values *stored* inside items being filtered through the predicates created by the following methods: *greater()*, *greater_or_equal()*, *less()*, *less_or_equal()*, *between()*. They always evaluate to `False` and therefore never pass such items.

Implicit Type Conversion

If the type of the stored value doesn't match the type of the value provided to the predicate, implicit type conversion is performed before predicate evaluation. The provided value is converted to match the type of the stored attribute value. If no conversion matching the type exists, `IllegalArgumentError` is thrown.

class PagingPredicate

Bases: *hazelcast.predicate.Predicate*

This class is a special Predicate which helps to get a page-by-page result of a query.

It can be constructed with a page-size, an inner predicate for filtering, and a comparator for sorting. This class is not thread-safe and stateless. To be able to reuse for another query, one should call *reset()*.

reset()

Resets the predicate for reuse.

next_page()

Sets page index to next page.

If new index is out of range, the query results that this paging predicate will retrieve will be an empty list.

Returns Updated page index

Return type int

previous_page()

Sets page index to previous page.

If current page index is 0, this method does nothing.

Returns Updated page index.

Return type int

property page

The current page index.

Getter Returns the current page index.

Setter Sets the current page index. If the page is out of range, the query results that this paging predicate will retrieve will be an empty list. New page index must be greater than or equal to `0`.

Type int

property page_size
The page size.

Getter Returns the page size.

Type int

sql(*expression*)
Creates a predicate that will pass items that match the given SQL `where` expression.

The following operators are supported: `=`, `<`, `>`, `<=`, `>= ==`, `!=`, `<>`, `BETWEEN`, `IN`, `LIKE`, `ILIKE`, `REGEX` `AND`, `OR`, `NOT`.

The operators are case-insensitive, but attribute names are case sensitive.

Example: `active AND (age > 20 OR salary < 60000)`

Differences to standard SQL:

- We don't use ternary boolean logic. `field=10` evaluates to `false`, if `field` is `null`, in standard SQL it evaluates to `UNKNOWN`.
- `IS [NOT] NULL` is not supported, use `=NULL` or `<>NULL`.
- `IS [NOT] DISTINCT FROM` is not supported, but = and <> behave like it.

Parameters **expression** (*str*) – The `where` expression.

Returns The created **sql** predicate instance.

Return type *Predicate*

equal(*attribute*, *value*)
Creates a predicate that will pass items if the given `value` and the value stored under the given item `attribute` are equal.

Parameters

- **attribute** (*str*) – The attribute to fetch the value for comparison from.
- **value** – The value to compare the attribute value against. Can be `None`.

Returns The created **equal** predicate instance.

Return type *Predicate*

not_equal(*attribute*, *value*)
Creates a predicate that will pass items if the given `value` and the value stored under the given item `attribute` are not equal.

Parameters

- **attribute** (*str*) – The attribute to fetch the value for comparison from.
- **value** – The value to compare the attribute value against. Can be `None`.

Returns The created **not equal** predicate instance.

Return type *Predicate*

like(*attribute*, *pattern*)
Creates a predicate that will pass items if the given `pattern` matches the value stored under the given item `attribute`.

Parameters

- **attribute** (*str*) – The attribute to fetch the value for matching from.

- **pattern** (*str*) – The pattern to match the attribute value against. The `%` (percentage sign) is a placeholder for multiple characters, the _ (underscore) is a placeholder for a single character. If you need to match the percentage sign or the underscore character itself, escape it with the backslash, for example `"\%"` string will match the percentage sign. Can be `None`.

Returns The created **like** predicate instance.

Return type *Predicate*

See also:

ilike() and *regex()*

ilike (*attribute*, *pattern*)

Creates a predicate that will pass items if the given `pattern` matches the value stored under the given item `attribute` in a case-insensitive manner.

Parameters

- **attribute** (*str*) – The attribute to fetch the value for matching from.
- **pattern** (*str*) – The pattern to match the attribute value against. The `%` (percentage sign) is a placeholder for multiple characters, the _ (underscore) is a placeholder for a single character. If you need to match the percentage sign or the underscore character itself, escape it with the backslash, for example `"\%"` string will match the percentage sign. Can be `None`.

Returns The created **case-insensitive like** predicate instance.

Return type *Predicate*

See also:

like() and *regex()*

regex (*attribute*, *pattern*)

Creates a predicate that will pass items if the given `pattern` matches the value stored under the given item `attribute`.

Parameters

- **attribute** (*str*) – The attribute to fetch the value for matching from.
- **pattern** (*str*) – The pattern to match the attribute value against. The pattern interpreted exactly the same as described in https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html. Can be `None`.

Returns The created **regex** predicate instance.

Return type *Predicate*

See also:

ilike() and *like()*

and_ (**predicates*)

Creates a predicate that will perform the logical `and` operation on the given predicates.

If no predicate is provided as argument, the created predicate will always evaluate to `true` and will pass any item.

Parameters ***predicates** (`Predicate`) – The child predicates to form the resulting `and` predicate from.

Returns The created **and** predicate instance.

Return type *Predicate*

or_(**predicates*)
Creates a predicate that will perform the logical `or` operation on the given predicates.

If no predicate is provided as argument, the created predicate will always evaluate to `false` and will never pass any items.

Parameters ***predicates** (`Predicate`) – The child predicates to form the resulting `or` predicate from.

Returns The created **or** predicate instance.

Return type *Predicate*

not_(*predicate*)
Creates a predicate that will negate the result of the given `predicate`.

Parameters **predicate** (`Predicate`) – The predicate to negate the value of.

Returns The created **not** predicate instance.

Return type *Predicate*

between(*attribute*, *from_*, *to*)
Creates a predicate that will pass items if the value stored under the given item `attribute` is contained inside the given range.

The range begins at the given `from_` bound and ends at the given `to` bound. The bounds are inclusive.

Parameters

- **attribute** (*str*) – The attribute to fetch the value to check from.
- **from** – The inclusive lower bound of the range to check.
- **to** – The inclusive upper bound of the range to check.

Returns The created **between** predicate.

Return type *Predicate*

in_(*attribute*, **values*)
Creates a predicate that will pass items if the value stored under the given item `attribute` is a member of the given `values`.

Parameters

- **attribute** (*str*) – The attribute to fetch the value to test from.
- ***values** – The values set to test the membership in. Individual values can be `None`.

Returns The created **in** predicate.

Return type *Predicate*

instance_of(*class_name*)
Creates a predicate that will pass entries for which the value class is an instance of the given `class_name`.

Parameters **class_name** (*str*) – The name of class the created predicate will check for.

Returns The created **instance of** predicate.

Return type *Predicate*

false()
Creates a predicate that will filter out all items.

Returns The created **false** predicate.

Return type *Predicate*

true()
Creates a predicate that will pass all items.

Returns The created **true** predicate.

Return type *Predicate*

paging(*predicate*, *page_size*, *comparator=None*)
Creates a paging predicate with an inner predicate, page size and comparator. Results will be filtered via inner predicate and will be ordered via comparator if provided.

Parameters

- **predicate** (Predicate) – The inner predicate through which results will be filtered. Can be None. In that case, results will not be filtered.
- **page_size** (*int*) – The page size.
- **comparator** (hazelcast.serialization.api.Portable *or* hazelcast.serialization.api.IdentifiedDataSerializable) – The comparator through which results will be ordered. The comparision logic must be defined on the server side. Can be None. In that case, the results will be returned in natural order.

Returns The created **paging** predicate.

Return type *PagingPredicate*

greater(*attribute*, *value*)
Creates a predicate that will pass items if the value stored under the given item attribute is greater than the given value.

Parameters

- **attribute** (*str*) – The left-hand side attribute to fetch the value for comparison from.
- **value** – The right-hand side value to compare the attribute value against.

Returns The created **greater than** predicate.

Return type *Predicate*

greater_or_equal(*attribute*, *value*)
Creates a predicate that will pass items if the value stored under the given item attribute is greater than or equal to the given value.

Parameters

- **attribute** (*str*) – the left-hand side attribute to fetch the value for comparison from.
- **value** – The right-hand side value to compare the attribute value against.

Returns The created **greater than or equal to** predicate.

Return type *Predicate*

less(*attribute*, *value*)
Creates a predicate that will pass items if the value stored under the given item attribute is less than the given value.

Parameters

- **attribute** (*str*) – The left-hand side attribute to fetch the value for comparison from.

- **value** – The right-hand side value to compare the attribute value against.

Returns The created **less than** predicate.

Return type *Predicate*

less_or_equal(*attribute*, *value*)
Creates a predicate that will pass items if the value stored under the given item `attribute` is less than or equal to the given `value`.

Parameters

- **attribute** (*str*) – The left-hand side attribute to fetch the value for comparison from.
- **value** – The right-hand side value to compare the attribute value against.

Returns The created **less than or equal to** predicate.

Return type *Predicate*

2.2.10 Hazelcast Proxies

Base

class Proxy(*service_name*, *name*, *context*)
Bases: `object`

Provides basic functionality for Hazelcast Proxies.

destroy()
Destroys this proxy.

Returns `True` if this proxy is destroyed successfully, `False` otherwise.

Return type bool

blocking()
Returns a version of this proxy with only blocking method calls.

class PartitionSpecificProxy(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.Proxy*

Provides basic functionality for Partition Specific Proxies.

class TransactionalProxy(*name*, *transaction*, *context*)
Bases: `object`

Provides an interface for all transactional distributed objects.

class ItemEventType
Bases: `object`

Type of item events.

ADDED = 1
Fired when an item is added.

REMOVED = 2
Fired when an item is removed.

class EntryEventType
Bases: `object`

Type of entry event.

ADDED = 1
Fired if an entry is added.

REMOVED = 2
Fired if an entry is removed.

UPDATED = 4
Fired if an entry is updated.

EVICTED = 8
Fired if an entry is evicted.

EXPIRED = 16
Fired if an entry is expired.

EVICT_ALL = 32
Fired if all entries are evicted.

CLEAR_ALL = 64
Fired if all entries are cleared.

MERGED = 128
Fired if an entry is merged after a network partition.

INVALIDATION = 256
Fired if an entry is invalidated.

LOADED = 512
Fired if an entry is loaded.

class ItemEvent(*name*, *item_data*, *event_type*, *member*, *to_object*)
Bases: `object`

Map Item event.

name
Name of the proxy that fired the event.

Type str

event_type
Type of the event.

Type *ItemEventType*

member
Member that fired the event.

Type *hazelcast.core.MemberInfo*

property item
The item related to the event.

class EntryEvent(*to_object*, *key*, *value*, *old_value*, *merging_value*, *event_type*, *uuid*, *number_of_affected_entries*)
Bases: `object`

Map Entry event.

event_type
Type of the event.

Type *EntryEventType*

uuid
UUID of the member that fired the event.

Type uuid.UUID

number_of_affected_entries
Number of affected entries by this event.

Type int

property key
The key of this entry event.

property old_value
The old value of the entry event.

property value
The value of the entry event.

property merging_value
The incoming merging value of the entry event.

class TopicMessage(*name*, *message_data*, *publish_time*, *member*, *to_object*)
Bases: `object`

Topic message.

name
Name of the proxy that fired the event.

Type str

publish_time
UNIX time that the event is published as seconds.

Type int

member
Member that fired the event.

Type *hazelcast.core.MemberInfo*

property message
The message sent to Topic.

CP Proxies

AtomicLong

class AtomicLong(*context*, *group_id*, *service_name*, *proxy_name*, *object_name*)
Bases: `hazelcast.proxy.cp.BaseCPProxy`

AtomicLong is a redundant and highly available distributed counter for 64-bit integers (`long` type in Java).

It works on top of the Raft consensus algorithm. It offers linearizability during crash failures and network partitions. It is CP with respect to the CAP principle. If a network partition occurs, it remains available on at most one side of the partition.

AtomicLong implementation does not offer exactly-once / effectively-once execution semantics. It goes with at-least-once execution semantics by default and can cause an API call to be committed multiple times in case of CP member failures. It can be tuned to offer at-most-once execution semantics. Please see *fail-on-indeterminate-operation-state* server-side setting.

add_and_get(*delta*)
Atomically adds the given value to the current value.

Parameters **delta** (*int*) – The value to add to the current value.

Returns The updated value, the given value added to the current value

Return type *hazelcast.future.Future*[int]

compare_and_set(*expect*, *update*)
Atomically sets the value to the given updated value only if the current value equals the expected value.

Parameters

- **expect** (*int*) – The expected value.
- **update** (*int*) – The new value.

Returns `True` if successful; or `False` if the actual value was not equal to the expected value.

Return type *hazelcast.future.Future*[bool]

decrement_and_get()
Atomically decrements the current value by one.

Returns The updated value, the current value decremented by one.

Return type *hazelcast.future.Future*[int]

get_and_decrement()
Atomically decrements the current value by one.

Returns The old value.

Return type *hazelcast.future.Future*[int]

get()
Gets the current value.

Returns The current value.

Return type *hazelcast.future.Future*[int]

get_and_add(*delta*)
Atomically adds the given value to the current value.

Parameters **delta** (*int*) – The value to add to the current value.

Returns The old value before the add.

Return type *hazelcast.future.Future*[int]

get_and_set(*new_value*)
Atomically sets the given value and returns the old value.

Parameters **new_value** (*int*) – The new value.

Returns The old value.

Return type *hazelcast.future.Future*[int]

increment_and_get()
Atomically increments the current value by one.

Returns The updated value, the current value incremented by one.

Return type *hazelcast.future.Future*[int]

get_and_increment()
Atomically increments the current value by one.

Returns The old value.

Return type *hazelcast.future.Future*[int]

set(*new_value*)
Atomically sets the given value.

Parameters **new_value** (*int*) – The new value

Returns

Return type *hazelcast.future.Future*[None]

alter(*function*)
Alters the currently stored value by applying a function on it.

Notes

`function` must be an instance of `IdentifiedDataSerializable` or `Portable` that has a counterpart that implements the `com.hazelcast.core.IFunction` interface registered on the server-side with the actual implementation of the function to be applied.

Parameters **function** (*hazelcast.serialization.api.Portable or hazelcast.serialization.api.IdentifiedDataSerializable*) – The function that alters the currently stored value.

Returns

Return type *hazelcast.future.Future*[None]

alter_and_get(*function*)
Alters the currently stored value by applying a function on it and gets the result.

Notes

`function` must be an instance of `IdentifiedDataSerializable` or `Portable` that has a counterpart that implements the `com.hazelcast.core.IFunction` interface registered on the server-side with the actual implementation of the function to be applied.

Parameters **function** (*hazelcast.serialization.api.Portable or hazelcast.serialization.api.IdentifiedDataSerializable*) – The function that alters the currently stored value.

Returns The new value.

Return type *hazelcast.future.Future*[int]

get_and_alter(*function*)
Alters the currently stored value by applying a function on it on and gets the old value.

Notes

`function` must be an instance of `IdentifiedDataSerializable` or `Portable` that has a counterpart that implements the `com.hazelcast.core.IFunction` interface registered on the server-side with the actual implementation of the function to be applied.

Parameters **function** (*hazelcast.serialization.api.Portable or hazelcast.serialization.api.IdentifiedDataSerializable*) – The function that alters the currently stored value.

Returns The old value.

Return type *hazelcast.future.Future*[int]

apply (*function*)
Applies a function on the value, the actual stored value will not change.

Notes

`function` must be an instance of `IdentifiedDataSerializable` or `Portable` that has a counterpart that implements the `com.hazelcast.core.IFunction` interface registered on the server-side with the actual implementation of the function to be applied.

Parameters **function** (*hazelcast.serialization.api.Portable or hazelcast.serialization.api.IdentifiedDataSerializable*) – The function applied to the currently stored value.

Returns The result of the function application.

Return type *hazelcast.future.Future*[any]

AtomicReference

class AtomicReference (*context*, *group_id*, *service_name*, *proxy_name*, *object_name*)
Bases: `hazelcast.proxy.cp.BaseCPProxy`

A distributed, highly available object reference with atomic operations.

AtomicReference offers linearizability during crash failures and network partitions. It is CP with respect to the CAP principle. If a network partition occurs, it remains available on at most one side of the partition.

The following are some considerations you need to know when you use AtomicReference:

- AtomicReference works based on the byte-content and not on the object-reference. If you use the `compare_and_set()` method, do not change to the original value because its serialized content will then be different.
- All methods returning an object return a private copy. You can modify the private copy, but the rest of the world is shielded from your changes. If you want these changes to be visible to the rest of the world, you need to write the change back to the AtomicReference; but be careful about introducing a data-race.
- The in-memory format of an AtomicReference is `binary`. The receiving side does not need to have the class definition available unless it needs to be deserialized on the other side., e.g., because a method like *alter()* is executed. This deserialization is done for every call that needs to have the object instead of the binary content, so be careful with expensive object graphs that need to be deserialized.
- If you have an object with many fields or an object graph and you only need to calculate some information or need a subset of fields, you can use the *apply()* method. With the *apply()* method, the whole object does not need to be sent over the line; only the information that is relevant is sent.

IAtomicReference does not offer exactly-once / effectively-once execution semantics. It goes with at-least-once execution semantics by default and can cause an API call to be committed multiple times in case of CP member failures. It can be tuned to offer at-most-once execution semantics. Please see *fail-on-indeterminate-operation-state* server-side setting.

compare_and_set (*expect*, *update*)
Atomically sets the value to the given updated value only if the current value is equal to the expected value.

Parameters

- **expect** – The expected value.

- **update** – The new value.

Returns `True` if successful, or `False` if the actual value was not equal to the expected value.

Return type *hazelcast.future.Future*[bool]

get()
Gets the current value.

Returns The current value.

Return type *hazelcast.future.Future*[any]

set(*new_value*)
Atomically sets the given value.

Parameters **new_value** – The new value.

Returns

Return type *hazelcast.future.Future*[None]

get_and_set(*new_value*)
Gets the old value and sets the new value.

Parameters **new_value** – The new value.

Returns The old value.

Return type *hazelcast.future.Future*[any]

is_none()
Checks if the stored reference is `None`.

Returns `True` if the stored reference is `None`, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

clear()
Clears the current stored reference, so it becomes `None`.

Returns

Return type *hazelcast.future.Future*[None]

contains(*value*)
Checks if the reference contains the value.

Parameters **value** – The value to check (is allowed to be `None`).

Returns `True` if the value is found, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

alter(*function*)
Alters the currently stored reference by applying a function on it.

Notes

`function` must be an instance of `IdentifiedDataSerializable` or `Portable` that has a counterpart that implements the *com.hazelcast.core.IFunction* interface registered on the server-side with the actual implementation of the function to be applied.

Parameters **function** (`hazelcast.serialization.api.Portable` *or* `hazelcast.serialization.api.IdentifiedDataSerializable`) – The function that alters the currently stored reference.

Returns

Return type *hazelcast.future.Future*[None]

alter_and_get (*function*)
Alters the currently stored reference by applying a function on it and gets the result.

Notes

`function` must be an instance of `IdentifiedDataSerializable` or `Portable` that has a counterpart that implements the *com.hazelcast.core.IFunction* interface registered on the server-side with the actual implementation of the function to be applied.

Parameters **function** (`hazelcast.serialization.api.Portable` *or* `hazelcast.serialization.api.IdentifiedDataSerializable`) – The function that alters the currently stored reference.

Returns The new value, the result of the applied function.

Return type *hazelcast.future.Future*[any]

get_and_alter (*function*)
Alters the currently stored reference by applying a function on it on and gets the old value.

Notes

`function` must be an instance of `IdentifiedDataSerializable` or `Portable` that has a counterpart that implements the *com.hazelcast.core.IFunction* interface registered on the server-side with the actual implementation of the function to be applied.

Parameters **function** (`hazelcast.serialization.api.Portable` *or* `hazelcast.serialization.api.IdentifiedDataSerializable`) – The function that alters the currently stored reference.

Returns The old value, the value before the function is applied.

Return type *hazelcast.future.Future*[any]

apply (*function*)
Applies a function on the value, the actual stored value will not change.

Notes

`function` must be an instance of `IdentifiedDataSerializable` or `Portable` that has a counterpart that implements the *com.hazelcast.core.IFunction* interface registered on the server-side with the actual implementation of the function to be applied.

Parameters **`function`** (`hazelcast.serialization.api.Portable` *or* `hazelcast.serialization.api.IdentifiedDataSerializable`) – The function applied on the currently stored reference.

Returns The result of the function application.

Return type *hazelcast.future.Future*[any]

CountDownLatch

class `CountDownLatch`(*context*, *group_id*, *service_name*, *proxy_name*, *object_name*)

Bases: `hazelcast.proxy.cp.BaseCPProxy`

A distributed, concurrent countdown latch data structure.

CountDownLatch is a cluster-wide synchronization aid that allows one or more callers to wait until a set of operations being performed in other callers completes.

CountDownLatch count can be reset using `try_set_count()` method after a countdown has finished but not during an active count. This allows the same latch instance to be reused.

There is no `await_latch()` method to do an unbound wait since this is undesirable in a distributed application: for example, a cluster can split or the master and replicas could all die. In most cases, it is best to configure an explicit timeout so you have the ability to deal with these situations.

All of the API methods in the CountDownLatch offer the exactly-once execution semantics. For instance, even if a `count_down()` call is internally retried because of crashed Hazelcast member, the counter value is decremented only once.

`await_latch`(*timeout*)

Causes the current thread to wait until the latch has counted down to zero, or an exception is thrown, or the specified waiting time elapses.

If the current count is zero then this method returns `True`.

If the current count is greater than zero, then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of the following things happen:

- The count reaches zero due to invocations of the `count_down()` method
- This CountDownLatch instance is destroyed
- The countdown owner becomes disconnected
- The specified waiting time elapses

If the count reaches zero, then the method returns with the value `True`.

If the specified waiting time elapses then the value `False` is returned. If the time is less than or equal to zero, the method will not wait at all.

Parameters **`timeout`** (*int*) – The maximum time to wait in seconds

Returns `True` if the count reached zero, `False` if the waiting time elapsed before the count reached zero

Return type *hazelcast.future.Future*[bool]

Raises ***IllegalStateError*** – If the Hazelcast instance was shut down while waiting.

count_down()
Decrements the count of the latch, releasing all waiting threads if the count reaches zero.

If the current count is greater than zero, then it is decremented. If the new count is zero:

- All waiting threads are re-enabled for thread scheduling purposes
- Countdown owner is set to `None`.

If the current count equals zero, then nothing happens.

Returns

Return type *hazelcast.future.Future*[None]

get_count()
Returns the current count.

Returns The current count.

Return type *hazelcast.future.Future*[int]

try_set_count(*count*)
Sets the count to the given value if the current count is zero.

If count is not zero, then this method does nothing and returns `False`.

Parameters **count** (*int*) – The number of times `count_down()` must be invoked before callers can pass through `await_latch()`.

Returns `True` if the new count was set, `False` if the current count is not zero.

Return type *hazelcast.future.Future*[bool]

FencedLock

class FencedLock(*context*, *group_id*, *service_name*, *proxy_name*, *object_name*)
Bases: `hazelcast.proxy.cp.SessionAwareCPProxy`

A linearizable, distributed lock.

FencedLock is CP with respect to the CAP principle. It works on top of the Raft consensus algorithm. It offers linearizability during crash-stop failures and network partitions. If a network partition occurs, it remains available on at most one side of the partition.

FencedLock works on top of CP sessions. Please refer to CP Session IMDG documentation section for more information.

By default, FencedLock is reentrant. Once a caller acquires the lock, it can acquire the lock reentrantly as many times as it wants in a linearizable manner. You can configure the reentrancy behaviour on the member side. For instance, reentrancy can be disabled and FencedLock can work as a non-reentrant mutex. One can also set a custom reentrancy limit. When the reentrancy limit is reached, FencedLock does not block a lock call. Instead, it fails with `LockAcquireLimitReachedError` or a specified return value. Please check the locking methods to see details about the behaviour.

It is advised to use this proxy in a blocking mode. Although it is possible, non-blocking usage requires an extra care. FencedLock uses the id of the thread that makes the request to distinguish lock owners. When used in a non-blocking mode, added callbacks or continuations are not generally executed in the thread that makes the request. That causes the code below to fail most of the time since the lock is acquired on the main thread but, unlock request is done in another thread.

```
lock = client.cp_subsystem.get_lock("lock")

def cb(_):
    lock.unlock()

lock.lock().add_done_callback(cb)
```

INVALID_FENCE = 0

lock()

Acquires the lock and returns the fencing token assigned to the current thread for this lock acquire.

If the lock is acquired reentrantly, the same fencing token is returned, or the `lock()` call can fail with `LockAcquireLimitReachedError` if the lock acquire limit is already reached.

If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired.

Fencing tokens are monotonic numbers that are incremented each time the lock switches from the free state to the acquired state. They are simply used for ordering lock holders. A lock holder can pass its fencing to the shared resource to fence off previous lock holders. When this resource receives an operation, it can validate the fencing token in the operation.

Consider the following scenario where the lock is free initially

```
lock = client.cp_subsystem.get_lock("lock").blocking()
fence1 = lock.lock()  # (1)
fence2 = lock.lock()  # (2)
assert fence1 == fence2
lock.unlock()
lock.unlock()
fence3 = lock.lock()  # (3)
assert fence3 > fence1
```

In this scenario, the lock is acquired by a thread in the cluster. Then, the same thread reentrantly acquires the lock again. The fencing token returned from the second acquire is equal to the one returned from the first acquire, because of reentrancy. After the second acquire, the lock is released 2 times, hence becomes free. There is a third lock acquire here, which returns a new fencing token. Because this last lock acquire is not reentrant, its fencing token is guaranteed to be larger than the previous tokens, independent of the thread that has acquired the lock.

Returns The fencing token.

Return type *hazelcast.future.Future*[int]

Raises

- ***LockOwnershipLostError*** – If the underlying CP session was closed before the client releases the lock
- ***LockAcquireLimitReachedError*** – If the lock call is reentrant and the configured lock acquire limit is already reached.

try_lock(*timeout=0*)

Acquires the lock if it is free within the given waiting time, or already held by the current thread at the time of invocation and, the acquire limit is not exceeded, and returns the fencing token assigned to the current thread for this lock acquire.

If the lock is acquired reentrantly, the same fencing token is returned. If the lock acquire limit is exceeded, then this method immediately returns *INVALID_FENCE* that represents a failed lock attempt.

If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock is acquired by the current thread or the specified waiting time elapses.

If the specified waiting time elapses, then *INVALID_FENCE* is returned. If the time is less than or equal to zero, the method does not wait at all. By default, timeout is set to zero.

A typical usage idiom for this method would be

```
lock = client.cp_subsystem.get_lock("lock").blocking()
fence = lock.try_lock()
if fence != lock.INVALID_FENCE:
    try:
        # manipulate the protected state
    finally:
        lock.unlock()
else:
    # perform another action
```

This usage ensures that the lock is unlocked if it was acquired, and doesn't try to unlock if the lock was not acquired.

See also:

lock() function for more information about fences.

Parameters **timeout** (*int*) – The maximum time to wait for the lock in seconds.

Returns The fencing token if the lock was acquired and *INVALID_FENCE* otherwise.

Return type *hazelcast.future.Future*[int]

Raises ***LockOwnershipLostError*** – If the underlying CP session was closed before the client releases the lock

unlock()

Releases the lock if the lock is currently held by the current thread.

Returns

Return type *hazelcast.future.Future*[None]

Raises

- ***LockOwnershipLostError*** – If the underlying CP session was closed before the client releases the lock
- ***IllegalMonitorStateError*** – If the lock is not held by the current thread

is_locked()

Returns whether this lock is locked or not.

Returns `True` if this lock is locked by any thread in the cluster, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

Raises ***LockOwnershipLostError*** – If the underlying CP session was closed before the client releases the lock

is_locked_by_current_thread()

Returns whether the lock is held by the current thread or not.

Returns `True` if the lock is held by the current thread, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

Raises ***LockOwnershipLostError*** – If the underlying CP session was closed before the client releases the lock

get_lock_count()
Returns the reentrant lock count if the lock is held by any thread in the cluster.

Returns The reentrant lock count if the lock is held by any thread in the cluster

Return type *hazelcast.future.Future*[int]

Raises ***LockOwnershipLostError*** – If the underlying CP session was closed before the client releases the lock

destroy()
Destroys this proxy.

Semaphore

class Semaphore(*context*, *group_id*, *service_name*, *proxy_name*, *object_name*)
Bases: `hazelcast.proxy.cp.BaseCPProxy`

A linearizable, distributed semaphore.

Semaphores are often used to restrict the number of callers that can access some physical or logical resource.

Semaphore is a cluster-wide counting semaphore. Conceptually, it maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Dually, each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the semaphore just keeps a count of the number available and acts accordingly.

Hazelcast's distributed semaphore implementation guarantees that callers invoking any of the `acquire()` methods are selected to obtain permits in the order of their invocations (first-in-first-out; FIFO). Note that FIFO ordering implies the order which the primary replica of an Semaphore receives these acquire requests. Therefore, it is possible for one member to invoke `acquire()` before another member, but its request hits the primary replica after the other member.

This class also provides convenient ways to work with multiple permits at once. Beware of the increased risk of indefinite postponement when using the multiple-permit acquire. If permits are released one by one, a caller waiting for one permit will acquire it before a caller waiting for multiple permits regardless of the call order.

Correct usage of a semaphore is established by programming convention in the application.

It works on top of the Raft consensus algorithm. It offers linearizability during crash failures and network partitions. It is CP with respect to the CAP principle. If a network partition occurs, it remains available on at most one side of the partition.

It has 2 variations:

- The default implementation accessed via `cp_subsystem` is session-aware. In this one, when a caller makes its very first `acquire()` call, it starts a new CP session with the underlying CP group. Then, liveliness of the caller is tracked via this CP session. When the caller fails, permits acquired by this caller are automatically and safely released. However, the session-aware version comes with a limitation, that is, a client cannot release permits before acquiring them first. In other words, a client can release only the permits it has acquired earlier. It means, you can acquire a permit from one thread and release it from another thread using the same Hazelcast client, but not different instances of Hazelcast client. You can use the session-aware CP Semaphore implementation by disabling JDK compatibility via `jdk-compatible` server-side setting. Although the session-aware implementation has a minor difference to the JDK Semaphore, we think it is a better fit for distributed environments because of its safe auto-cleanup mechanism for acquired permits.

- The second implementation offered by `cp_subsystem` is sessionless. This implementation does not perform auto-cleanup of acquired permits on failures. Acquired permits are not bound to threads and permits can be released without acquiring first. However, you need to handle failed permit owners on your own. If a Hazelcast server or a client fails while holding some permits, they will not be automatically released. You can use the sessionless CP Semaphore implementation by enabling JDK compatibility via `jdk-compatible` server-side setting.

There is a subtle difference between the lock and semaphore abstractions. A lock can be assigned to at most one endpoint at a time, so we have a total order among its holders. However, permits of a semaphore can be assigned to multiple endpoints at a time, which implies that we may not have a total order among permit holders. In fact, permit holders are partially ordered. For this reason, the fencing token approach, which is explained in *FencedLock*, does not work for the semaphore abstraction. Moreover, each permit is an independent entity. Multiple permit acquires and reentrant lock acquires of a single endpoint are not equivalent. The only case where a semaphore behaves like a lock is the binary case, where the semaphore has only 1 permit. In this case, the semaphore works like a non-reentrant lock.

All of the API methods in the new CP Semaphore implementation offer the exactly-once execution semantics for the session-aware version. For instance, even if a `release()` call is internally retried because of a crashed Hazelcast member, the permit is released only once. However, this guarantee is not given for the sessionless, a.k.a, JDK-compatible CP Semaphore.

init(*permits*)

Tries to initialize this Semaphore instance with the given permit count.

Parameters **permits** (*int*) – The given permit count.

Returns `True` if the initialization succeeds, `False` if already initialized.

Return type *hazelcast.future.Future*[bool]

Raises **AssertionError** – If the `permits` is negative.

acquire(*permits=1*)

Acquires the given number of permits if they are available, and returns immediately, reducing the number of available permits by the given amount.

If insufficient permits are available then the result of the returned future is not set until one of the following things happens:

- Some other caller invokes one of the `release` methods for this semaphore, the current caller is next to be assigned permits and the number of available permits satisfies this request,
- This Semaphore instance is destroyed

Parameters **permits** (*int*) – Optional number of permits to acquire; defaults to `1` when not specified

Returns

Return type *hazelcast.future.Future*[None]

Raises **AssertionError** – If the `permits` is not positive.

available_permits()

Returns the current number of permits currently available in this semaphore.

This method is typically used for debugging and testing purposes.

Returns The number of permits available in this semaphore.

Return type *hazelcast.future.Future*[int]

drain_permits()
Acquires and returns all permits that are available at invocation time.

Returns The number of permits drained.

Return type *hazelcast.future.Future*[int]

reduce_permits(*reduction*)
Reduces the number of available permits by the indicated amount.

This method differs from `acquire` as it does not block until permits become available. Similarly, if the caller has acquired some permits, they are not released with this call.

Parameters **reduction** (*int*) – The number of permits to reduce.

Returns

Return type *hazelcast.future.Future*[None]

Raises **AssertionError** – If the `reduction` is negative.

increase_permits(*increase*)
Increases the number of available permits by the indicated amount.

If there are some callers waiting for permits to become available, they will be notified. Moreover, if the caller has acquired some permits, they are not released with this call.

Parameters **increase** (*int*) – The number of permits to increase.

Returns

Return type *hazelcast.future.Future*[None]

Raises **AssertionError** – If `increase` is negative.

release(*permits=1*)
Releases the given number of permits and increases the number of available permits by that amount.

If some callers in the cluster are blocked for acquiring permits, they will be notified.

If the underlying Semaphore implementation is non-JDK-compatible (configured via `jdk-compatible` server-side setting), then a client can only release a permit which it has acquired before. In other words, a client cannot release a permit without acquiring it first.

Otherwise, which means the underlying implementation is JDK compatible (configured via `jdk-compatible` server-side setting), there is no requirement that a client that releases a permit must have acquired that permit by calling one of the `acquire()` methods. A client can freely release a permit without acquiring it first. In this case, correct usage of a semaphore is established by programming convention in the application.

Parameters **permits** (*int*) – Optional number of permits to release; defaults to `1` when not specified.

Returns

Return type *hazelcast.future.Future*[None]

Raises

- **AssertionError** – If the `permits` is not positive.
- ***IllegalStateError*** – if the Semaphore is non-JDK-compatible and the caller does not have a permit

try_acquire(*permits=1*, *timeout=0*)
Acquires the given number of permits and returns `True`, if they become available during the given waiting time.

If permits are acquired, the number of available permits in the Semaphore instance is also reduced by the given amount.

If no sufficient permits are available, then the result of the returned future is not set until one of the following things happens:

- Permits are released by other callers, the current caller is next to be assigned permits and the number of available permits satisfies this request
- The specified waiting time elapses

Parameters

- **permits** (*int*) – The number of permits to acquire; defaults to `1` when not specified.
- **timeout** (*int*) – Optional timeout in seconds to wait for the permits; when it's not specified the operation will return immediately after the acquire attempt

Returns `True` if all permits were acquired, `False` if the waiting time elapsed before all permits could be acquired

Return type *hazelcast.future.Future*[bool]

Raises **AssertionError** – If the `permits` is not positive.

Executor

class Executor(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.Proxy*

An object that executes submitted executable tasks.

execute_on_key_owner(*key*, *task*)
Executes a task on the owner of the specified key.

Parameters

- **key** – The specified key.
- **task** – A task executed on the owner of the specified key.

Returns future representing pending completion of the task.

Return type *hazelcast.future.Future*

execute_on_member(*member*, *task*)
Executes a task on the specified member.

Parameters

- **member** (`hazelcast.core.MemberInfo`) – The specified member.
- **task** – The task executed on the specified member.

Returns Future representing pending completion of the task.

Return type *hazelcast.future.Future*

execute_on_members(*members*, *task*)
Executes a task on each of the specified members.

Parameters

- **members** (*list[hazelcast.core.MemberInfo]*) – The specified members.
- **task** – The task executed on the specified members.

Returns Futures representing pending completion of the task on each member.

Return type list[*hazelcast.future.Future*]

execute_on_all_members(*task*)
Executes a task on all of the known cluster members.

Parameters **task** – The task executed on the all of the members.

Returns Futures representing pending completion of the task on each member.

Return type list[*hazelcast.future.Future*]

is_shutdown()
Determines whether this executor has been shutdown or not.

Returns `True` if the executor has been shutdown, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

shutdown()
Initiates a shutdown process which works orderly. Tasks that were submitted before shutdown are executed but new task will not be accepted.

Returns

Return type *hazelcast.future.Future*[None]

FlakeIdGenerator

class FlakeIdGenerator(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.Proxy*

A cluster-wide unique ID generator. Generated IDs are int (long in case of the Python 2 on 32 bit architectures) values and are k-ordered (roughly ordered). IDs are in the range from 0 to 2^63 - 1.

The IDs contain timestamp component and a node ID component, which is assigned when the member joins the cluster. This allows the IDs to be ordered and unique without any coordination between members, which makes the generator safe even in split-brain scenario.

Timestamp component is in milliseconds since 1.1.2018, 0:00 UTC and has 41 bits. This caps the useful lifespan of the generator to little less than 70 years (until ~2088). The sequence component is 6 bits. If more than 64 IDs are requested in single millisecond, IDs will gracefully overflow to the next millisecond and uniqueness is guaranteed in this case. The implementation does not allow overflowing by more than 15 seconds, if IDs are requested at higher rate, the call will block. Note, however, that clients are able to generate even faster because each call goes to a different (random) member and the 64 IDs/ms limit is for single member.

Node ID overflow: It is possible to generate IDs on any member or client as long as there is at least one member with join version smaller than 2^16 in the cluster. The remedy is to restart the cluster: nodeId will be assigned from zero again. Uniqueness after the restart will be preserved thanks to the timestamp component.

new_id()
Generates and returns a cluster-wide unique ID.

This method goes to a random member and gets a batch of IDs, which will then be returned locally for limited time. The pre-fetch size and the validity time can be configured.

Note: Values returned from this method may not be strictly ordered.

Returns hazelcast.future.Future[int], new cluster-wide unique ID.

Raises ***HazelcastError*** – if node ID for all members in the cluster is out of valid range. See `Node ID overflow` note above.

List

class List(*service_name*, *name*, *context*)

Bases: *hazelcast.proxy.base.PartitionSpecificProxy*

Concurrent, distributed implementation of List.

The Hazelcast List is not a partitioned data-structure. So all the content of the List is stored in a single machine (and in the backup). So the List will not scale by adding more members in the cluster.

add(*item*)

Adds the specified item to the end of this list.

Parameters **item** – the specified item to be appended to this list.

Returns `True` if item is added, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

add_at(*index*, *item*)

Adds the specified item at the specific position in this list. Element in this position and following elements are shifted to the right, if any.

Parameters

- **index** (*int*) – The specified index to insert the item.
- **item** – The specified item to be inserted.

Returns

Return type *hazelcast.future.Future*[None]

add_all(*items*)

Adds all of the items in the specified collection to the end of this list.

The order of new elements is determined by the specified collection's iterator.

Parameters **items** (*list*) – The specified collection which includes the elements to be added to list.

Returns `True` if this call changed the list, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

add_all_at(*index*, *items*)

Adds all of the elements in the specified collection into this list at the specified position.

Elements in this positions and following elements are shifted to the right, if any. The order of new elements is determined by the specified collection's iterator.

Parameters

- **index** (*int*) – The specified index at which the first element of specified collection is added.

- **items** (*list*) – The specified collection which includes the elements to be added to list.

Returns `True` if this call changed the list, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

add_listener(*include_value=False*, *item_added_func=None*, *item_removed_func=None*)
Adds an item listener for this list. Listener will be notified for all list add/remove events.

Parameters

- **include_value** (*bool*) – Whether received events include the updated item or not.
- **item_added_func** (*function*) – To be called when an item is added to this list.
- **item_removed_func** (*function*) – To be called when an item is deleted from this list.

Returns A registration id which is used as a key to remove the listener.

Return type *hazelcast.future.Future*[str]

clear()
Clears the list.

List will be empty with this call.

Returns

Return type *hazelcast.future.Future*[None]

contains(*item*)
Determines whether this list contains the specified item or not.

Parameters **item** – The specified item.

Returns `True` if the specified item exists in this list, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

contains_all(*items*)
Determines whether this list contains all of the items in specified collection or not.

Parameters **items** (*list*) – The specified collection which includes the items to be searched.

Returns `True` if all of the items in specified collection exist in this list, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

get(*index*)
Returns the item which is in the specified position in this list.

Parameters **index** (*int*) – the specified index of the item to be returned.

Returns the item in the specified position in this list.

Return type *hazelcast.future.Future*[any]

get_all()
Returns all of the items in this list.

Returns All of the items in this list.

Return type *hazelcast.future.Future*[list]

iterator()
Returns an iterator over the elements in this list in proper sequence, same with `get_all`.

Returns All of the items in this list.

Return type *hazelcast.future.Future*[list]

index_of(*item*)
Returns the first index of specified items's occurrences in this list.

If specified item is not present in this list, returns -1.

Parameters **item** – The specified item to be searched for.

Returns The first index of specified items's occurrences, `-1` if item is not present in this list.

Return type *hazelcast.future.Future*[int]

is_empty()
Determines whether this list is empty or not.

Returns `True` if the list contains no elements, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

last_index_of(*item*)
Returns the last index of specified items's occurrences in this list.

If specified item is not present in this list, returns -1.

Parameters **item** – The specified item to be searched for.

Returns The last index of specified items's occurrences, `-1` if item is not present in this list.

Return type *hazelcast.future.Future*[int]

list_iterator(*index=0*)
Returns a list iterator of the elements in this list.

If an index is provided, iterator starts from this index.

Parameters **index** – (int), index of first element to be returned from the list iterator.

Returns List of the elements in this list.

Return type *hazelcast.future.Future*[list]

remove(*item*)
Removes the specified element's first occurrence from the list if it exists in this list.

Parameters **item** – The specified element.

Returns `True` if the specified element is present in this list, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove_at(*index*)
Removes the item at the specified position in this list.

Element in this position and following elements are shifted to the left, if any.

Parameters **index** (*int*) – Index of the item to be removed.

Returns The item previously at the specified index.

Return type *hazelcast.future.Future*[any]

remove_all(*items*)
Removes all of the elements that is present in the specified collection from this list.

Parameters **items** (*list*) – The specified collection.

Returns `True` if this list changed as a result of the call, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove_listener(*registration_id*)
Removes the specified item listener.

Returns silently if the specified listener was not added before.

Parameters **registration_id** (*str*) – Id of the listener to be deleted.

Returns `True` if the item listener is removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

retain_all(*items*)
Retains only the items that are contained in the specified collection.

It means, items which are not present in the specified collection are removed from this list.

Parameters **items** (*list*) – Collections which includes the elements to be retained in this list.

Returns `True` if this list changed as a result of the call, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

size()
Returns the number of elements in this list.

Returns Number of elements in this list.

Return type *hazelcast.future.Future*[int]

set_at(*index*, *item*)
Replaces the specified element with the element at the specified position in this list.

Parameters

- **index** (*int*) – Index of the item to be replaced.
- **item** – Item to be stored.

Returns the previous item in the specified index.

Return type *hazelcast.future.Future*[any]

sub_list(*from_index*, *to_index*)
Returns a sublist from this list, from from_index(inclusive) to to_index(exclusive).

The returned list is backed by this list, so non-structural changes in the returned list are reflected in this list, and vice-versa.

Parameters

- **from_index** (*int*) – The start point(inclusive) of the sub_list.
- **to_index** (*int*) – The end point(exclusive) of the sub_list.

Returns A view of the specified range within this list.

Return type *hazelcast.future.Future*[list]

Map

class Map(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.Proxy*

Hazelcast Map client proxy to access the map on the cluster.

Concurrent, distributed, observable and queryable map. This map can work both async(non-blocking) or sync(blocking). Blocking calls return the value of the call and block the execution until return value is calculated. However, async calls return `hazelcast.future.Future` and do not block execution. Result of the `hazelcast.future.Future` can be used whenever ready. A `hazelcast.future.Future`'s result can be obtained with blocking the execution by calling `future.result()`.

Example

```
>>> my_map = client.get_map("my_map").blocking()  # sync map, all operations are␣
↪blocking
>>> print("map.put", my_map.put("key", "value"))
>>> print("map.contains_key", my_map.contains_key("key"))
>>> print("map.get", my_map.get("key"))
>>> print("map.size", my_map.size())
```

Example

```
>>> my_map = client.get_map("map")  # async map, all operations are non-blocking
>>> def put_callback(f):
>>>     print("map.put", f.result())
>>> my_map.put("key", "async_val").add_done_callback(put_callback)
>>>
>>> print("map.size", my_map.size().result())
>>>
>>> def contains_key_callback(f):
>>>     print("map.contains_key", f.result())
>>> my_map.contains_key("key").add_done_callback(contains_key_callback)
```

This class does not allow `None` to be used as a key or value.

add_entry_listener(*include_value=False*, *key=None*, *predicate=None*, *added_func=None*, *removed_func=None*, *updated_func=None*, *evicted_func=None*, *evict_all_func=None*, *clear_all_func=None*, *merged_func=None*, *expired_func=None*, *loaded_func=None*)
Adds a continuous entry listener for this map.

Listener will get notified for map events filtered with given parameters.

Parameters

- **include_value** (*bool*) – Whether received event should include the value or not.
- **key** – Key for filtering the events.
- **predicate** (`hazelcast.predicate.Predicate`) – Predicate for filtering the events.
- **added_func** (*function*) – Function to be called when an entry is added to map.
- **removed_func** (*function*) – Function to be called when an entry is removed from map.

- **updated_func** (*function*) – Function to be called when an entry is updated.
- **evicted_func** (*function*) – Function to be called when an entry is evicted from map.
- **evict_all_func** (*function*) – Function to be called when entries are evicted from map.
- **clear_all_func** (*function*) – Function to be called when entries are cleared from map.
- **merged_func** (*function*) – Function to be called when WAN replicated entry is merged_func.
- **expired_func** (*function*) – Function to be called when an entry's live time is expired.
- **loaded_func** (*function*) – Function to be called when an entry is loaded from a map loader.

Returns A registration id which is used as a key to remove the listener.

Return type *hazelcast.future.Future*[str]

add_index(*attributes=None*, *index_type=0*, *name=None*, *bitmap_index_options=None*)
Adds an index to this map for the specified entries so that queries can run faster.

Example

Let's say your map values are Employee objects.

```
>>> class Employee(IdentifiedDataSerializable):
>>>     active = false
>>>     age = None
>>>     name = None
>>>     #other fields
>>>
>>>     #methods
```

If you query your values mostly based on age and active fields, you should consider indexing these.

```
>>> employees = client.get_map("employees")
>>> employees.add_index(attributes=["age"]) # Sorted index for range queries
>>> employees.add_index(attributes=["active"], index_type=IndexType.HASH)) #␣
↪Hash index for equality predicates
```

Index attribute should either have a getter method or be public. You should also make sure to add the indexes before adding entries to this map.

Indexing time is executed in parallel on each partition by operation threads. The Map is not blocked during this operation. The time taken in proportional to the size of the Map and the number Members.

Until the index finishes being created, any searches for the attribute will use a full Map scan, thus avoiding using a partially built index and returning incorrect results.

Parameters

- **attributes** (*list[str]*) – List of indexed attributes.
- **index_type** (*int|str*) – Type of the index. By default, set to `SORTED`. See the *hazelcast.config.IndexType* for possible values.

- **name** (*str*) – Name of the index.
- **bitmap_index_options** (*dict*) – Bitmap index options.
  - **unique_key:** (str): The unique key attribute is used as a source of values which uniquely identify each entry being inserted into an index. Defaults to `KEY_ATTRIBUTE_NAME`. See the *hazelcast.config.QueryConstants* for possible values.
  - **unique_key_transformation** (int|str): The transformation is applied to every value extracted from the unique key attribue. Defaults to `OBJECT`. See the *hazelcast.config.UniqueKeyTransformation* for possible values.

Returns

Return type *hazelcast.future.Future*[None]

add_interceptor(*interceptor*)

Adds an interceptor for this map.

Added interceptor will intercept operations and execute user defined methods.

Parameters **interceptor** – Interceptor for the map which includes user defined methods.

Returns Id of registered interceptor.

Return type *hazelcast.future.Future*[str]

clear()

Clears the map.

The `MAP_CLEARED` event is fired for any registered listeners.

Returns

Return type *hazelcast.future.Future*[None]

contains_key(*key*)

Determines whether this map contains an entry with the key.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **key** – The specified key.

Returns `True` if this map contains an entry for the specified key, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

contains_value(*value*)

Determines whether this map contains one or more keys for the specified value.

Parameters **value** – The specified value.

Returns `True` if this map contains an entry for the specified value, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

delete(*key*)

Removes the mapping for a key from this map if it is present (optional operation).

Unlike remove(object), this operation does not return the removed value, which avoids the serialization cost of the returned value. If the removed value will not be used, a delete operation is preferred over a remove operation for better performance.

The map will not contain a mapping for the specified key once the call returns.

> **Warning:** This method breaks the contract of EntryListener. When an entry is removed by delete(), it fires an `EntryEvent` with a `None` oldValue. Also, a listener with predicates will have `None` values, so only the keys can be queried via predicates.

Parameters **`key`** – Key of the mapping to be deleted.

Returns

Return type *hazelcast.future.Future*[None]

`entry_set`(*predicate=None*)

Returns a list clone of the mappings contained in this map.

> **Warning:** The list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Parameters **`predicate`** (`hazelcast.predicate.Predicate`) – Predicate for the map to filter entries.

Returns The list of key-value tuples in the map.

Return type *hazelcast.future.Future*[list]

`evict`(*key*)

Evicts the specified key from this map.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **`key`** – Key to evict.

Returns `True` if the key is evicted, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

`evict_all`()

Evicts all keys from this map except the locked ones.

The `EVICT_ALL` event is fired for any registered listeners.

Returns

Return type *hazelcast.future.Future*[None]

`execute_on_entries`(*entry_processor*, *predicate=None*)

Applies the user defined EntryProcessor to all the entries in the map or entries in the map which satisfies the predicate if provided. Returns the results mapped by each key in the map.

Parameters

- **`entry_processor`** – A stateful serializable object which represents the EntryProcessor defined on server side. This object must have a serializable EntryProcessor counter part registered on server side with the actual `com.hazelcast.map.EntryProcessor` implementation.

- **predicate** (hazelcast.predicate.Predicate) – Predicate for filtering the entries.

Returns List of map entries which includes the keys and the results of the entry process.

Return type *hazelcast.future.Future*[list]

execute_on_key(*key*, *entry_processor*)

Applies the user defined EntryProcessor to the entry mapped by the key. Returns the object which is the result of EntryProcessor's process method.

Parameters

- **key** – Specified key for the entry to be processed.
- **entry_processor** – A stateful serializable object which represents the EntryProcessor defined on server side. This object must have a serializable EntryProcessor counter part registered on server side with the actual `com.hazelcast.map.EntryProcessor` implementation.

Returns Result of entry process.

Return type *hazelcast.future.Future*[any]

execute_on_keys(*keys*, *entry_processor*)

Applies the user defined EntryProcessor to the entries mapped by the collection of keys. Returns the results mapped by each key in the collection.

Parameters

- **keys** (*list*) – Collection of the keys for the entries to be processed.
- **entry_processor** – A stateful serializable object which represents the EntryProcessor defined on server side. This object must have a serializable EntryProcessor counter part registered on server side with the actual `com.hazelcast.map.EntryProcessor` implementation.

Returns List of map entries which includes the keys and the results of the entry process.

Return type *hazelcast.future.Future*[list]

flush()

Flushes all the local dirty entries.

Returns

Return type *hazelcast.future.Future*[None]

force_unlock(*key*)

Releases the lock for the specified key regardless of the lock owner.

It always successfully unlocks the key, never blocks, and returns immediately.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **key** – The key to lock.

Returns

Return type *hazelcast.future.Future*[None]

get(*key*)
Returns the value for the specified key, or `None` if this map does not contain this key.

> **Warning:** This method returns a clone of original value, modifying the returned value does not change the actual value in the map. One should put modified value back to make changes visible to all nodes.
>
> ```
> >>> value = my_map.get(key)
> >>> value.update_some_property()
> >>> my_map.put(key,value)
> ```

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **key** – The specified key.

Returns The value for the specified key.

Return type *hazelcast.future.Future*[any]

get_all(*keys*)
Returns the entries for the given keys.

> **Warning:** The returned map is NOT backed by the original map, so changes to the original map are NOT reflected in the returned map, and vice-versa.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **keys** (*list*) – Keys to get.

Returns List of map entries.

Return type *hazelcast.future.Future*[list[tuple]]

get_entry_view(*key*)
Returns the EntryView for the specified key.

> **Warning:** This method returns a clone of original mapping, modifying the returned value does not change the actual value in the map. One should put modified value back to make changes visible to all nodes.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **key** – The key of the entry.

Returns EntryView of the specified key.

Return type *hazelcast.future.Future*[*hazelcast.core.SimpleEntryView*]

is_empty()
Returns whether this map contains no key-value mappings or not.

Returns `True` if this map contains no key-value mappings, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

is_locked(*key*)
Checks the lock for the specified key.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **key** – The key that is checked for lock

Returns `True` if lock is acquired, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

key_set(*predicate=None*)
Returns a List clone of the keys contained in this map or the keys of the entries filtered with the predicate if provided.

> **Warning:** The list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Parameters **predicate** (`hazelcast.predicate.Predicate`) –

Returns A list of the clone of the keys.

Return type *hazelcast.future.Future*[list]

load_all(*keys=None*, *replace_existing_values=True*)
Loads all keys from the store at server side or loads the given keys if provided.

Parameters

- **keys** (*list*) – Keys of the entry values to load.
- **replace_existing_values** (*bool*) – Whether the existing values will be replaced or not with those loaded from the server side MapLoader.

Returns

Return type *hazelcast.future.Future*[None]

lock(*key*, *lease_time=None*)
Acquires the lock for the specified key infinitely or for the specified lease time if provided.

If the lock is not available, the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired.

You get a lock whether the value is present in the map or not. Other threads (possibly on other systems) would block on their invoke of lock() until the non-existent key is unlocked. If the lock holder introduces the key to the map, the put() operation is not blocked. If a thread not holding a lock on the non-existent key tries to introduce the key while a lock exists on the non-existent key, the put() operation blocks until it is unlocked.

Scope of the lock is this map only. Acquired lock is only for the key in this map.

Locks are re-entrant; so, if the key is locked N times, it should be unlocked N times before another thread can acquire it.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters

- **key** – The key to lock.
- **lease_time** (*int*) – Time in seconds to wait before releasing the lock.

Returns

Return type *hazelcast.future.Future*[None]

put(*key*, *value*, *ttl=None*, *max_idle=None*)

Associates the specified value with the specified key in this map.

If the map previously contained a mapping for the key, the old value is replaced by the specified value. If ttl is provided, entry will expire and get evicted after the ttl.

> **Warning:** This method returns a clone of the previous value, not the original (identically equal) value previously put into the map.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters

- **key** – The specified key.
- **value** – The value to associate with the key.
- **ttl** (*int*) – Maximum time in seconds for this entry to stay in the map. If not provided, the value configured on the server side configuration will be used. Setting this to `0` means infinite time-to-live.
- **max_idle** (*int*) – Maximum time in seconds for this entry to stay idle in the map. If not provided, the value configured on the server side configuration will be used. Setting this to `0` means infinite max idle time.

Returns Previous value associated with key or `None` if there was no mapping for key.

Return type *hazelcast.future.Future*[any]

put_all(*map*)

Copies all of the mappings from the specified map to this map.

No atomicity guarantees are given. In the case of a failure, some of the key-value tuples may get written, while others are not.

Parameters **map** (*dict*) – Map which includes mappings to be stored in this map.

Returns

Return type *hazelcast.future.Future*[None]

put_if_absent(*key*, *value*, *ttl=None*, *max_idle=None*)

Associates the specified key with the given value if it is not already associated.

If ttl is provided, entry will expire and get evicted after the ttl.

This is equivalent to below, except that the action is performed atomically:

```
>>> if not my_map.contains_key(key):
>>>     return my_map.put(key,value)
>>> else:
>>>     return my_map.get(key)
```

Warning: This method returns a clone of the previous value, not the original (identically equal) value previously put into the map.

Warning: This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters

- **key** – Key of the entry.
- **value** – Value of the entry.
- **ttl** (*int*) – Maximum time in seconds for this entry to stay in the map. If not provided, the value configured on the server side configuration will be used. Setting this to 0 means infinite time-to-live.
- **max_idle** (*int*) – Maximum time in seconds for this entry to stay idle in the map. If not provided, the value configured on the server side configuration will be used. Setting this to 0 means infinite max idle time.

Returns Old value of the entry.

Return type *hazelcast.future.Future*[any]

put_transient(*key*, *value*, *ttl=None*, *max_idle=None*)

Same as put, but MapStore defined at the server side will not be called.

Warning: This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters

- **key** – Key of the entry.
- **value** – Value of the entry.
- **ttl** (*int*) – Maximum time in seconds for this entry to stay in the map. If not provided, the value configured on the server side configuration will be used. Setting this to 0 means infinite time-to-live.

- **max_idle** (*int*) – Maximum time in seconds for this entry to stay idle in the map. If not provided, the value configured on the server side configuration will be used. Setting this to `0` means infinite max idle time.

Returns

Return type *hazelcast.future.Future*[None]

remove(*key*)

Removes the mapping for a key from this map if it is present.

The map will not contain a mapping for the specified key once the call returns.

Warning: This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **key** – Key of the mapping to be deleted.

Returns The previous value associated with key, or `None` if there was no mapping for key.

Return type *hazelcast.future.Future*[any]

remove_if_same(*key*, *value*)

Removes the entry for a key only if it is currently mapped to a given value.

This is equivalent to below, except that the action is performed atomically:

```
>>> if my_map.contains_key(key) and my_map.get(key) == value:
>>>     my_map.remove(key)
>>>     return True
>>> else:
>>>     return False
```

Warning: This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters

- **key** – The specified key.
- **value** – Remove the key if it has this value.

Returns `True` if the value was removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove_entry_listener(*registration_id*)

Removes the specified entry listener.

Returns silently if there is no such listener added before.

Parameters **registration_id** (*str*) – Id of registered listener.

Returns `True` if registration is removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

replace(*key*, *value*)
Replaces the entry for a key only if it is currently mapped to some value.

This is equivalent to below, except that the action is performed atomically:

```
>>> if my_map.contains_key(key):
>>>     return my_map.put(key,value)
>>> else:
>>>     return None
```

Warning: This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Warning: This method returns a clone of the previous value, not the original (identically equal) value previously put into the map.

Parameters

- **key** – The specified key.
- **value** – The value to replace the previous value.

Returns Previous value associated with key, or `None` if there was no mapping for key.

Return type *hazelcast.future.Future*[any]

replace_if_same(*key*, *old_value*, *new_value*)
Replaces the entry for a key only if it is currently mapped to a given value.

This is equivalent to below, except that the action is performed atomically:

```
>>> if my_map.contains_key(key) and my_map.get(key) == old_value:
>>>     my_map.put(key, new_value)
>>>     return True
>>> else:
>>>     return False
```

Warning: This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters

- **key** – The specified key.
- **old_value** – Replace the key value if it is the old value.
- **new_value** – The new value to replace the old value.

Returns `True` if the value was replaced, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

set(*key*, *value*, *ttl=None*, *max_idle=None*)
Puts an entry into this map.

Similar to the put operation except that set doesn't return the old value, which is more efficient. If ttl is provided, entry will expire and get evicted after the ttl.

Warning: This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters

- **key** – Key of the entry.
- **value** – Value of the entry.
- **ttl** (*int*) – Maximum time in seconds for this entry to stay in the map. If not provided, the value configured on the server side configuration will be used. Setting this to `0` means infinite time-to-live.
- **max_idle** (*int*) – Maximum time in seconds for this entry to stay idle in the map. If not provided, the value configured on the server side configuration will be used. Setting this to `0` means infinite max idle time.

Returns

Return type *hazelcast.future.Future*[None]

set_ttl(*key*, *ttl*)

Updates the TTL (time to live) value of the entry specified by the given key with a new TTL value.

New TTL value is valid starting from the time this operation is invoked, not since the time the entry was created. If the entry does not exist or is already expired, this call has no effect.

Parameters

- **key** – The key of the map entry.
- **ttl** (*int*) – Maximum time in seconds for this entry to stay in the map. Setting this to `0` means infinite time-to-live.

Returns

Return type *hazelcast.future.Future*[None]

size()

Returns the number of entries in this map.

Returns Number of entries in this map.

Return type *hazelcast.future.Future*[int]

try_lock(*key*, *lease_time=None*, *timeout=0*)

Tries to acquire the lock for the specified key.

When the lock is not available:

- If the timeout is not provided, the current thread doesn't wait and returns `False` immediately.
- If the timeout is provided, the current thread becomes disabled for thread scheduling purposes and lies dormant until one of the followings happens:
  - The lock is acquired by the current thread, or
  - The specified waiting time elapses.

If the lease time is provided, lock will be released after this time elapses.

Parameters

- **key** – Key to lock in this map.
- **lease_time** (*int*) – Time in seconds to wait before releasing the lock.
- **timeout** (*int*) – Maximum time in seconds to wait for the lock.

Returns `True` if the lock was acquired, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

try_put(*key*, *value*, *timeout=0*)

Tries to put the given key and value into this map and returns immediately if timeout is not provided.

If timeout is provided, operation waits until it is completed or timeout is reached.

Parameters

- **key** – Key of the entry.
- **value** – Value of the entry.
- **timeout** (*int*) – Maximum time in seconds to wait.

Returns hazelcast.future.Future[bool] `True` if the put is successful, `False` otherwise.

try_remove(*key*, *timeout=0*)

Tries to remove the given key from this map and returns immediately if timeout is not provided.

If timeout is provided, operation waits until it is completed or timeout is reached.

Parameters

- **key** – Key of the entry to be deleted.
- **timeout** (*int*) – Maximum time in seconds to wait.

Returns `True` if the remove is successful, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

unlock(*key*)

Releases the lock for the specified key.

It never blocks and returns immediately. If the current thread is the holder of this lock, then the hold count is decremented. If the hold count is zero, then the lock is released.

Parameters **key** – The key to lock.

Returns

Return type *hazelcast.future.Future*[None]

values(*predicate=None*)

Returns a list clone of the values contained in this map or values of the entries which are filtered with the predicate if provided.

> **Warning:** The list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Parameters **predicate** (`hazelcast.predicate.Predicate`) – Predicate to filter the entries.

Returns A list of clone of the values contained in this map.

Return type *hazelcast.future.Future*[list]

MultiMap

class MultiMap(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.Proxy*

A specialized map whose keys can be associated with multiple values.

add_entry_listener(*include_value=False*, *key=None*, *added_func=None*, *removed_func=None*, *clear_all_func=None*)
Adds an entry listener for this multimap.

The listener will be notified for all multimap add/remove/clear-all events.

Parameters

- **include_value** (*bool*) – Whether received event should include the value or not.
- **key** – Key for filtering the events.
- **added_func** (*function*) – Function to be called when an entry is added to map.
- **removed_func** (*function*) – Function to be called when an entry is removed from map.
- **clear_all_func** (*function*) – Function to be called when entries are cleared from map.

Returns A registration id which is used as a key to remove the listener.

Return type *hazelcast.future.Future*[str]

contains_key(*key*)
Determines whether this multimap contains an entry with the key.

> **Warning:** This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters **key** – The specified key.

Returns `True` if this multimap contains an entry for the specified key, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

contains_value(*value*)
Determines whether this map contains one or more keys for the specified value.

Parameters **value** – The specified value.

Returns `True` if this multimap contains an entry for the specified value, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

contains_entry(*key*, *value*)
Returns whether the multimap contains an entry with the value.

Parameters

- **key** – The specified key.
- **value** – The specified value.

Returns `True` if this multimap contains the key-value tuple, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

clear()
Clears the multimap. Removes all key-value tuples.

Returns

Return type *hazelcast.future.Future*[None]

entry_set()
Returns the list of key-value tuples in the multimap.

> **Warning:** The list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Returns The list of key-value tuples in the multimap.

Return type *hazelcast.future.Future*[list]

get(*key*)
Returns the list of values associated with the key. `None` if this map does not contain this key.

> **Warning:** This method uses `__hash__` and `__eq__` of the binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in the key's class.

> **Warning:** The list is NOT backed by the multimap, so changes to the map are list reflected in the collection, and vice-versa.

Parameters **key** – The specified key.

Returns The list of the values associated with the specified key.

Return type *hazelcast.future.Future*[list]

is_locked(*key*)
Checks the lock for the specified key.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **key** – The key that is checked for lock.

Returns `True` if lock is acquired, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

force_unlock(*key*)
Releases the lock for the specified key regardless of the lock owner.

It always successfully unlocks the key, never blocks, and returns immediately.

> **Warning:** This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters **key** – The key to lock.

Returns

Return type *hazelcast.future.Future*[None]

key_set()
Returns the list of keys in the multimap.

> **Warning:** The list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Returns A list of the clone of the keys.

Return type *hazelcast.future.Future*[list]

lock(*key*, *lease_time=None*)
Acquires the lock for the specified key infinitely or for the specified lease time if provided.

If the lock is not available, the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired.

Scope of the lock is this map only. Acquired lock is only for the key in this map.

Locks are re-entrant; so, if the key is locked N times, it should be unlocked N times before another thread can acquire it.

> **Warning:** This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters

- **key** – The key to lock.
- **lease_time** (*int*) – Time in seconds to wait before releasing the lock.

Returns

Return type *hazelcast.future.Future*[None]

remove(*key*, *value*)
Removes the given key-value tuple from the multimap.

> **Warning:** This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters

- **key** – The key of the entry to remove.
- **value** – The value of the entry to remove.

Returns `True` if the size of the multimap changed after the remove operation, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove_all(*key*)
Removes all the entries with the given key and returns the value list associated with this key.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

> **Warning:** The returned list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Parameters **key** – The key of the entries to remove.

Returns The collection of removed values associated with the given key.

Return type *hazelcast.future.Future*[list]

put(*key*, *value*)
Stores a key-value tuple in the multimap.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters

- **key** – The key to be stored.
- **value** – The value to be stored.

Returns `True` if size of the multimap is increased, `False` if the multimap already contains the key-value tuple.

Return type *hazelcast.future.Future*[bool]

remove_entry_listener(*registration_id*)
Removes the specified entry listener.

Returns silently if there is no such listener added before.

Parameters **registration_id** (*str*) – Id of registered listener.

Returns `True` if registration is removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

size()
Returns the number of entries in this multimap.

Returns Number of entries in this multimap.

Return type *hazelcast.future.Future*[int]

value_count(*key*)
Returns the number of values that match the given key in the multimap.

> **Warning:** This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters **key** – The key whose values count is to be returned.

Returns The number of values that match the given key in the multimap.

Return type *hazelcast.future.Future*[int]

values()
Returns the list of values in the multimap.

> **Warning:** The returned list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Returns The list of values in the multimap.

Return type *hazelcast.future.Future*[list]

try_lock(*key*, *lease_time=None*, *timeout=0*)
Tries to acquire the lock for the specified key.

When the lock is not available:

- If the timeout is not provided, the current thread doesn't wait and returns `False` immediately.
- If the timeout is provided, the current thread becomes disabled for thread scheduling purposes and lies dormant until one of the followings happens:
  - The lock is acquired by the current thread, or
  - The specified waiting time elapses.

If the lease time is provided, lock will be released after this time elapses.

Parameters

- **key** – Key to lock in this map.
- **lease_time** (*int*) – Time in seconds to wait before releasing the lock.
- **timeout** (*int*) – Maximum time in seconds to wait for the lock.

Returns `True` if the lock was acquired, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

unlock(*key*)
Releases the lock for the specified key. It never blocks and returns immediately.

> **Warning:** This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters **key** – The key to lock.

Returns

Return type *hazelcast.future.Future*[None]

Queue

class Queue(*service_name*, *name*, *context*)

Bases: *hazelcast.proxy.base.PartitionSpecificProxy*

Concurrent, blocking, distributed, observable queue.

Queue is not a partitioned data-structure. All of the Queue content is stored in a single machine (and in the backup). Queue will not scale by adding more members in the cluster.

add(*item*)

Adds the specified item to this queue if there is available space.

Parameters **item** – The specified item.

Returns `True` if element is successfully added, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

add_all(*items*)

Adds the elements in the specified collection to this queue.

Parameters **items** (*list*) – Collection which includes the items to be added.

Returns `True` if this queue is changed after call, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

add_listener(*include_value=False*, *item_added_func=None*, *item_removed_func=None*)

Adds an item listener for this queue. Listener will be notified for all queue add/remove events.

Parameters

- **include_value** (*bool*) – Whether received events include the updated item or not.
- **item_added_func** (*function*) – Function to be called when an item is added to this set.
- **item_removed_func** (*function*) – Function to be called when an item is deleted from this set.

Returns A registration id which is used as a key to remove the listener.

Return type *hazelcast.future.Future*[str]

clear()

Clears this queue. Queue will be empty after this call.

Returns

Return type *hazelcast.future.Future*[None]

contains(*item*)

Determines whether this queue contains the specified item or not.

Parameters **item** – The specified item to be searched.

Returns `True` if the specified item exists in this queue, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

contains_all(*items*)
Determines whether this queue contains all of the items in the specified collection or not.

Parameters **items** (*list*) – The specified collection which includes the items to be searched.

Returns `True` if all of the items in the specified collection exist in this queue, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

drain_to(*target_list*, *max_size=-1*)
Transfers all available items to the given *target_list* and removes these items from this queue.

If a max_size is specified, it transfers at most the given number of items. In case of a failure, an item can exist in both collections or none of them.

This operation may be more efficient than polling elements repeatedly and putting into collection.

Parameters

- **target_list** (*list*) – the list where the items in this queue will be transferred.
- **max_size** (*int*) – The maximum number items to transfer.

Returns Number of transferred items.

Return type *hazelcast.future.Future*[int]

iterator()
Returns all of the items in this queue.

Returns Collection of items in this queue.

Return type list

is_empty()
Determines whether this set is empty or not.

Returns `True` if this queue is empty, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

offer(*item*, *timeout=0*)
Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.

If there is no space currently available:

- If the timeout is provided, it waits until this timeout elapses and returns the result.
- If the timeout is not provided, returns `False` immediately.

Parameters

- **item** – The item to be added.
- **timeout** (*int*) – Maximum time in seconds to wait for addition.

Returns `True` if the element was added to this queue, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

peek()
Retrieves the head of queue without removing it from the queue.

Returns the head of this queue, or `None` if this queue is empty.

Return type *hazelcast.future.Future*[any]

poll(*timeout=0*)
Retrieves and removes the head of this queue.

If this queue is empty:

- If the timeout is provided, it waits until this timeout elapses and returns the result.
- If the timeout is not provided, returns `None`.

Parameters **timeout** (*int*) – Maximum time in seconds to wait for addition.

Returns The head of this queue, or `None` if this queue is empty or specified timeout elapses before an item is added to the queue.

Return type *hazelcast.future.Future*[any]

put(*item*)
Adds the specified element into this queue.

If there is no space, it waits until necessary space becomes available.

Parameters **item** – The specified item.

Returns

Return type *hazelcast.future.Future*[None]

remaining_capacity()
Returns the remaining capacity of this queue.

Returns Remaining capacity of this queue.

Return type *hazelcast.future.Future*[int]

remove(*item*)
Removes the specified element from the queue if it exists.

Parameters **item** – The specified element to be removed.

Returns `True` if the specified element exists in this queue, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove_all(*items*)
Removes all of the elements of the specified collection from this queue.

Parameters **items** (*list*) – The specified collection.

Returns `True` if the call changed this queue, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove_listener(*registration_id*)
Removes the specified item listener.

Returns silently if the specified listener was not added before.

Parameters **registration_id** (*str*) – Id of the listener to be deleted.

Returns `True` if the item listener is removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

retain_all(*items*)
Removes the items which are not contained in the specified collection.

In other words, only the items that are contained in the specified collection will be retained.

Parameters **items** (*list*) – Collection which includes the elements to be retained in this set.

Returns `True` if this queue changed as a result of the call, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

size()
Returns the number of elements in this collection.

If the size is greater than `2**31 - 1`, it returns `2**31 - 1`

Returns Size of the queue.

Return type *hazelcast.future.Future*[int]

take()
Retrieves and removes the head of this queue, if necessary, waits until an item becomes available.

Returns The head of this queue.

Return type *hazelcast.future.Future*[any]

PNCounter

class PNCounter(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.Proxy*

PN (Positive-Negative) CRDT counter.

The counter supports adding and subtracting values as well as retrieving the current counter value. Each replica of this counter can perform operations locally without coordination with the other replicas, thus increasing availability. The counter guarantees that whenever two nodes have received the same set of updates, possibly in a different order, their state is identical, and any conflicting updates are merged automatically. If no new updates are made to the shared state, all nodes that can communicate will eventually have the same data.

When invoking updates from the client, the invocation is remote. This may lead to indeterminate state - the update may be applied but the response has not been received. In this case, the caller will be notified with a TargetDisconnectedError.

The read and write methods provide monotonic read and RYW (read-your-write) guarantees. These guarantees are session guarantees which means that if no replica with the previously observed state is reachable, the session guarantees are lost and the method invocation will throw a ConsistencyLostError. This does not mean that an update is lost. All of the updates are part of some replica and will be eventually reflected in the state of all other replicas. This exception just means that you cannot observe your own writes because all replicas that contain your updates are currently unreachable. After you have received a ConsistencyLostError, you can either wait for a sufficiently up-to-date replica to become reachable in which case the session can be continued or you can reset the session by calling the reset() method. If you have called the reset() method, a new session is started with the next invocation to a CRDT replica.

Notes

The CRDT state is kept entirely on non-lite (data) members. If there aren't any and the methods here are invoked on a lite member, they will fail with an NoDataMemberInClusterError.

get()
Returns the current value of the counter.

Returns The current value of the counter.

Return type *hazelcast.future.Future*[int]

Raises

- ***NoDataMemberInClusterError*** – if the cluster does not contain any data members.
- ***ConsistencyLostError*** – if the session guarantees have been lost.

get_and_add(*delta*)
Adds the given value to the current value and returns the previous value.

Parameters **delta** (*int*) – The value to add.

Returns The previous value.

Return type *hazelcast.future.Future*[int]

Raises

- ***NoDataMemberInClusterError*** – if the cluster does not contain any data members.
- ***ConsistencyLostError*** – if the session guarantees have been lost.

add_and_get(*delta*)
Adds the given value to the current value and returns the updated value.

Parameters **delta** (*int*) – The value to add.

Returns The updated value.

Return type *hazelcast.future.Future*[int]

Raises

- ***NoDataMemberInClusterError*** – if the cluster does not contain any data members.
- ***ConsistencyLostError*** – if the session guarantees have been lost.

get_and_subtract(*delta*)
Subtracts the given value from the current value and returns the previous value.

Parameters **delta** (*int*) – The value to subtract.

Returns The previous value.

Return type *hazelcast.future.Future*[int]

Raises

- ***NoDataMemberInClusterError*** – if the cluster does not contain any data members.
- ***ConsistencyLostError*** – if the session guarantees have been lost.

subtract_and_get(*delta*)
Subtracts the given value from the current value and returns the updated value.

Parameters **delta** (*int*) – The value to subtract.

Returns The updated value.

Return type *hazelcast.future.Future*[int]

Raises

- ***NoDataMemberInClusterError*** – if the cluster does not contain any data members.
- ***ConsistencyLostError*** – if the session guarantees have been lost.

get_and_decrement()
Decrements the counter value by one and returns the previous value.

Returns The previous value.

Return type *hazelcast.future.Future*[int]

Raises

- ***NoDataMemberInClusterError*** – if the cluster does not contain any data members.
- ***ConsistencyLostError*** – if the session guarantees have been lost.

decrement_and_get()
Decrements the counter value by one and returns the updated value.

Returns The updated value.

Return type *hazelcast.future.Future*[int]

Raises

- ***NoDataMemberInClusterError*** – if the cluster does not contain any data members.
- ***ConsistencyLostError*** – if the session guarantees have been lost.

get_and_increment()
Increments the counter value by one and returns the previous value.

Returns The previous value.

Return type *hazelcast.future.Future*[int]

Raises

- ***NoDataMemberInClusterError*** – if the cluster does not contain any data members.
- ***ConsistencyLostError*** – if the session guarantees have been lost.

increment_and_get()
Increments the counter value by one and returns the updated value.

Returns The updated value.

Return type *hazelcast.future.Future*[int]

Raises

- ***NoDataMemberInClusterError*** – if the cluster does not contain any data members.
- ***UnsupportedOperationError*** – if the cluster version is less than 3.10.
- ***ConsistencyLostError*** – if the session guarantees have been lost.

reset()
Resets the observed state by this PN counter.

This method may be used after a method invocation has thrown a `ConsistencyLostError` to reset the proxy and to be able to start a new session.

ReplicatedMap

class ReplicatedMap(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.Proxy*

A ReplicatedMap is a map-like data structure with weak consistency and values locally stored on every node of the cluster.

Whenever a value is written asynchronously, the new value will be internally distributed to all existing cluster members, and eventually every node will have the new value.

When a new node joins the cluster, the new node initially will request existing values from older nodes and replicate them locally.

add_entry_listener(*key=None*, *predicate=None*, *added_func=None*, *removed_func=None*, *updated_func=None*, *evicted_func=None*, *clear_all_func=None*)

Adds a continuous entry listener for this map.

Listener will get notified for map events filtered with given parameters.

Parameters

- **key** – Key for filtering the events.
- **predicate** (hazelcast.predicate.Predicate) – Predicate for filtering the events.
- **added_func** (*function*) – Function to be called when an entry is added to map.
- **removed_func** (*function*) – Function to be called when an entry is removed from map.
- **updated_func** (*function*) – Function to be called when an entry is updated.
- **evicted_func** (*function*) – Function to be called when an entry is evicted from map.
- **clear_all_func** (*function*) – Function to be called when entries are cleared from map.

Returns A registration id which is used as a key to remove the listener.

Return type *hazelcast.future.Future*[str]

clear()

Wipes data out of the replicated map.

Returns

Return type *hazelcast.future.Future*[None]

contains_key(*key*)

Determines whether this map contains an entry with the key.

> **Warning:** This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters **key** – The specified key.

Returns `True` if this map contains an entry for the specified key, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

contains_value(*value*)

Determines whether this map contains one or more keys for the specified value.

Parameters **value** – The specified value.

Returns `True` if this map contains an entry for the specified value, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

entry_set()
Returns a List clone of the mappings contained in this map.

> **Warning:** The list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Returns The list of key-value tuples in the map.

Return type *hazelcast.future.Future*[list[tuple]]

get(*key*)
Returns the value for the specified key, or `None` if this map does not contain this key.

> **Warning:** This method uses `__hash__` and `__eq__` methods of binary form of the key, not the actual implementations of `__hash__` and `__eq__` defined in key's class.

Parameters **key** – The specified key.

Returns The value associated with the specified key.

Return type *hazelcast.future.Future*[any]

is_empty()
Returns `True` if this map contains no key-value mappings.

Returns `True` if this map contains no key-value mappings.

Return type *hazelcast.future.Future*[bool]

key_set()
Returns the list of keys in the ReplicatedMap.

> **Warning:** The list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Returns A list of the clone of the keys.

Return type *hazelcast.future.Future*[list]

put(*key*, *value*, *ttl=0*)
Associates the specified value with the specified key in this map.

If the map previously contained a mapping for the key, the old value is replaced by the specified value. If ttl is provided, entry will expire and get evicted after the ttl.

Parameters

- **key** – The specified key.
- **value** – The value to associate with the key.
- **ttl** (*int*) – Maximum time in seconds for this entry to stay, if not provided, the value configured on server side configuration will be used.

Returns Previous value associated with key or `None` if there was no mapping for key.

Return type *hazelcast.future.Future*[any]

put_all(*source*)
Copies all of the mappings from the specified map to this map.

No atomicity guarantees are given. In the case of a failure, some of the key-value tuples may get written, while others are not.

Parameters **source** (*dict*) – Map which includes mappings to be stored in this map.

Returns

Return type *hazelcast.future.Future*[None]

remove(*key*)
Removes the mapping for a key from this map if it is present.

The map will not contain a mapping for the specified key once the call returns.

> **Warning:** This method uses __hash__ and __eq__ methods of binary form of the key, not the actual implementations of __hash__ and __eq__ defined in key's class.

Parameters **key** – Key of the mapping to be deleted.

Returns The previous value associated with key, or `None` if there was no mapping for key.

Return type *hazelcast.future.Future*[any]

remove_entry_listener(*registration_id*)
Removes the specified entry listener.

Returns silently if there is no such listener added before.

Parameters **registration_id** (*str*) – Id of registered listener.

Returns `True` if registration is removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

size()
Returns the number of entries in this multimap.

Returns Number of entries in this multimap.

Return type *hazelcast.future.Future*[int]

values()
Returns the list of values in the map.

> **Warning:** The returned list is NOT backed by the map, so changes to the map are NOT reflected in the list, and vice-versa.

Returns The list of values in the map.

Return type *hazelcast.future.Future*[list]

RingBuffer

OVERFLOW_POLICY_OVERWRITE = 0
Configuration property for DEFAULT overflow policy. When an item is tried to be added on full Ringbuffer, oldest item in the Ringbuffer is overwritten and item is added.

OVERFLOW_POLICY_FAIL = 1
Configuration property for overflow policy. When an item is tried to be added on full Ringbuffer, the call fails and item is not added.

The reason that FAIL exist is to give the opportunity to obey the ttl. If blocking behavior is required, this can be implemented using retrying in combination with an exponential backoff.

```
>>> sleepMS = 100;
>>> while true:
>>>     result = ringbuffer.add(item, -1)
>>>     if result != -1:
>>>         break
>>>     sleep(sleepMS / 1000)
>>>     sleepMS *= 2
```

MAX_BATCH_SIZE = 1000
The maximum number of items to be added to RingBuffer or read from RingBuffer at a time.

class Ringbuffer(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.PartitionSpecificProxy*

A Ringbuffer is a data-structure where the content is stored in a ring like structure.

A Ringbuffer has a capacity so it won't grow beyond that capacity and endanger the stability of the system. If that capacity is exceeded, than the oldest item in the Ringbuffer is overwritten. The Ringbuffer has 2 always incrementing sequences:

- Tail_sequence: This is the side where the youngest item is found. So the tail is the side of the Ringbuffer where items are added to.
- Head_sequence: This is the side where the oldest items are found. So the head is the side where items gets discarded.

The items in the Ringbuffer can be found by a sequence that is in between (inclusive) the head and tail sequence.

A Ringbuffer currently is not a distributed data-structure. So all data is stored in a single partition; comparable to the IQueue implementation. But we'll provide an option to partition the data in the near future. A Ringbuffer can be used in a similar way as a queue, but one of the key differences is that a queue.take is destructive, meaning that only 1 thread is able to take an item. A Ringbuffer.read is not destructive, so you can have multiple threads reading the same item multiple times.

capacity()
Returns the capacity of this Ringbuffer.

Returns The capacity of Ringbuffer.

Return type *hazelcast.future.Future*[int]

size()
Returns number of items in the Ringbuffer.

Returns The size of Ringbuffer.

Return type *hazelcast.future.Future*[int]

tail_sequence()
Returns the sequence of the tail.

The tail is the side of the Ringbuffer where the items are added to. The initial value of the tail is -1.

Returns The sequence of the tail.

Return type *hazelcast.future.Future*[int]

head_sequence()
Returns the sequence of the head.

The head is the side of the Ringbuffer where the oldest items in the Ringbuffer are found. If the Ringbuffer is empty, the head will be one more than the tail. The initial value of the head is 0 (1 more than tail).

Returns The sequence of the head.

Return type *hazelcast.future.Future*[int]

remaining_capacity()
Returns the remaining capacity of the Ringbuffer.

Returns The remaining capacity of Ringbuffer.

Return type *hazelcast.future.Future*[int]

add(*item*, *overflow_policy=0*)
Adds the specified item to the tail of the Ringbuffer.

If there is no space in the Ringbuffer, the action is determined by overflow policy as `OVERFLOW_POLICY_OVERWRITE` or `OVERFLOW_POLICY_FAIL`.

Parameters

- **item** – The specified item to be added.
- **overflow_policy** (*int*) – the OverflowPolicy to be used when there is no space.

Returns The sequenceId of the added item, or `-1` if the add failed.

Return type *hazelcast.future.Future*[int]

add_all(*items*, *overflow_policy=0*)
Adds all of the item in the specified collection to the tail of the Ringbuffer.

An add_all is likely to outperform multiple calls to add(object) due to better io utilization and a reduced number of executed operations. The items are added in the order of the Iterator of the collection.

If there is no space in the Ringbuffer, the action is determined by overflow policy as `OVERFLOW_POLICY_OVERWRITE` or `OVERFLOW_POLICY_FAIL`.

Parameters

- **items** (*list*) – The specified collection which contains the items to be added.
- **overflow_policy** (*int*) – The OverflowPolicy to be used when there is no space.

Returns

The sequenceId of the last written item, or `-1` of the last write is failed.

Return type *hazelcast.future.Future*[int]

read_one(*sequence*)
Reads one item from the Ringbuffer.

If the sequence is one beyond the current tail, this call blocks until an item is added. Currently it isn't possible to control how long this call is going to block.

Parameters **sequence** (*int*) – The sequence of the item to read.

Returns The read item.

read_many(*start_sequence*, *min_count*, *max_count*)
Reads a batch of items from the Ringbuffer.

If the number of available items after the first read item is smaller than the max_count, these items are returned. So it could be the number of items read is smaller than the max_count. If there are less items available than min_count, then this call blocks. Reading a batch of items is likely to perform better because less overhead is involved.

Parameters

- **start_sequence** (*int*) – The start_sequence of the first item to read.
- **min_count** (*int*) – The minimum number of items to read.
- **max_count** (*int*) – The maximum number of items to read.

Returns The list of read items.

Return type *hazelcast.future.Future*[list]

Set

class Set(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.PartitionSpecificProxy*

Concurrent, distributed implementation of Set

add(*item*)
Adds the specified item if it is not exists in this set.

Parameters **item** – The specified item to be added.

Returns `True` if this set is changed after call, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

add_all(*items*)
Adds the elements in the specified collection if they're not exist in this set.

Parameters **items** (*list*) – Collection which includes the items to be added.

Returns `True` if this set is changed after call, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

add_listener(*include_value=False*, *item_added_func=None*, *item_removed_func=None*)
Adds an item listener for this container.

Listener will be notified for all container add/remove events.

Parameters

- **include_value** (*bool*) – Whether received events include the updated item or not.
- **item_added_func** (*function*) – Function to be called when an item is added to this set.
- **item_removed_func** (*function*) – Function to be called when an item is deleted from this set.

Returns A registration id which is used as a key to remove the listener.

Return type *hazelcast.future.Future*[str]

clear()
Clears the set. Set will be empty with this call.

Returns

Return type *hazelcast.future.Future*[None]

contains(*item*)
Determines whether this set contains the specified item or not.

Parameters **item** – The specified item to be searched.

Returns `True` if the specified item exists in this set, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

contains_all(*items*)
Determines whether this set contains all of the items in the specified collection or not.

Parameters **items** (*list*) – The specified collection which includes the items to be searched.

Returns `True` if all of the items in the specified collection exist in this set, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

get_all()
Returns all of the items in the set.

Returns List of the items in this set.

Return type *hazelcast.future.Future*[list]

is_empty()
Determines whether this set is empty or not.

Returns `True` if this set is empty, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove(*item*)
Removes the specified element from the set if it exists.

Parameters **item** – The specified element to be removed.

Returns `True` if the specified element exists in this set, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove_all(*items*)
Removes all of the elements of the specified collection from this set.

Parameters **items** (*list*) – The specified collection.

Returns `True` if the call changed this set, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove_listener(*registration_id*)
Removes the specified item listener.

Returns silently if the specified listener was not added before.

Parameters **registration_id** (*str*) – Id of the listener to be deleted.

Returns `True` if the item listener is removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

retain_all(*items*)
Removes the items which are not contained in the specified collection.

In other words, only the items that are contained in the specified collection will be retained.

Parameters **items** (*list*) – Collection which includes the elements to be retained in this set.

Returns `True` if this set changed as a result of the call, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

size()
Returns the number of items in this set.

Returns Number of items in this set.

Return type *hazelcast.future.Future*[int]

Topic

class Topic(*service_name*, *name*, *context*)
Bases: *hazelcast.proxy.base.PartitionSpecificProxy*

Hazelcast provides distribution mechanism for publishing messages that are delivered to multiple subscribers, which is also known as a publish/subscribe (pub/sub) messaging model.

Publish and subscriptions are cluster-wide. When a member subscribes for a topic, it is actually registering for messages published by any member in the cluster, including the new members joined after you added the listener.

Messages are ordered, meaning that listeners(subscribers) will process the messages in the order they are actually published.

add_listener(*on_message=None*)
Subscribes to this topic.

When someone publishes a message on this topic, `on_message` function is called if provided.

Parameters **on_message** (*function*) – Function to be called when a message is published.

Returns A registration id which is used as a key to remove the listener.

Return type *hazelcast.future.Future*[str]

publish(*message*)
Publishes the message to all subscribers of this topic

Parameters **message** – The message to be published.

Returns

Return type *hazelcast.future.Future*[None]

remove_listener(*registration_id*)
Stops receiving messages for the given message listener.

If the given listener already removed, this method does nothing.

Parameters **registration_id** (*str*) – Registration id of the listener to be removed.

Returns `True` if the listener is removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

TransactionalList

class TransactionalList(*name*, *transaction*, *context*)
Bases: *hazelcast.proxy.base.TransactionalProxy*

Transactional implementation of *List*.

add(*item*)
Transactional implementation of *List.add(item)*

Parameters **item** – The new item to be added.

Returns `True` if the item is added successfully, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove(*item*)
Transactional implementation of *List.remove(item)*

Parameters **item** – The specified item to be removed.

Returns `True` if the item is removed successfully, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

size()
Transactional implementation of *List.size()*

Returns The size of the list.

Return type *hazelcast.future.Future*[int]

TransactionalMap

class TransactionalMap(*name*, *transaction*, *context*)
Bases: *hazelcast.proxy.base.TransactionalProxy*

Transactional implementation of *Map*.

contains_key(*key*)
Transactional implementation of *Map.contains_key(key)*

Parameters **key** – The specified key.

Returns `True` if this map contains an entry for the specified key, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

get(*key*)
Transactional implementation of *Map.get(key)*

Parameters **key** – The specified key.

Returns The value for the specified key.

Return type *hazelcast.future.Future*[any]

get_for_update(*key*)
Locks the key and then gets and returns the value to which the specified key is mapped.

Lock will be released at the end of the transaction (either commit or rollback).

Parameters **key** – The specified key.

Returns The value for the specified key.

Return type *hazelcast.future.Future*[any]

See also:

Map.get(key)

size()
Transactional implementation of *Map.size()*

Returns Number of entries in this map.

Return type *hazelcast.future.Future*[int]

is_empty()
Transactional implementation of *Map.is_empty()*

Returns `True` if this map contains no key-value mappings, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

put(*key*, *value*, *ttl=None*)
Transactional implementation of *Map.put(key, value, ttl)*

The object to be put will be accessible only in the current transaction context till the transaction is committed.

Parameters

- **key** – The specified key.
- **value** – The value to associate with the key.
- **ttl** (*int*) – Maximum time in seconds for this entry to stay.

Returns Previous value associated with key or `None` if there was no mapping for key.

Return type *hazelcast.future.Future*[any]

put_if_absent(*key*, *value*)
Transactional implementation of *Map.put_if_absent(key, value)*

The object to be put will be accessible only in the current transaction context till the transaction is committed.

Parameters

- **key** – Key of the entry.
- **value** – Value of the entry.

Returns Old value of the entry.

Return type *hazelcast.future.Future*[any]

set(*key*, *value*)
Transactional implementation of *Map.set(key, value)*

The object to be set will be accessible only in the current transaction context till the transaction is committed.

Parameters

- **key** – Key of the entry.
- **value** – Value of the entry.

Returns

Return type *hazelcast.future.Future*[None]

replace(*key*, *value*)
Transactional implementation of *Map.replace(key, value)*

The object to be replaced will be accessible only in the current transaction context till the transaction is committed.

Parameters

- **key** – The specified key.
- **value** – The value to replace the previous value.

Returns Previous value associated with key, or `None` if there was no mapping for key.

Return type *hazelcast.future.Future*[any]

replace_if_same(*key*, *old_value*, *new_value*)
Transactional implementation of *Map.replace_if_same(key, old_value, new_value)*

The object to be replaced will be accessible only in the current transaction context till the transaction is committed.

Parameters

- **key** – The specified key.
- **old_value** – Replace the key value if it is the old value.
- **new_value** – The new value to replace the old value.

Returns `True` if the value was replaced, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

remove(*key*)
Transactional implementation of *Map.remove(key)*

The object to be removed will be removed from only the current transaction context until the transaction is committed.

Parameters **key** – Key of the mapping to be deleted.

Returns The previous value associated with key, or `None` if there was no mapping for key.

Return type *hazelcast.future.Future*[any]

remove_if_same(*key*, *value*)
Transactional implementation of *Map.remove_if_same(key, value)*

The object to be removed will be removed from only the current transaction context until the transaction is committed.

Parameters

- **key** – The specified key.
- **value** – Remove the key if it has this value.

Returns `True` if the value was removed, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

delete(*key*)
Transactional implementation of *Map.delete(key)*

The object to be deleted will be removed from only the current transaction context until the transaction is committed.

Parameters **key** – Key of the mapping to be deleted.

Returns

Return type *hazelcast.future.Future*[None]

key_set (*predicate=None*)
Transactional implementation of *Map.key_set(predicate)*

Parameters **predicate** (hazelcast.predicate.Predicate) – Predicate to filter the entries.

Returns A list of the clone of the keys.

Return type *hazelcast.future.Future*[list]

values (*predicate=None*)
Transactional implementation of *Map.values(predicate)*

Parameters **predicate** (hazelcast.predicate.Predicate) – Predicate to filter the entries.

Returns A list of clone of the values contained in this map.

Return type *hazelcast.future.Future*[list]

TransactionalMultiMap

class TransactionalMultiMap (*name*, *transaction*, *context*)
Bases: *hazelcast.proxy.base.TransactionalProxy*

Transactional implementation of *MultiMap*.

put (*key*, *value*)
Transactional implementation of *MultiMap.put(key, value)*

Parameters

- **key** – The key to be stored.
- **value** – The value to be stored.

Returns `True` if the size of the multimap is increased, `False` if the multimap already contains the key-value tuple.

Return type *hazelcast.future.Future*[bool]

get (*key*)
Transactional implementation of *MultiMap.get(key)*

Parameters **key** – The key whose associated values are returned.

Returns The collection of the values associated with the key.

Return type *hazelcast.future.Future*[list]

remove (*key*, *value*)
Transactional implementation of *MultiMap.remove(key, value)*

Parameters

- **key** – The key of the entry to remove.
- **value** – The value of the entry to remove.

Returns `True` if the item is removed, `False` otherwise

Return type *hazelcast.future.Future*[bool]

remove_all(*key*)
Transactional implementation of *MultiMap.remove_all(key)*

Parameters **key** – The key of the entries to remove.

Returns The collection of the values associated with the key.

Return type *hazelcast.future.Future*[list]

value_count(*key*)
Transactional implementation of *MultiMap.value_count(key)*

Parameters **key** – The key whose number of values is to be returned.

Returns The number of values matching the given key in the multimap.

Return type *hazelcast.future.Future*[int]

size()
Transactional implementation of *MultiMap.size()*

Returns the number of key-value tuples in the multimap.

Return type *hazelcast.future.Future*[int]

TransactionalQueue

class TransactionalQueue(*name*, *transaction*, *context*)
Bases: *hazelcast.proxy.base.TransactionalProxy*

Transactional implementation of *Queue*.

offer(*item*, *timeout=0*)
Transactional implementation of *Queue.offer(item, timeout)*

Parameters

- **item** – The item to be added.
- **timeout** (*int*) – Maximum time in seconds to wait for addition.

Returns `True` if the element was added to this queue, `False` otherwise.

Return type *hazelcast.future.Future*[bool]

take()
Transactional implementation of *Queue.take()*

Returns The head of this queue.

Return type *hazelcast.future.Future*[any]

poll(*timeout=0*)
Transactional implementation of *Queue.poll(timeout)*

Parameters **timeout** (*int*) – Maximum time in seconds to wait for addition.

Returns The head of this queue, or `None` if this queue is empty or specified timeout elapses before an item is added to the queue.

Return type *hazelcast.future.Future*[any]

peek(*timeout=0*)
Transactional implementation of *Queue.peek(timeout)*

Parameters **timeout** (*int*) – Maximum time in seconds to wait for addition.

Returns The head of this queue, or None if this queue is empty or specified timeout elapses before an item is added to the queue.

Return type *hazelcast.future.Future*[any]

size()
Transactional implementation of *Queue.size()*

Returns Size of the queue.

Return type *hazelcast.future.Future*[int]

TransactionalSet

class TransactionalSet(*name*, *transaction*, *context*)
Bases: *hazelcast.proxy.base.TransactionalProxy*

Transactional implementation of *Set*.

add(*item*)
Transactional implementation of *Set.add(item)*

Parameters **item** – The new item to be added.

Returns True if item is added successfully, False otherwise.

Return type *hazelcast.future.Future*[bool]

remove(*item*)
Transactional implementation of *Set.remove(item)*

Parameters **item** – The specified item to be deleted.

Returns True if item is remove successfully, False otherwise.

Return type *hazelcast.future.Future*[bool]

size()
Transactional implementation of *Set.size()*

Returns Size of the set.

Return type *hazelcast.future.Future*[int]

2.2.11 Serialization

User API for Serialization.

class ObjectDataOutput
Bases: object

ObjectDataOutput provides an interface to convert any of primitive types or arrays of them to series of bytes and write them on a stream.

write_from(*buff*, *offset=None*, *length=None*)
Writes the content of the buffer to this output stream.

Parameters

- **buff** (*bytearray*) – Input buffer.
- **offset** (*int*) – Offset of the buffer where copy begin.

- **length** (*int*) – Length of data to be copied from the offset into stream.

write_boolean(*val*)
Writes a bool value to this output stream.

Single byte value 1 represent True, 0 represent False

Parameters **val** (*bool*) – The bool to be written.

write_byte(*val*)
Writes a byte value to this output stream.

Parameters **val** (*int*) – The byte value to be written.

write_short(*val*)
Writes a short value to this output stream.

Parameters **val** (*int*) – The short value to be written.

write_char(*val*)
Writes a char value to this output stream.

Parameters **val** (*str*) – The char value to be written.

write_int(*val*)
Writes a int value to this output stream.

Parameters **val** (*int*) – The int value to be written.

write_long(*val*)
Writes a long value to this output stream.

Parameters **val** (*int*) – The long value to be written.

write_float(*val*)
Writes a float value to this output stream.

Parameters **val** (*float*) – The float value to be written.

write_double(*val*)
Writes a double value to this output stream.

Parameters **val** (*float*) – The double value to be written.

write_bytes(*val*)
Writes a string to this output stream.

Parameters **val** (*str*) – The string to be written.

write_chars(*val*)
Writes every character of string to this output stream.

Parameters **val** (*str*) – The string to be written.

write_utf(*val*)
Writes UTF-8 string to this output stream.

Parameters **val** (*str*) – The UTF-8 string to be written.

write_byte_array(*val*)
Writes a byte array to this output stream.

Parameters **val** (*bytearray*) – The byte array to be written.

write_boolean_array(*val*)
Writes a bool array to this output stream.

Parameters **val** (*list[bool]*) – The bool array to be written.

write_char_array(*val*)
Writes a char array to this output stream.

Parameters **val** (*list[str]*) – The char array to be written.

write_int_array(*val*)
Writes a int array to this output stream.

Parameters **val** (*list[int]*) – The int array to be written.

write_long_array(*val*)
Writes a long array to this output stream.

Parameters **val** (*list[int]*) – The long array to be written.

write_double_array(*val*)
Writes a double array to this output stream.

Parameters **val** (*list[float]*) – The double array to be written.

write_float_array(*val*)
Writes a float array to this output stream.

Parameters **val** (*list[float]*) – The float array to be written.

write_short_array(*val*)
Writes a short array to this output stream.

Parameters **val** (*list[int]*) – The short array to be written.

write_utf_array(*val*)
Writes a UTF-8 String array to this output stream.

Parameters **val** (*list[str]*) – The UTF-8 String array to be written.

write_object(*val*)
Writes an object to this output stream.

Parameters **val** – The object to be written.

to_byte_array()
Returns a copy of internal byte array.

Returns The copy of internal byte array

Return type bytearray

get_byte_order()
Returns the order of bytes, as BIG_ENDIAN or LITTLE_ENDIAN.

Returns

Return type str

class ObjectDataInput
Bases: object

ObjectDataInput provides an interface to read bytes from a stream and reconstruct it to any of primitive types or arrays of them.

read_into(*buff*, *offset=None*, *length=None*)
Reads the content of the buffer into an array of bytes.

Parameters

- **buff** (*bytearray*) – Input buffer.
- **offset** (*int*) – Offset of the buffer where the read begin.
- **length** (*int*) – Length of data to be read.

Returns The read data.

Return type bytearray

skip_bytes(*count*)
Skips over given number of bytes from input stream.

Parameters **count** (*int*) – Number of bytes to be skipped.

Returns The actual number of bytes skipped.

Return type int

read_boolean()
Reads 1 byte from input stream and convert it to a bool value.

Returns The bool value read.

Return type bool

read_byte()
Reads 1 byte from input stream and returns it.

Returns The byte value read.

Return type int

read_unsigned_byte()
Reads 1 byte from input stream, zero-extends it and returns.

Returns The unsigned byte value read.

Return type int

read_short()
Reads 2 bytes from input stream and returns a short value.

Returns The short value read.

Return type int

read_unsigned_short()
Reads 2 bytes from input stream and returns an int value.

Returns The unsigned short value read.

Return type int

read_char()
Reads 2 bytes from the input stream and returns a str value.

Returns The char value read.

Return type str

read_int()
Reads 4 bytes from input stream and returns an int value.

Returns The int value read.

Return type int

read_long()
Reads 8 bytes from input stream and returns a long value.

Returns The int value read.

Return type int

read_float()
Reads 4 bytes from input stream and returns a float value.

Returns The float value read.

Return type float

read_double()
Reads 8 bytes from input stream and returns a double value.

Returns The double value read.

Return type float

read_utf()
Reads a UTF-8 string from input stream and returns it.

Returns The UTF-8 string read.

Return type str

read_byte_array()
Reads a byte array from input stream and returns it.

Returns The byte array read.

Return type bytearray

read_boolean_array()
Reads a bool array from input stream and returns it.

Returns The bool array read.

Return type list[bool]

read_char_array()
Reads a char array from input stream and returns it.

Returns The char array read.

Return type list[str]

read_int_array()
Reads a int array from input stream and returns it.

Returns The int array read.

Return type list[int]

read_long_array()
Reads a long array from input stream and returns it.

Returns The long array read.

Return type list[int]

read_double_array()
Reads a double array from input stream and returns it.

Returns The double array read.

Return type list[float]

read_float_array()
Reads a float array from input stream and returns it.

Returns The float array read.

Return type list[float]

read_short_array()
Reads a short array from input stream and returns it.

Returns The short array read.

Return type list[int]

read_utf_array()
Reads a UTF-8 string array from input stream and returns it.

Returns The UTF-8 string array read.

Return type list[str]

read_object()
Reads a object from input stream and returns it.

Returns The object read.

get_byte_order()
Returns the order of bytes, as BIG_ENDIAN or LITTLE_ENDIAN.

Returns

Return type str

position()
Returns current position in buffer.

Returns Current position in buffer.

Return type int

size()
Returns size of buffer.

Returns size of buffer.

Return type int

class IdentifiedDataSerializable
Bases: object

IdentifiedDataSerializable is an alternative serialization method to Python pickle, which also avoids reflection during de-serialization.

Each IdentifiedDataSerializable is created by a registered DataSerializableFactory.

write_data(*object_data_output*)
Writes object fields to output stream.

Parameters **object_data_output** (hazelcast.serialization.api.ObjectDataOutput) – The output.

read_data(*object_data_input*)
Reads fields from the input stream.

Parameters **object_data_input** (hazelcast.serialization.api.ObjectDataInput) – The input.

get_factory_id()
Returns DataSerializableFactory factory id for this class.

Returns The factory id.

Return type int

get_class_id()
Returns type identifier for this class. Id should be unique per DataSerializableFactory.

Returns The type id.

Return type int

class Portable
Bases: object

Portable provides an alternative serialization method.

Instead of relying on reflection, each Portable is created by a registered PortableFactory. Portable serialization has the following advantages:

- Support multiversion of the same object type.
- Fetching individual fields without having to rely on reflection.
- Querying and indexing support without de-serialization and/or reflection.

write_portable(*writer*)
Serialize this portable object using given PortableWriter.

Parameters **writer** (hazelcast.serialization.api.PortableWriter) – The PortableWriter.

read_portable(*reader*)
Read portable fields using PortableReader.

Parameters **reader** (hazelcast.serialization.api.PortableReader) – The PortableReader.

get_factory_id()
Returns PortableFactory id for this portable class

Returns The factory id.

Return type int

get_class_id()
Returns class identifier for this portable class. Class id should be unique per PortableFactory.

Returns The class id.

Return type int

class StreamSerializer
Bases: object

A base class for custom serialization.

write(*out*, *obj*)
Writes object to ObjectDataOutput

Parameters

- **out** (hazelcast.serialization.api.ObjectDataOutput) – Stream that object will be written to.
- **obj** – The object to be written.

read(*inp*)
Reads object from objectDataInputStream

Parameters **inp** (hazelcast.serialization.api.ObjectDataInput) – Stream that object will read from.

Returns The read object.

get_type_id()
Returns typeId of serializer.

Returns The type id of the serializer.

Return type int

destroy()
Called when instance is shutting down.

It can be used to clear used resources.

class PortableReader
Bases: object

Provides a mean of reading portable fields from binary in form of Python primitives and arrays of these primitives, nested portable fields and array of portable fields.

get_version()
Returns the global version of portable classes.

Returns Global version of portable classes.

Return type int

has_field(*field_name*)
Determine whether the field name exists in this portable class or not.

Parameters **field_name** (*str*) – name of the field (does not support nested paths).

Returns True if the field name exists in class, False otherwise.

Return type bool

get_field_names()
Returns the set of field names on this portable class.

Returns Set of field names on this portable class.

Return type set

get_field_type(*field_name*)
Returns the field type of given field name.

Parameters **field_name** (*str*) – Name of the field.

Returns The field type.

Return type hazelcast.serialization.portable.classdef.FieldType

get_field_class_id(*field_name*)
Returns the class id of given field.

Parameters **field_name** (*str*) – Name of the field.

Returns class id of given field.

Return type int

read_int(*field_name*)
Reads a primitive int.

Parameters **field_name** (*str*) – Name of the field.

Returns The int value read.

Return type int

read_long(*field_name*)
Reads a primitive long.

Parameters **field_name** (*str*) – Name of the field.

Returns The long value read.

Return type int

read_utf(*field_name*)
Reads a UTF-8 String.

Parameters **field_name** (*str*) – Name of the field.

Returns The UTF-8 String read.

Return type str

read_boolean(*field_name*)
Reads a primitive bool.

Parameters **field_name** (*str*) – Name of the field.

Returns The bool value read.

Return type bool

read_byte(*field_name*)
Reads a primitive byte.

Parameters **field_name** (*str*) – Name of the field.

Returns The byte value read.

Return type int

read_char(*field_name*)
Reads a primitive char.

Parameters **field_name** (*str*) – Name of the field.

Returns The char value read.

Return type str

read_double(*field_name*)
Reads a primitive double.

Parameters **field_name** (*str*) – Name of the field.

Returns The double value read.

Return type float

read_float(*field_name*)
Reads a primitive float.

Parameters **field_name** (*str*) – Name of the field.

Returns The float value read.

Return type float

read_short(*field_name*)
Reads a primitive short.

Parameters **field_name** (*str*) – Name of the field.

Returns The short value read.

Return type int

read_portable(*field_name*)
Reads a portable.

Parameters **field_name** (*str*) – Name of the field.

Returns The portable read.

Return type *hazelcast.serialization.api.Portable*

read_byte_array(*field_name*)
Reads a primitive byte array.

Parameters **field_name** (*str*) – Name of the field.

Returns The byte array read.

Return type bytearray

read_boolean_array(*field_name*)
Reads a primitive bool array.

Parameters **field_name** (*str*) – Name of the field.

Returns The bool array read.

Return type list[bool])

read_char_array(*field_name*)
Reads a primitive char array.

Parameters **field_name** (*str*) – Name of the field.

Returns The char array read.

Return type list[str])

read_int_array(*field_name*)
Reads a primitive int array.

Parameters **field_name** (*str*) – Name of the field.

Returns The int array read.

Return type list[int]

read_long_array(*field_name*)
Reads a primitive long array.

Parameters **field_name** (*str*) – Name of the field.

Returns The long array read.

Return type list[int]

read_double_array(*field_name*)
Reads a primitive double array.

Parameters **field_name** (*str*) – Name of the field.

Returns The double array read.

Return type list[float]

read_float_array(*field_name*)
Reads a primitive float array.

Parameters **field_name** (*str*) – Name of the field.

Returns The float array read.

Return type list[float]

read_short_array(*field_name*)
Reads a primitive short array.

Parameters **field_name** (*str*) – Name of the field.

Returns The short array read.

Return type list[int]

read_utf_array(*field_name*)
Reads a UTF-8 String array.

Parameters **field_name** (*str*) – Name of the field.

Returns The UTF-8 String array read.

Return type str

read_portable_array(*field_name*)
Reads a portable array.

Parameters **field_name** (*str*) – Name of the field.

Returns The portable array read.

Return type list[*hazelcast.serialization.api.Portable*]

get_raw_data_input()
After reading portable fields, one can read remaining fields in old fashioned way consecutively from the end of stream. After get_raw_data_input() called, no data can be read.

Returns The input.

Return type *hazelcast.serialization.api.ObjectDataInput*

class PortableWriter
Bases: object

Provides a mean of writing portable fields to a binary in form of Python primitives and arrays of these primitives, nested portable fields and array of portable fields.

write_int(*field_name*, *value*)
Writes a primitive int.

Parameters

- **field_name** (*str*) – Name of the field.
- **value** (*int*) – Int value to be written.

write_long (*field_name*, *value*)
Writes a primitive long.

Parameters

- **field_name** (*str*) – Name of the field.
- **value** (*int*) – Long value to be written.

write_utf (*field_name*, *value*)
Writes an UTF string.

Parameters

- **field_name** (*str*) – Name of the field.
- **value** (*str*) – UTF string value to be written.

write_boolean (*field_name*, *value*)
Writes a primitive bool.

Parameters

- **field_name** (*str*) – Name of the field.
- **value** (*bool*) – Bool value to be written.

write_byte (*field_name*, *value*)
Writes a primitive byte.

Parameters

- **field_name** (*str*) – Name of the field.
- **value** (*int*) – Byte value to be written.

write_char (*field_name*, *value*)
Writes a primitive char.

Parameters

- **field_name** (*str*) – Name of the field.
- **value** (*str*) – Char value to be written.

write_double (*field_name*, *value*)
Writes a primitive double.

Parameters

- **field_name** (*str*) – Name of the field.
- **value** (*float*) – Double value to be written.

write_float (*field_name*, *value*)
Writes a primitive float.

Parameters

- **field_name** (*str*) – Name of the field.
- **value** (*float*) – Float value to be written.

write_short (*field_name*, *value*)
Writes a primitive short.

Parameters

- **field_name** (*str*) – Name of the field.

- **value** (*int*) – Short value to be written.

write_portable (*field_name*, *portable*)
Writes a Portable.

Parameters

- **field_name** (*str*) – Name of the field.
- **portable** (hazelcast.serialization.api.Portable) – Portable to be written.

write_null_portable (*field_name*, *factory_id*, *class_id*)
To write a null portable value, user needs to provide class and factory ids of related class.

Parameters

- **field_name** (*str*) – Name of the field.
- **factory_id** (*int*) – Factory id of related portable class.
- **class_id** (*int*) – Class id of related portable class.

write_byte_array (*field_name*, *values*)
Writes a primitive byte array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** (*bytearray*) – Bytearray to be written.

write_boolean_array (*field_name*, *values*)
Writes a primitive bool array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** (*list[bool]*) – Bool array to be written.

write_char_array (*field_name*, *values*)
Writes a primitive char array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** (*list[str]*) – Char array to be written.

write_int_array (*field_name*, *values*)
Writes a primitive int array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** (*list[int]*) – Int array to be written.

write_long_array (*field_name*, *values*)
Writes a primitive long array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** (*list[int]*) – Long array to be written.

write_double_array (*field_name*, *values*)
Writes a primitive double array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** (*list[float]*) – Double array to be written.

write_float_array (*field_name*, *values*)
Writes a primitive float array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** (*list[float]*) – Float array to be written.

write_short_array (*field_name*, *values*)
Writes a primitive short array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** – (list[int]): Short array to be written.

write_utf_array (*field_name*, *values*)
Writes a UTF-8 String array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** – (str): UTF-8 String array to be written.

write_portable_array (*field_name*, *values*)
Writes a portable array.

Parameters

- **field_name** (*str*) – Name of the field.
- **values** (*list[hazelcast.serialization.api.Portable]*) – Portable array to be written.

get_raw_data_output ()
After writing portable fields, one can write remaining fields in old fashioned way consecutively at the end of stream. After get_raw_data_output() called, no data can be written.

Returns The output.

Return type *hazelcast.serialization.api.ObjectDataOutput*

2.2.12 Transaction

TWO_PHASE = 1
The two phase commit is separated in 2 parts. First it tries to execute the prepare; if there are any conflicts, the prepare will fail. Once the prepare has succeeded, the commit (writing the changes) can be executed.

Hazelcast also provides three phase transaction by automatically copying the backlog to another member so that in case of failure during a commit, another member can continue the commit from backup.

ONE_PHASE = 2
The one phase transaction executes a transaction using a single step at the end; committing the changes. There is no prepare of the transactions, so conflicts are not detected. If there is a conflict, then when the transaction commits the changes, some of the changes are written and others are not; leaving the system in a potentially permanent inconsistent state.

class TransactionManager(*context*)
Bases: `object`

Manages the execution of client transactions and provides Transaction objects.

new_transaction(*timeout*, *durability*, *transaction_type*)
Creates a Transaction object with given timeout, durability and transaction type.

Parameters

- **timeout** (*int*) – The timeout in seconds determines the maximum lifespan of a transaction.
- **durability** (*int*) – The durability is the number of machines that can take over if a member fails during a transaction commit or rollback
- **transaction_type** (*int*) – the transaction type which can be `hazelcast.transaction.TWO_PHASE` or `hazelcast.transaction.ONE_PHASE`

Returns New created Transaction.

Return type *hazelcast.transaction.Transaction*

class Transaction(*context*, *connection*, *timeout*, *durability*, *transaction_type*)
Bases: `object`

Provides transactional operations: beginning/committing transactions, but also retrieving transactional data-structures like the TransactionalMap.

state = 'not_started'

id = None

start_time = None

thread_id = None

begin()
Begins this transaction.

commit()
Commits this transaction.

rollback()
Rollback of this current transaction.

get_list(*name*)
Returns the transactional list instance with the specified name.

Parameters **name** (*str*) – The specified name.

Returns

The instance of Transactional List with the specified name.

Return type hazelcast.proxy.transactional_list.TransactionalList`

get_map(*name*)
Returns the transactional map instance with the specified name.

Parameters **name** (*str*) – The specified name.

Returns

The instance of Transactional Map with the specified name.

Return type *hazelcast.proxy.transactional_map.TransactionalMap*

get_multi_map(*name*)
Returns the transactional multimap instance with the specified name.

Parameters **name** (*str*) – The specified name.

Returns

The instance of Transactional MultiMap with the specified name.

Return type *hazelcast.proxy.transactional_multi_map.TransactionalMultiMap*

get_queue(*name*)
Returns the transactional queue instance with the specified name.

Parameters **name** (*str*) – The specified name.

Returns

The instance of Transactional Queue with the specified name.

Return type *hazelcast.proxy.transactional_queue.TransactionalQueue*

get_set(*name*)
Returns the transactional set instance with the specified name.

Parameters **name** (*str*) – The specified name.

Returns

The instance of Transactional Set with the specified name.

Return type *hazelcast.proxy.transactional_set.TransactionalSet*

2.2.13 Util

class LoadBalancer
Bases: object

Load balancer allows you to send operations to one of a number of endpoints (Members). It is up to the implementation to use different load balancing policies.

If the client is configured with smart routing, only the operations that are not key based will be routed to the endpoint

init(*cluster_service*)
Initializes the load balancer.

Parameters **cluster_service** (hazelcast.cluster.ClusterService) – The cluster service to select members from

next()
Returns the next member to route to.

Returns the next member or None if no member is available.

Return type *hazelcast.core.MemberInfo*

class RoundRobinLB

Bases: hazelcast.util._AbstractLoadBalancer

A load balancer implementation that relies on using round robin to a next member to send a request to.

Round robin is done based on best effort basis, the order of members for concurrent calls to the next() is not guaranteed.

next()

Returns the next member to route to.

Returns the next member or None if no member is available.

Return type *hazelcast.core.MemberInfo*

class RandomLB

Bases: hazelcast.util._AbstractLoadBalancer

A load balancer that selects a random member to route to.

next()

Returns the next member to route to.

Returns the next member or None if no member is available.

Return type *hazelcast.core.MemberInfo*

2.3 Getting Started

This chapter provides information on how to get started with your Hazelcast Python client. It outlines the requirements, installation and configuration of the client, setting up a cluster, and provides a simple application that uses a distributed map in Python client.

2.3.1 Requirements

- Windows, Linux/UNIX or Mac OS X
- Python 2.7 or Python 3.4 or newer
- Java 8 or newer
- Hazelcast IMDG 4.0 or newer
- Latest Hazelcast Python client

2.3.2 Working with Hazelcast IMDG Clusters

Hazelcast Python client requires a working Hazelcast IMDG cluster to run. This cluster handles storage and manipulation of the user data. Clients are a way to connect to the Hazelcast IMDG cluster and access such data.

Hazelcast IMDG cluster consists of one or more cluster members. These members generally run on multiple virtual or physical machines and are connected to each other via network. Any data put on the cluster is partitioned to multiple members transparent to the user. It is therefore very easy to scale the system by adding new members as the data grows. Hazelcast IMDG cluster also offers resilience. Should any hardware or software problem causes a crash to any member, the data on that member is recovered from backups and the cluster continues to operate without any downtime. Hazelcast clients are an easy way to connect to a Hazelcast IMDG cluster and perform tasks on distributed data structures that live on the cluster.

In order to use Hazelcast Python client, we first need to setup a Hazelcast IMDG cluster.

Setting Up a Hazelcast IMDG Cluster

There are following options to start a Hazelcast IMDG cluster easily:

- You can run standalone members by downloading and running JAR files from the website.
- You can embed members to your Java projects.
- You can use our Docker images.

We are going to download JARs from the website and run a standalone member for this guide.

Running Standalone JARs

Follow the instructions below to create a Hazelcast IMDG cluster:

1. Go to Hazelcast's download page and download either the `.zip` or `.tar` distribution of Hazelcast IMDG.
2. Decompress the contents into any directory that you want to run members from.
3. Change into the directory that you decompressed the Hazelcast content and then into the `bin` directory.
4. Use either `start.sh` or `start.bat` depending on your operating system. Once you run the start script, you should see the Hazelcast IMDG logs in the terminal.

You should see a log similar to the following, which means that your 1-member cluster is ready to be used:

```
Sep 03, 2020 2:21:57 PM com.hazelcast.core.LifecycleService
INFO: [192.168.1.10]:5701 [dev] [4.1-SNAPSHOT] [192.168.1.10]:5701 is STARTING
Sep 03, 2020 2:21:58 PM com.hazelcast.internal.cluster.ClusterService
INFO: [192.168.1.10]:5701 [dev] [4.1-SNAPSHOT]

Members {size:1, ver:1} [
    Member [192.168.1.10]:5701 - 7362c66f-ef9f-4a6a-a003-f8b33dfd292a this
]

Sep 03, 2020 2:21:58 PM com.hazelcast.core.LifecycleService
INFO: [192.168.1.10]:5701 [dev] [4.1-SNAPSHOT] [192.168.1.10]:5701 is STARTED
```

Adding User Library to CLASSPATH

When you want to use features such as querying and language interoperability, you might need to add your own Java classes to the Hazelcast member in order to use them from your Python client. This can be done by adding your own compiled code to the `CLASSPATH`. To do this, compile your code with the `CLASSPATH` and add the compiled files to the `user-lib` directory in the extracted `hazelcast-<version>.zip` (or `tar`). Then, you can start your Hazelcast member by using the start scripts in the `bin` directory. The start scripts will automatically add your compiled classes to the `CLASSPATH`.

Note that if you are adding an `IdentifiedDataSerializable` or a `Portable` class, you need to add its factory too. Then, you should configure the factory in the `hazelcast.xml` configuration file. This file resides in the `bin` directory where you extracted the `hazelcast-<version>.zip` (or `tar`).

The following is an example configuration when you are adding an `IdentifiedDataSerializable` class:

```
<hazelcast>
    ...
    <serialization>
       <data-serializable-factories>
```

(continues on next page)

(continued from previous page)

```
            <data-serializable-factory factory-id=<identified-factory-id>>
                IdentifiedFactoryClassName
            </data-serializable-factory>
        </data-serializable-factories>
    </serialization>
    ...
</hazelcast>
```

If you want to add a `Portable` class, you should use `<portable-factories>` instead of `<data-serializable-factories>` in the above configuration.

See the Hazelcast IMDG Reference Manual for more information on setting up the clusters.

2.3.3 Downloading and Installing

You can download and install the Python client from PyPI using pip. Run the following command:

```
pip install hazelcast-python-client
```

Alternatively, it can be installed from the source using the following command:

```
python setup.py install
```

2.3.4 Basic Configuration

If you are using Hazelcast IMDG and Python client on the same computer, generally the default configuration should be fine. This is great for trying out the client. However, if you run the client on a different computer than any of the cluster members, you may need to do some simple configurations such as specifying the member addresses.

The Hazelcast IMDG members and clients have their own configuration options. You may need to reflect some of the member side configurations on the client side to properly connect to the cluster.

This section describes the most common configuration elements to get you started in no time. It discusses some member side configuration options to ease the understanding of Hazelcast's ecosystem. Then, the client side configuration options regarding the cluster connection are discussed. The configurations for the Hazelcast IMDG data structures that can be used in the Python client are discussed in the following sections.

See the Hazelcast IMDG Reference Manual and *Configuration Overview* section for more information.

Configuring Hazelcast IMDG

Hazelcast IMDG aims to run out-of-the-box for most common scenarios. However if you have limitations on your network such as multicast being disabled, you may have to configure your Hazelcast IMDG members so that they can find each other on the network. Also, since most of the distributed data structures are configurable, you may want to configure them according to your needs. We will show you the basics about network configuration here.

You can use the following options to configure Hazelcast IMDG:

- Using the `hazelcast.xml` configuration file.
- Programmatically configuring the member before starting it from the Java code.

Since we use standalone servers, we will use the `hazelcast.xml` file to configure our cluster members.

When you download and unzip `hazelcast-<version>.zip` (or `tar`), you see the `hazelcast.xml` in the `bin` directory. When a Hazelcast member starts, it looks for the `hazelcast.xml` file to load the configuration from. A sample `hazelcast.xml` is shown below.

```
<hazelcast>
    <cluster-name>dev</cluster-name>
    <network>
        <port auto-increment="true" port-count="100">5701</port>
        <join>
            <multicast enabled="true">
                <multicast-group>224.2.2.3</multicast-group>
                <multicast-port>54327</multicast-port>
            </multicast>
            <tcp-ip enabled="false">
                <interface>127.0.0.1</interface>
                <member-list>
                    <member>127.0.0.1</member>
                </member-list>
            </tcp-ip>
        </join>
        <ssl enabled="false"/>
    </network>
    <partition-group enabled="false"/>
    <map name="default">
        <backup-count>1</backup-count>
    </map>
</hazelcast>
```

We will go over some important configuration elements in the rest of this section.

- `<cluster-name>`: Specifies which cluster this member belongs to. A member connects only to the other members that are in the same cluster as itself. You may give your clusters different names so that they can live in the same network without disturbing each other. Note that the cluster name should be the same across all members and clients that belong to the same cluster.
- `<network>`
  - `<port>`: Specifies the port number to be used by the member when it starts. Its default value is 5701. You can specify another port number, and if you set `auto-increment` to `true`, then Hazelcast will try the subsequent ports until it finds an available port or the `port-count` is reached.
  - `<join>`: Specifies the strategies to be used by the member to find other cluster members. Choose which strategy you want to use by setting its `enabled` attribute to `true` and the others to `false`.
    * `<multicast>`: Members find each other by sending multicast requests to the specified address and port. It is very useful if IP addresses of the members are not static.
    * `<tcp>`: This strategy uses a pre-configured list of known members to find an already existing cluster. It is enough for a member to find only one cluster member to connect to the cluster. The rest of the member list is automatically retrieved from that member. We recommend putting multiple known member addresses there to avoid disconnectivity should one of the members in the list is unavailable at the time of connection.

These configuration elements are enough for most connection scenarios. Now we will move onto the configuration of the Python client.

Configuring Hazelcast Python Client

To configure your Hazelcast Python client, you need to pass configuration options as keyword arguments to your client at the startup. The names of the configuration options is similar to `hazelcast.xml` configuration file used when configuring the member, but flatter. It is done this way to make it easier to transfer Hazelcast skills to multiple platforms.

This section describes some network configuration settings to cover common use cases in connecting the client to a cluster. See the *Configuration Overview* section and the following sections for information about detailed network configurations and/or additional features of Hazelcast Python client configuration.

```
import hazelcast

client = hazelcast.HazelcastClient(
    cluster_members=[
        "some-ip-address:port"
    ],
    cluster_name="name-of-your-cluster",
)
```

It's also possible to omit the keyword arguments in order to use the default settings.

```
import hazelcast

client = hazelcast.HazelcastClient()
```

If you run the Hazelcast IMDG members in a different server than the client, you most probably have configured the members' ports and cluster names as explained in the previous section. If you did, then you need to make certain changes to the network settings of your client.

Cluster Name Setting

You need to provide the name of the cluster, if it is defined on the server side, to which you want the client to connect.

```
import hazelcast

client = hazelcast.HazelcastClient(
    cluster_name="name-of-your-cluster",
)
```

Network Settings

You need to provide the IP address and port of at least one member in your cluster so the client can find it.

```
import hazelcast

client = hazelcast.HazelcastClient(
    cluster_members=["some-ip-address:port"]
)
```

2.3.5 Basic Usage

Now that we have a working cluster and we know how to configure both our cluster and client, we can run a simple program to use a distributed map in the Python client.

```
import logging
import hazelcast

# Enable logging to see the logs
logging.basicConfig(level=logging.INFO)

# Connect to Hazelcast cluster
client = hazelcast.HazelcastClient()

client.shutdown()
```

This should print logs about the cluster members such as address, port and UUID to the `stderr`.

```
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is STARTING
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is STARTED
INFO:hazelcast.connection:Trying to connect to Address(host=127.0.0.1, port=5701)
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is CONNECTED
INFO:hazelcast.connection:Authenticated with server Address(host=172.17.0.2,␣
↪port=5701):7682c357-3bec-4841-b330-6f9ae0c08253, server version: 4.0, local␣
↪address: Address(host=127.0.0.1, port=56718)
INFO:hazelcast.cluster:

Members [1] {
    Member [172.17.0.2]:5701 - 7682c357-3bec-4841-b330-6f9ae0c08253
}

INFO:hazelcast.client:Client started
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is SHUTTING_DOWN
INFO:hazelcast.connection:Removed connection to Address(host=127.0.0.1,␣
↪port=5701):7682c357-3bec-4841-b330-6f9ae0c08253, connection: Connection(id=0,␣
↪live=False, remote_address=Address(host=172.17.0.2, port=5701))
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is DISCONNECTED
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is SHUTDOWN
```

Congratulations. You just started a Hazelcast Python client.

Using a Map

Let's manipulate a distributed map (similar to Python's builtin `dict`) on a cluster using the client.

```
import hazelcast

client = hazelcast.HazelcastClient()

personnel_map = client.get_map("personnel-map")
personnel_map.put("Alice", "IT")
personnel_map.put("Bob", "IT")
personnel_map.put("Clark", "IT")

print("Added IT personnel. Printing all known personnel")

for person, department in personnel_map.entry_set().result():
    print("%s is in %s department" % (person, department))
```

(continues on next page)

(continued from previous page)

```
client.shutdown()
```

Output

```
Added IT personnel. Printing all known personnel
Alice is in IT department
Clark is in IT department
Bob is in IT department
```

You see this example puts all the IT personnel into a cluster-wide `personnel-map` and then prints all the known personnel.

Now, run the following code.

```
import hazelcast

client = hazelcast.HazelcastClient()

personnel_map = client.get_map("personnel-map")
personnel_map.put("Denise", "Sales")
personnel_map.put("Erwing", "Sales")
personnel_map.put("Faith", "Sales")

print("Added Sales personnel. Printing all known personnel")

for person, department in personnel_map.entry_set().result():
    print("%s is in %s department" % (person, department))

client.shutdown()
```

Output

```
Added Sales personnel. Printing all known personnel
Denise is in Sales department
Erwing is in Sales department
Faith is in Sales department
Alice is in IT department
Clark is in IT department
Bob is in IT department
```

Note: For the sake of brevity we are going to omit boilerplate parts, like `imports`, in the later code snippets. Refer to the *Code Samples* section to see samples with the complete code.

You will see this time we add only the sales employees but we get the list of all known employees including the ones in IT. That is because our map lives in the cluster and no matter which client we use, we can access the whole map.

You may wonder why we have used `result()` method over the `entry_set()` method of the `personnel_map`. That is because the Hazelcast Python client is designed to be fully asynchronous. Every method call over distributed objects such as `put()`, `get()`, `entry_set()`, etc. will return a `Future` object that is similar to the Future class of the concurrent.futures module.

With this design choice, method calls over the distributed objects can be executed asynchronously without blocking the execution order of your program.

You may get the value returned by the method calls using the `result()` method of the `Future` class. This will

block the execution of your program and will wait until the future finishes running. Then, it will return the value returned by the call which are key-value pairs in our `entry_set()` method call.

You may also attach a function to the future objects that will be called, with the future as its only argument, when the future finishes running.

For example, the part where we printed the personnel in above code can be rewritten with a callback attached to the `entry_set()`, as shown below..

```
def entry_set_cb(future):
    for person, department in future.result():
        print("%s is in %s department" % (person, department))


personnel_map.entry_set().add_done_callback(entry_set_cb)
time.sleep(1)  # wait for Future to complete
```

Asynchronous operations are far more efficient in single threaded Python interpreter but you may want all of your method calls over distributed objects to be blocking. For this purpose, Hazelcast Python client provides a helper method called `blocking()`. This method blocks the execution of your program for all the method calls over distributed objects until the return value of your call is calculated and returns that value directly instead of a `Future` object.

To make the `personnel_map` presented previously in this section blocking, you need to call `blocking()` method over it.

```
personnel_map = client.get_map("personnel-map").blocking()
```

Now, all the methods over the `personnel_map`, such as `put()` and `entry_set()`, will be blocking. So, you don't need to call `result()` over it or attach a callback to it anymore.

```
for person, department in personnel_map.entry_set():
    print("%s is in %s department" % (person, department))
```

2.3.6 Code Samples

See the Hazelcast Python examples for more code samples.

2.4 Features

Hazelcast Python client supports the following data structures and features:

- Map
- Queue
- Set
- List
- MultiMap
- Replicated Map
- Ringbuffer
- Topic

- CRDT PN Counter
- Flake Id Generator
- Distributed Executor Service
- Event Listeners
- Sub-Listener Interfaces for Map Listener
- Entry Processor
- Transactional Map
- Transactional MultiMap
- Transactional Queue
- Transactional List
- Transactional Set
- Query (Predicates)
- Entry Processor
- Built-in Predicates
- Listener with Predicate
- Near Cache Support
- Programmatic Configuration
- SSL Support (requires Enterprise server)
- Mutual Authentication (requires Enterprise server)
- Authorization
- Management Center Integration / Awareness
- Client Near Cache Stats
- Client Runtime Stats
- Client Operating Systems Stats
- Hazelcast Cloud Discovery
- Smart Client
- Unisocket Client
- Lifecycle Service
- Hazelcast Cloud Discovery
- IdentifiedDataSerializable Serialization
- Portable Serialization
- Custom Serialization
- JSON Serialization
- Global Serialization
- Connection Strategy
- Connection Retry

2.5 Configuration Overview

For configuration of the Hazelcast Python client, just pass the keyword arguments to the client to configure the desired aspects. An example is shown below.

```
client = hazelcast.HazelcastClient(
    cluster_members=["127.0.0.1:5701"]
)
```

See the API documentation of *hazelcast.client.HazelcastClient* for details.

2.6 Serialization

Serialization is the process of converting an object into a stream of bytes to store the object in the memory, a file or database, or transmit it through the network. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization. Hazelcast offers you its own native serialization methods. You will see these methods throughout this chapter.

Hazelcast serializes all your objects before sending them to the server. The `bool`, `int`, `long` (for Python 2), `float`, `str`, `unicode` (for Python 2) and `bytearray` types are serialized natively and you cannot override this behavior. The following table is the conversion of types for the Java server side.

Python	Java
bool	Boolean
int	Byte, Short, Integer, Long, BigInteger
long	Byte, Short, Integer, Long, BigInteger
float	Float, Double
str	String
unicode	String
bytearray	byte[]

Note: A `int` or `long` type is serialized as `Integer` by default. You can configure this behavior using the `default_int_type` argument.

Arrays of the above types can be serialized as `boolean[]`, `byte[]`, `short[]`, `int[]`, `float[]`, `double[]`, `long[]` and `string[]` for the Java server side, respectively.

Serialization Priority

When Hazelcast Python client serializes an object:

1. It first checks whether the object is `None`.
2. If the above check fails, then it checks if it is an instance of `IdentifiedDataSerializable`.
3. If the above check fails, then it checks if it is an instance of `Portable`.
4. If the above check fails, then it checks if it is an instance of one of the default types (see the default types above).
5. If the above check fails, then it looks for a user-specified *Custom Serialization*.
6. If the above check fails, it will use the registered *Global Serialization* if one exists.
7. If the above check fails, then the Python client uses `cPickle` (for Python 2) or `pickle` (for Python 3) by default.

However, `cPickle/pickle Serialization` is not the best way of serialization in terms of performance and interoperability between the clients in different languages. If you want the serialization to work faster or you use the clients in different languages, Hazelcast offers its own native serialization types, such as *IdentifiedDataSerializable Serialization* and *Portable Serialization*.

On top of all, if you want to use your own serialization type, you can use a *Custom Serialization*.

2.6.1 IdentifiedDataSerializable Serialization

For a faster serialization of objects, Hazelcast recommends to extend the `IdentifiedDataSerializable` class.

The following is an example of a class that extends `IdentifiedDataSerializable`:

```
from hazelcast.serialization.api import IdentifiedDataSerializable

class Address(IdentifiedDataSerializable):
    def __init__(self, street=None, zip_code=None, city=None, state=None):
        self.street = street
        self.zip_code = zip_code
        self.city = city
        self.state = state

    def get_class_id(self):
        return 1

    def get_factory_id(self):
        return 1

    def write_data(self, output):
        output.write_utf(self.street)
        output.write_int(self.zip_code)
        output.write_utf(self.city)
        output.write_utf(self.state)

    def read_data(self, input):
        self.street = input.read_utf()
        self.zip_code = input.read_int()
        self.city = input.read_utf()
        self.state = input.read_utf()
```

Note: Refer to `ObjectDataInput/ObjectDataOutput` classes in the `hazelcast.serialization.api` package to understand methods available on the `input/output` objects.

The IdentifiedDataSerializable uses `get_class_id()` and `get_factory_id()` methods to reconstitute the object. To complete the implementation, an `IdentifiedDataSerializable` factory should also be created and registered into the client using the `data_serializable_factories` argument. A factory is a dictionary that stores class ID and the `IdentifiedDataSerializable` class type pairs as the key and value. The factory's responsibility is to store the right `IdentifiedDataSerializable` class type for the given class ID.

A sample `IdentifiedDataSerializable` factory could be created as follows:

```
factory = {
    1: Address
}
```

Note that the keys of the dictionary should be the same as the class IDs of their corresponding `IdentifiedDataSerializable` class types.

Note: For IdentifiedDataSerializable to work in Python client, the class that inherits it should have default valued parameters in its `__init__` method so that an instance of that class can be created without passing any arguments to it.

The last step is to register the `IdentifiedDataSerializable` factory to the client.

```
client = hazelcast.HazelcastClient(
    data_serializable_factories={
        1: factory
    }
)
```

Note that the ID that is passed as the key of the factory is same as the factory ID that the `Address` class returns.

2.6.2 Portable Serialization

As an alternative to the existing serialization methods, Hazelcast offers portable serialization. To use it, you need to extend the `Portable` class. Portable serialization has the following advantages:

- Supporting multiversion of the same object type.
- Fetching individual fields without having to rely on the reflection.
- Querying and indexing support without deserialization and/or reflection.

In order to support these features, a serialized `Portable` object contains meta information like the version and concrete location of the each field in the binary data. This way Hazelcast is able to navigate in the binary data and deserialize only the required field without actually deserializing the whole object which improves the query performance.

With multiversion support, you can have two members each having different versions of the same object; Hazelcast stores both meta information and uses the correct one to serialize and deserialize portable objects depending on the member. This is very helpful when you are doing a rolling upgrade without shutting down the cluster.

Also note that portable serialization is totally language independent and is used as the binary protocol between Hazelcast server and clients.

A sample portable implementation of a `Foo` class looks like the following:

```
from hazelcast.serialization.api import Portable

class Foo(Portable):
    def __init__(self, foo=None):
        self.foo = foo

    def get_class_id(self):
        return 1

    def get_factory_id(self):
        return 1

    def write_portable(self, writer):
        writer.write_utf("foo", self.foo)
```

(continues on next page)

(continued from previous page)

```
    def read_portable(self, reader):
        self.foo = reader.read_utf("foo")
```

Note: Refer to `PortableReader`/`PortableWriter` classes in the `hazelcast.serialization.api` package to understand methods available on the `reader`/`writer` objects.

Note: For Portable to work in Python client, the class that inherits it should have default valued parameters in its `__init__` method so that an instance of that class can be created without passing any arguments to it.

Similar to `IdentifiedDataSerializable`, a `Portable` class must provide the `get_class_id()` and `get_factory_id()` methods. The factory dictionary will be used to create the `Portable` object given the class ID.

A sample `Portable` factory could be created as follows:

```
factory = {
    1: Foo
}
```

Note that the keys of the dictionary should be the same as the class IDs of their corresponding `Portable` class types.

The last step is to register the `Portable` factory to the client.

```
client = hazelcast.HazelcastClient(
    portable_factories={
        1: factory
    }
)
```

Note that the ID that is passed as the key of the factory is same as the factory ID that `Foo` class returns.

Versioning for Portable Serialization

More than one version of the same class may need to be serialized and deserialized. For example, a client may have an older version of a class and the member to which it is connected may have a newer version of the same class.

Portable serialization supports versioning. It is a global versioning, meaning that all portable classes that are serialized through a member get the globally configured portable version.

You can declare the version using the `portable_version` argument, as shown below.

```
client = hazelcast.HazelcastClient(
    portable_version=1
)
```

If you update the class by changing the type of one of the fields or by adding a new field, it is a good idea to upgrade the version of the class, rather than sticking to the global version specified in the configuration. In the Python client, you can achieve this by simply adding the `get_class_version()` method to your class's implementation of `Portable`, and returning class version different than the default global version.

Note: If you do not use the `get_class_version()` method in your `Portable` implementation, it will have the

global version, by default.

Here is an example implementation of creating a version 2 for the above Foo class:

```
from hazelcast.serialization.api import Portable

class Foo(Portable):
    def __init__(self, foo=None, foo2=None):
        self.foo = foo
        self.foo2 = foo2

    def get_class_id(self):
        return 1

    def get_factory_id(self):
        return 1

    def get_class_version(self):
        return 2

    def write_portable(self, writer):
        writer.write_utf("foo", self.foo)
        writer.write_utf("foo2", self.foo2)

    def read_portable(self, reader):
        self.foo = reader.read_utf("foo")
        self.foo2 = reader.read_utf("foo2")
```

You should consider the following when you perform versioning:

- It is important to change the version whenever an update is performed in the serialized fields of a class, for example by incrementing the version.
- If a client performs a Portable deserialization on a field and then that Portable is updated by removing that field on the cluster side, this may lead to problems such as an AttributeError being raised when an older version of the client tries to access the removed field.
- Portable serialization does not use reflection and hence, fields in the class and in the serialized content are not automatically mapped. Field renaming is a simpler process. Also, since the class ID is stored, renaming the Portable does not lead to problems.
- Types of fields need to be updated carefully. Hazelcast performs basic type upgradings, such as `int` to `float`.

Example Portable Versioning Scenarios:

Assume that a new client joins to the cluster with a class that has been modified and class's version has been upgraded due to this modification.

If you modified the class by adding a new field, the new client's put operations include that new field. If this new client tries to get an object that was put from the older clients, it gets null for the newly added field.

If you modified the class by removing a field, the old clients get null for the objects that are put by the new client.

If you modified the class by changing the type of a field to an incompatible type (such as from `int` to `str`), a `TypeError` (wrapped as `HazelcastSerializationError`) is generated as the client tries accessing an object with the older version of the class. The same applies if a client with the old version tries to access a new version object.

If you did not modify a class at all, it works as usual.

2.6.3 Custom Serialization

Hazelcast lets you plug a custom serializer to be used for serialization of objects.

Let's say you have a class called `Musician` and you would like to customize the serialization for it, since you may want to use an external serializer for only one class.

```
class Musician:
    def __init__(self, name):
        self.name = name
```

Let's say your custom `MusicianSerializer` will serialize `Musician`. This time, your custom serializer must extend the `StreamSerializer` class.

```
from hazelcast.serialization.api import StreamSerializer

class MusicianSerializer(StreamSerializer):
    def get_type_id(self):
        return 10

    def destroy(self):
        pass

    def write(self, output, obj):
        output.write_utf(obj.name)

    def read(self, input):
        name = input.read_utf()
        return Musician(name)
```

Note that the serializer `id` must be unique as Hazelcast will use it to lookup the `MusicianSerializer` while it deserializes the object. Now the last required step is to register the `MusicianSerializer` to the client.

```
client = hazelcast.HazelcastClient(
    custom_serializers={
        Musician: MusicianSerializer
    }
)
```

From now on, Hazelcast will use `MusicianSerializer` to serialize `Musician` objects.

2.6.4 JSON Serialization

You can use the JSON formatted strings as objects in Hazelcast cluster. Creating JSON objects in the cluster does not require any server side coding and hence you can just send a JSON formatted string object to the cluster and query these objects by fields.

In order to use JSON serialization, you should use the `HazelcastJsonValue` object for the key or value.

`HazelcastJsonValue` is a simple wrapper and identifier for the JSON formatted strings. You can get the JSON string from the `HazelcastJsonValue` object using the `to_string()` method.

You can construct `HazelcastJsonValue` from strings or JSON serializable Python objects. If a Python object is provided to the constructor, `HazelcastJsonValue` tries to convert it to a JSON string. If an error occurs during the conversion, it is raised directly. If a string argument is provided to the constructor, it is used as it is.

No JSON parsing is performed but it is your responsibility to provide correctly formatted JSON strings. The client will not validate the string, and it will send it to the cluster as it is. If you submit incorrectly formatted JSON strings

and, later, if you query those objects, it is highly possible that you will get formatting errors since the server will fail to deserialize or find the query fields.

Here is an example of how you can construct a `HazelcastJsonValue` and put to the map:

```
# From JSON string
json_map.put("item1", HazelcastJsonValue("{\"age\": 4}"))

# # From JSON serializable object
json_map.put("item2", HazelcastJsonValue({"age": 20}))
```

You can query JSON objects in the cluster using the `Predicates` of your choice. An example JSON query for querying the values whose age is less than 6 is shown below:

```
# Get the objects whose age is less than 6
result = json_map.values(less_or_equal("age", 6))
print("Retrieved %s values whose age is less than 6." % len(result))
print("Entry is", result[0].to_string())
```

2.6.5 Global Serialization

The global serializer is identical to custom serializers from the implementation perspective. The global serializer is registered as a fallback serializer to handle all other objects if a serializer cannot be located for them.

By default, `cPickle/pickle` serialization is used if the class is not `IdentifiedDataSerializable` or `Portable` or there is no custom serializer for it. When you configure a global serializer, it is used instead of `cPickle/pickle` serialization.

Use Cases:

- Third party serialization frameworks can be integrated using the global serializer.
- For your custom objects, you can implement a single serializer to handle all of them.

A sample global serializer that integrates with a third party serializer is shown below.

```
import some_third_party_serializer
from hazelcast.serialization.api import StreamSerializer

class GlobalSerializer(StreamSerializer):
    def get_type_id(self):
        return 20

    def destroy(self):
        pass

    def write(self, output, obj):
        output.write_utf(some_third_party_serializer.serialize(obj))

    def read(self, input):
        return some_third_party_serializer.deserialize(input.read_utf())
```

You should register the global serializer to the client.

```
client = hazelcast.HazelcastClient(
    global_serializer=GlobalSerializer
)
```

2.7 Setting Up Client Network

Main parts of network related configuration for Hazelcast Python client may be tuned via the arguments described in this section.

Here is an example of configuring the network for Python client.

```
client = hazelcast.HazelcastClient(
    cluster_members=[
        "10.1.1.21",
        "10.1.1.22:5703"
    ],
    smart_routing=True,
    redo_operation=False,
    connection_timeout=6.0
)
```

2.7.1 Providing Member Addresses

Address list is the initial list of cluster addresses which the client will connect to. The client uses this list to find an alive member. Although it may be enough to give only one address of a member in the cluster (since all members communicate with each other), it is recommended that you give the addresses for all the members.

```
client = hazelcast.HazelcastClient(
    cluster_members=[
        "10.1.1.21",
        "10.1.1.22:5703"
    ]
)
```

If the port part is omitted, then `5701`, `5702` and `5703` will be tried in a random order.

You can specify multiple addresses with or without the port information as seen above. The provided list is shuffled and tried in a random order. Its default value is `localhost`.

2.7.2 Setting Smart Routing

Smart routing defines whether the client mode is smart or unisocket. See the *Python Client Operation Modes* section for the description of smart and unisocket modes.

```
client = hazelcast.HazelcastClient(
    smart_routing=True,
)
```

Its default value is `True` (smart client mode).

2.7.3 Enabling Redo Operation

It enables/disables redo-able operations. While sending the requests to the related members, the operations can fail due to various reasons. Read-only operations are retried by default. If you want to enable retry for the other operations, you can set the `redo_operation` to `True`.

```
client = hazelcast.HazelcastClient(
    redo_operation=False
)
```

Its default value is `False` (disabled).

2.7.4 Setting Connection Timeout

Connection timeout is the timeout value in seconds for the members to accept the client connection requests.

```
client = hazelcast.HazelcastClient(
    connection_timeout=6.0
)
```

Its default value is `5.0` seconds.

2.7.5 Enabling Client TLS/SSL

You can use TLS/SSL to secure the connection between the clients and members. If you want to enable TLS/SSL for the client-cluster connection, you should set the SSL configuration. Please see the *TLS/SSL* section.

As explained in the *TLS/SSL* section, Hazelcast members have key stores used to identify themselves (to other members) and Hazelcast Python clients have certificate authorities used to define which members they can trust. Hazelcast has the mutual authentication feature which allows the Python clients also to have their private keys and public certificates, and members to have their certificate authorities so that the members can know which clients they can trust. See the *Mutual Authentication* section.

2.7.6 Enabling Hazelcast Cloud Discovery

Hazelcast Python client can discover and connect to Hazelcast clusters running on Hazelcast Cloud. For this, provide authentication information as `cluster_name` and enable cloud discovery by setting your `cloud_discovery_token` as shown below.

```
client = hazelcast.HazelcastClient(
    cluster_name="name-of-your-cluster",
    cloud_discovery_token="discovery-token"
)
```

If you have enabled encryption for your cluster, you should also enable TLS/SSL configuration for the client to secure communication between your client and cluster members as described in the *TLS/SSL for Hazelcast Python Clients* section.

2.7.7 Configuring Backup Acknowledgment

When an operation with sync backup is sent by a client to the Hazelcast member(s), the acknowledgment of the operation's backup is sent to the client by the backup replica member(s). This improves the performance of the client operations.

To disable backup acknowledgement, you should use the `backup_ack_to_client_enabled` configuration option.

```
client = hazelcast.HazelcastClient(
    backup_ack_to_client_enabled=False,
)
```

Its default value is `True`. This option has no effect for unisocket clients.

You can also fine-tune this feature using the config options as described below:

- `operation_backup_timeout`: Default value is `5` seconds. If an operation has backups, this property specifies how long the invocation waits for acks from the backup replicas. If acks are not received from some of the backups, there will not be any rollback on the other successful replicas.
- `fail_on_indeterminate_operation_state`: Default value is `False`. When it is `True`, if an operation has sync backups and acks are not received from backup replicas in time, or the member which owns primary replica of the target partition leaves the cluster, then the invocation fails. However, even if the invocation fails, there will not be any rollback on other successful replicas.

2.8 Client Connection Strategy

Hazelcast Python client can be configured to connect to a cluster in an async manner during the client start and reconnecting after a cluster disconnect. Both of these options are configured via arguments below.

You can configure the client's starting mode as async or sync using the configuration element `async_start`. When it is set to `True` (async), the behavior of `hazelcast.HazelcastClient()` call changes. It returns a client instance without waiting to establish a cluster connection. In this case, the client rejects any network dependent operation with `ClientOfflineError` immediately until it connects to the cluster. If it is `False`, the call is not returned and the client is not created until a connection with the cluster is established. Its default value is `False` (sync).

You can also configure how the client reconnects to the cluster after a disconnection. This is configured using the configuration element `reconnect_mode`; it has three options:

- `OFF`: Client rejects to reconnect to the cluster and triggers the shutdown process.
- `ON`: Client opens a connection to the cluster in a blocking manner by not resolving any of the waiting invocations.
- `ASYNC`: Client opens a connection to the cluster in a non-blocking manner by resolving all the waiting invocations with `ClientOfflineError`.

Its default value is `ON`.

The example configuration below show how to configure a Python client's starting and reconnecting modes.

```
from hazelcast.config import ReconnectMode

client = hazelcast.HazelcastClient(
    async_start=False,
    # You can also set this to "ON"
    # without importing anything.
```

(continues on next page)

(continued from previous page)

```
    reconnect_mode=ReconnectMode.ON
)
```

2.8.1 Configuring Client Connection Retry

When the client is disconnected from the cluster, it searches for new connections to reconnect. You can configure the frequency of the reconnection attempts and client shutdown behavior using the argumentes below.

```
client = hazelcast.HazelcastClient(
    retry_initial_backoff=1,
    retry_max_backoff=15,
    retry_multiplier=1.5,
    retry_jitter=0.2,
    cluster_connect_timeout=120
)
```

The following are configuration element descriptions:

- `retry_initial_backoff`: Specifies how long to wait (backoff), in seconds, after the first failure before retrying. Its default value is `1`. It must be non-negative.
- `retry_max_backoff`: Specifies the upper limit for the backoff in seconds. Its default value is `30`. It must be non-negative.
- `retry_multiplier`: Factor to multiply the backoff after a failed retry. Its default value is `1`. It must be greater than or equal to `1`.
- `retry_jitter`: Specifies by how much to randomize backoffs. Its default value is `0`. It must be in range `0` to `1`.
- `cluster_connect_timeout`: Timeout value in seconds for the client to give up to connect to the current cluster. Its default value is `120`.

A pseudo-code is as follows:

```
begin_time = get_current_time()
current_backoff = INITIAL_BACKOFF
while (try_connect(connection_timeout)) != SUCCESS) {
    if (get_current_time() - begin_time >= CLUSTER_CONNECT_TIMEOUT) {
        // Give up to connecting to the current cluster and switch to another if␣
↪exists.
    }
    sleep(current_backoff + uniform_random(-JITTER * current_backoff, JITTER *␣
↪current_backoff))
    current_backoff = min(current_backoff * MULTIPLIER, MAX_BACKOFF)
}
```

Note that, `try_connect` above tries to connect to any member that the client knows, and for each connection we have a connection timeout; see the *Setting Connection Timeout* section.

2.9 Using Python Client with Hazelcast IMDG

This chapter provides information on how you can use Hazelcast IMDG's data structures in the Python client, after giving some basic information including an overview to the client API, operation modes of the client and how it handles the failures.

2.9.1 Python Client API Overview

Hazelcast Python client is designed to be fully asynchronous. See the *Basic Usage* section to learn more about the asynchronous nature of the Python Client.

If you are ready to go, let's start to use Hazelcast Python client.

The first step is configuration. See the *Configuration Overview* section for details.

The following is an example on how to configure and initialize the `HazelcastClient` to connect to the cluster:

```
client = hazelcast.HazelcastClient(
    cluster_name="dev",
    cluster_members=[
        "198.51.100.2"
    ]
)
```

This client object is your gateway to access all the Hazelcast distributed objects.

Let's create a map and populate it with some data, as shown below.

```
# Get a Map called 'my-distributed-map'
customer_map = client.get_map("customers").blocking()

# Write and read some data
customer_map.put("1", "John Stiles")
customer_map.put("2", "Richard Miles")
customer_map.put("3", "Judy Doe")
```

As the final step, if you are done with your client, you can shut it down as shown below. This will release all the used resources and close connections to the cluster.

```
client.shutdown()
```

2.9.2 Python Client Operation Modes

The client has two operation modes because of the distributed nature of the data and cluster: smart and unisocket. Refer to the *Setting Smart Routing* section to see how to configure the client for different operation modes.

Smart Client

In the smart mode, the clients connect to each cluster member. Since each data partition uses the well known and consistent hashing algorithm, each client can send an operation to the relevant cluster member, which increases the overall throughput and efficiency. Smart mode is the default mode.

Unisocket Client

For some cases, the clients can be required to connect to a single member instead of each member in the cluster. Firewalls, security or some custom networking issues can be the reason for these cases.

In the unisocket client mode, the client will only connect to one of the configured addresses. This single member will behave as a gateway to the other members. For any operation requested from the client, it will redirect the request to the relevant member and return the response back to the client returned from this member.

2.9.3 Handling Failures

There are two main failure cases you should be aware of. Below sections explain these and the configurations you can perform to achieve proper behavior.

Handling Client Connection Failure

While the client is trying to connect initially to one of the members in the `cluster_members`, all the members might not be available. Instead of giving up, throwing an error and stopping the client, the client retries to connect as configured. This behavior is described in the *Configuring Client Connection Retry* section.

The client executes each operation through the already established connection to the cluster. If this connection(s) disconnects or drops, the client will try to reconnect as configured.

Handling Retry-able Operation Failure

While sending the requests to the related members, the operations can fail due to various reasons. Read-only operations are retried by default. If you want to enable retrying for the other operations, you can set the `redo_operation` to `True`. See the *Enabling Redo Operation* section.

You can set a timeout for retrying the operations sent to a member. This can be tuned by passing the `invocation_timeout` argument to the client. The client will retry an operation within this given period, of course, if it is a read-only operation or you enabled the `redo_operation` as stated in the above. This timeout value is important when there is a failure resulted by either of the following causes:

- Member throws an exception.
- Connection between the client and member is closed.
- Client's heartbeat requests are timed out.

When a connection problem occurs, an operation is retried if it is certain that it has not run on the member yet or if it is idempotent such as a read-only operation, i.e., retrying does not have a side effect. If it is not certain whether the operation has run on the member, then the non-idempotent operations are not retried. However, as explained in the first paragraph of this section, you can force all the client operations to be retried (`redo_operation`) when there is a connection failure between the client and member. But in this case, you should know that some operations may run multiple times causing conflicts. For example, assume that your client sent a `queue.offer` operation to the member and then the connection is lost. Since there will be no response for this operation, you will not know whether

it has run on the member or not. I f you enabled `redo_operation`, it means this operation may run again, which may cause two instances of the same object in the queue.

When invocation is being retried, the client may wait some time before it retries again. This duration can be configured using the `invocation_retry_pause` argument.

The default retry pause time is `1` second.

2.9.4 Using Distributed Data Structures

Most of the distributed data structures are supported by the Python client. In this chapter, you will learn how to use these distributed data structures.

Using Map

Hazelcast Map is a distributed dictionary. Through the Python client, you can perform operations like reading and writing from/to a Hazelcast Map with the well known get and put methods. For details, see the Map section in the Hazelcast IMDG Reference Manual.

A Map usage example is shown below.

```
# Get a Map called 'my-distributed-map'
my_map = client.get_map("my-distributed-map").blocking()

# Run Put and Get operations
my_map.put("key", "value")
my_map.get("key")

# Run concurrent Map operations (optimistic updates)
my_map.put_if_absent("somekey", "somevalue")
my_map.replace_if_same("key", "value", "newvalue")
```

Using MultiMap

Hazelcast MultiMap is a distributed and specialized map where you can store multiple values under a single key. For details, see the MultiMap section in the Hazelcast IMDG Reference Manual.

A MultiMap usage example is shown below.

```
# Get a MultiMap called 'my-distributed-multimap'
multi_map = client.get_multi_map("my-distributed-multimap").blocking()

# Put values in the map against the same key
multi_map.put("my-key", "value1")
multi_map.put("my-key", "value2")
multi_map.put("my-key", "value3")

# Read and print out all the values for associated with key called 'my-key'
# Outputs '['value2', 'value1', 'value3']'
values = multi_map.get("my-key")
print(values)

# Remove specific key/value pair
multi_map.remove("my-key", "value2")
```

Using Replicated Map

Hazelcast Replicated Map is a distributed key-value data structure where the data is replicated to all members in the cluster. It provides full replication of entries to all members for high speed access. For details, see the Replicated Map section in the Hazelcast IMDG Reference Manual.

A Replicated Map usage example is shown below.

```
# Get a ReplicatedMap called 'my-replicated-map'
replicated_map = client.get_replicated_map("my-replicated-map").blocking()

# Put and get a value from the Replicated Map
# (key/value is replicated to all members)
replaced_value = replicated_map.put("key", "value")

# Will be None as its first update
print("replaced value = {}".format(replaced_value)) # Outputs 'replaced value = None'

# The value is retrieved from a random member in the cluster
value = replicated_map.get("key")

print("value for key = {}".format(value)) # Outputs 'value for key = value'
```

Using Queue

Hazelcast Queue is a distributed queue which enables all cluster members to interact with it. For details, see the Queue section in the Hazelcast IMDG Reference Manual.

A Queue usage example is shown below.

```
# Get a Queue called 'my-distributed-queue'
queue = client.get_queue("my-distributed-queue").blocking()

# Offer a string into the Queue
queue.offer("item")

# Poll the Queue and return the string
item = queue.poll()

# Timed-restricted operations
queue.offer("another-item", 0.5)  # waits up to 0.5 seconds
another_item = queue.poll(5)  # waits up to 5 seconds

# Indefinitely blocking Operations
queue.put("yet-another-item")

print(queue.take()) # Outputs 'yet-another-item'
```

Using Set

Hazelcast Set is a distributed set which does not allow duplicate elements. For details, see the Set section in the Hazelcast IMDG Reference Manual.

A Set usage example is shown below.

```
# Get a Set called 'my-distributed-set'
my_set = client.get_set("my-distributed-set").blocking()

# Add items to the Set with duplicates
my_set.add("item1")
my_set.add("item1")
my_set.add("item2")
my_set.add("item2")
my_set.add("item2")
my_set.add("item3")

# Get the items. Note that there are no duplicates.
for item in my_set.get_all():
    print(item)
```

Using List

Hazelcast List is a distributed list which allows duplicate elements and preserves the order of elements. For details, see the List section in the Hazelcast IMDG Reference Manual.

A List usage example is shown below.

```
# Get a List called 'my-distributed-list'
my_list = client.get_list("my-distributed-list").blocking()

# Add element to the list
my_list.add("item1")
my_list.add("item2")

# Remove the first element
print("Removed:", my_list.remove_at(0))  # Outputs 'Removed: item1'

# There is only one element left
print("Current size is", my_list.size())  # Outputs 'Current size is 1'

# Clear the list
my_list.clear()
```

Using Ringbuffer

Hazelcast Ringbuffer is a replicated but not partitioned data structure that stores its data in a ring-like structure. You can think of it as a circular array with a given capacity. Each Ringbuffer has a tail and a head. The tail is where the items are added and the head is where the items are overwritten or expired. You can reach each element in a Ringbuffer using a sequence ID, which is mapped to the elements between the head and tail (inclusive) of the Ringbuffer. For details, see the Ringbuffer section in the Hazelcast IMDG Reference Manual.

A Ringbuffer usage example is shown below.

```
# Get a RingBuffer called "my-ringbuffer"
ringbuffer = client.get_ringbuffer("my-ringbuffer").blocking()

# Add two items into ring buffer
ringbuffer.add(100)
ringbuffer.add(200)

# We start from the oldest item.
# If you want to start from the next item, call ringbuffer.tail_sequence()+1
sequence = ringbuffer.head_sequence()
print(ringbuffer.read_one(sequence))  # Outputs '100'

sequence += 1
print(ringbuffer.read_one(sequence))  # Outputs '200'
```

Using Topic

Hazelcast Topic is a distribution mechanism for publishing messages that are delivered to multiple subscribers. For details, see the Topic section in the Hazelcast IMDG Reference Manual.

A Topic usage example is shown below.

```
# Function to be called when a message is published
def print_on_message(topic_message):
    print("Got message:", topic_message.message)

# Get a Topic called "my-distributed-topic"
topic = client.get_topic("my-distributed-topic").blocking()

# Add a Listener to the Topic
topic.add_listener(print_on_message)

# Publish a message to the Topic
topic.publish("Hello to distributed world") # Outputs 'Got message: Hello to␣
↪distributed world'
```

Using Transactions

Hazelcast Python client provides transactional operations like beginning transactions, committing transactions and retrieving transactional data structures like the `TransactionalMap`, `TransactionalSet`, `TransactionalList`, `TransactionalQueue` and `TransactionalMultiMap`.

You can create a `Transaction` object using the Python client to begin, commit and rollback a transaction. You can obtain transaction-aware instances of queues, maps, sets, lists and multimaps via the `Transaction` object, work with them and commit or rollback in one shot. For details, see the Transactions section in the Hazelcast IMDG Reference Manual.

```
# Create a Transaction object and begin the transaction
transaction = client.new_transaction(timeout=10)
transaction.begin()

# Get transactional distributed data structures
txn_map = transaction.get_map("transactional-map")
txn_queue = transaction.get_queue("transactional-queue")
```

(continues on next page)

(continued from previous page)

```
txt_set = transaction.get_set("transactional-set")
try:
    obj = txn_queue.poll()

    # Process obj

    txn_map.put("1", "value1")
    txt_set.add("value")

    # Do other things

    # Commit the above changes done in the cluster.
    transaction.commit()
except Exception as ex:
    # In the case of a transactional failure, rollback the transaction
    transaction.rollback()
    print("Transaction failed! {}".format(ex.args))
```

In a transaction, operations will not be executed immediately. Their changes will be local to the `Transaction` object until committed. However, they will ensure the changes via locks.

For the above example, when `txn_map.put()` is executed, no data will be put in the map but the key will be locked against changes. While committing, operations will be executed, the value will be put to the map and the key will be unlocked.

The isolation level in Hazelcast Transactions is `READ_COMMITTED` on the level of a single partition. If you are in a transaction, you can read the data in your transaction and the data that is already committed. If you are not in a transaction, you can only read the committed data.

Using PN Counter

Hazelcast `PNCounter` (Positive-Negative Counter) is a CRDT positive-negative counter implementation. It is an eventually consistent counter given there is no member failure. For details, see the PN Counter section in the Hazelcast IMDG Reference Manual.

A PN Counter usage example is shown below.

```
# Get a PN Counter called 'pn-counter'
pn_counter = client.get_pn_counter("pn-counter").blocking()

# Counter is initialized with 0
print(pn_counter.get()) # 0

# xx_and_get() variants does the operation
# and returns the final value
print(pn_counter.add_and_get(5))  # 5
print(pn_counter.decrement_and_get())  # 4

# get_and_xx() variants returns the current
# value and then does the operation
print(pn_counter.get_and_increment())  # 4
print(pn_counter.get())  # 5
```

Using Flake ID Generator

Hazelcast `FlakeIdGenerator` is used to generate cluster-wide unique identifiers. Generated identifiers are long primitive values and are k-ordered (roughly ordered). IDs are in the range from `0` to `2^63-1` (maximum signed long value). For details, see the FlakeIdGenerator section in the Hazelcast IMDG Reference Manual.

```
# Get a Flake ID Generator called 'flake-id-generator'
generator = client.get_flake_id_generator("flake-id-generator").blocking()

# Generate a some unique identifier
print("ID:", generator.new_id())
```

Configuring Flake ID Generator

You may configure Flake ID Generators using the `flake_id_generators` argument:

```
client = hazelcast.HazelcastClient(
    flake_id_generators={
        "flake-id-generator": {
            "prefetch_count": 123,
            "prefetch_validity": 150
        }
    }
)
```

The following are the descriptions of configuration elements and attributes:

- keys of the dictionary: Name of the Flake ID Generator.
- `prefetch_count`: Count of IDs which are pre-fetched on the background when one call to `generator.newId()` is made. Its value must be in the range `1` - `100,000`. Its default value is `100`.
- `prefetch_validity`: Specifies for how long the pre-fetched IDs can be used. After this time elapses, a new batch of IDs are fetched. Time unit is seconds. Its default value is `600` seconds (`10` minutes). The IDs contain a timestamp component, which ensures a rough global ordering of them. If an ID is assigned to an object that was created later, it will be out of order. If ordering is not important, set this value to `0`.

CP Subsystem

Hazelcast IMDG 4.0 introduces CP concurrency primitives with respect to the CAP principle, i.e., they always maintain linearizability and prefer consistency to availability during network partitions and client or server failures.

All data structures within CP Subsystem are available through `client.cp_subsystem` component of the client.

Before using Atomic Long, Lock, and Semaphore, CP Subsystem has to be enabled on cluster-side. Refer to CP Subsystem documentation for more information.

Data structures in CP Subsystem run in CP groups. Each CP group elects its own Raft leader and runs the Raft consensus algorithm independently. The CP data structures differ from the other Hazelcast data structures in two aspects. First, an internal commit is performed on the METADATA CP group every time you fetch a proxy from this interface. Hence, callers should cache returned proxy objects. Second, if you call `distributed_object.destroy()` on a CP data structure proxy, that data structure is terminated on the underlying CP group and cannot be reinitialized until the CP group is force-destroyed. For this reason, please make sure that you are completely done with a CP data structure before destroying its proxy.

Using AtomicLong

Hazelcast `AtomicLong` is the distributed implementation of atomic 64-bit integer counter. It offers various atomic operations such as `get`, `set`, `get_and_set`, `compare_and_set` and `increment_and_get`. This data structure is a part of CP Subsystem.

An Atomic Long usage example is shown below.

```
# Get an AtomicLong called "my-atomic-long"
atomic_long = client.cp_subsystem.get_atomic_long("my-atomic-long").blocking()
# Get current value
value = atomic_long.get()
print("Value:", value)
# Prints:
# Value: 0

# Increment by 42
atomic_long.add_and_get(42)
# Set to 0 atomically if the current value is 42
result = atomic_long.compare_and_set(42, 0)
print ('CAS operation result:', result)
# Prints:
# CAS operation result: True
```

AtomicLong implementation does not offer exactly-once / effectively-once execution semantics. It goes with at-least-once execution semantics by default and can cause an API call to be committed multiple times in case of CP member failures. It can be tuned to offer at-most-once execution semantics. Please see fail-on-indeterminate-operation-state server-side setting.

Using Lock

Hazelcast `FencedLock` is the distributed and reentrant implementation of a linearizable lock. It is CP with respect to the CAP principle. It works on top of the Raft consensus algorithm. It offers linearizability during crash-stop failures and network partitions. If a network partition occurs, it remains available on at most one side of the partition.

A basic Lock usage example is shown below.

```
# Get a FencedLock called "my-lock"
lock = client.cp_subsystem.get_lock("my-lock").blocking()
# Acquire the lock and get the fencing token
fence = lock.lock()
try:
    # Your guarded code goes here
    pass
finally:
    # Make sure to release the lock
    lock.unlock()
```

FencedLock works on top of CP sessions. It keeps a CP session open while the lock is acquired. Please refer to CP Session documentation for more information.

By default, FencedLock is reentrant. Once a caller acquires the lock, it can acquire the lock reentrantly as many times as it wants in a linearizable manner. You can configure the reentrancy behavior on the member side. For instance, reentrancy can be disabled and FencedLock can work as a non-reentrant mutex. You can also set a custom reentrancy limit. When the reentrancy limit is already reached, FencedLock does not block a lock call. Instead, it fails with `LockAcquireLimitReachedError` or a specified return value.

Distributed locks are unfortunately *not equivalent* to single-node mutexes because of the complexities in distributed systems, such as uncertain communication patterns, and independent and partial failures. In an asynchronous network, no lock service can guarantee mutual exclusion, because there is no way to distinguish between a slow and a crashed process. Consider the following scenario, where a Hazelcast client acquires a FencedLock, then hits a long pause. Since it will not be able to commit session heartbeats while paused, its CP session will be eventually closed. After this moment, another Hazelcast client can acquire this lock. If the first client wakes up again, it may not immediately notice that it has lost ownership of the lock. In this case, multiple clients think they hold the lock. If they attempt to perform an operation on a shared resource, they can break the system. To prevent such situations, you can choose to use an infinite session timeout, but this time probably you are going to deal with liveliness issues. For the scenario above, even if the first client actually crashes, requests sent by 2 clients can be re-ordered in the network and hit the external resource in reverse order.

There is a simple solution for this problem. Lock holders are ordered by a monotonic fencing token, which increments each time the lock is assigned to a new owner. This fencing token can be passed to external services or resources to ensure sequential execution of side effects performed by lock holders.

The following diagram illustrates the idea. Client-1 acquires the lock first and receives `1` as its fencing token. Then, it passes this token to the external service, which is our shared resource in this scenario. Just after that, Client-1 hits a long GC pause and eventually loses ownership of the lock because it misses to commit CP session heartbeats. Then, Client-2 chimes in and acquires the lock. Similar to Client-1, Client-2 passes its fencing token to the external service. After that, once Client-1 comes back alive, its write request will be rejected by the external service, and only Client-2 will be able to safely talk to it.

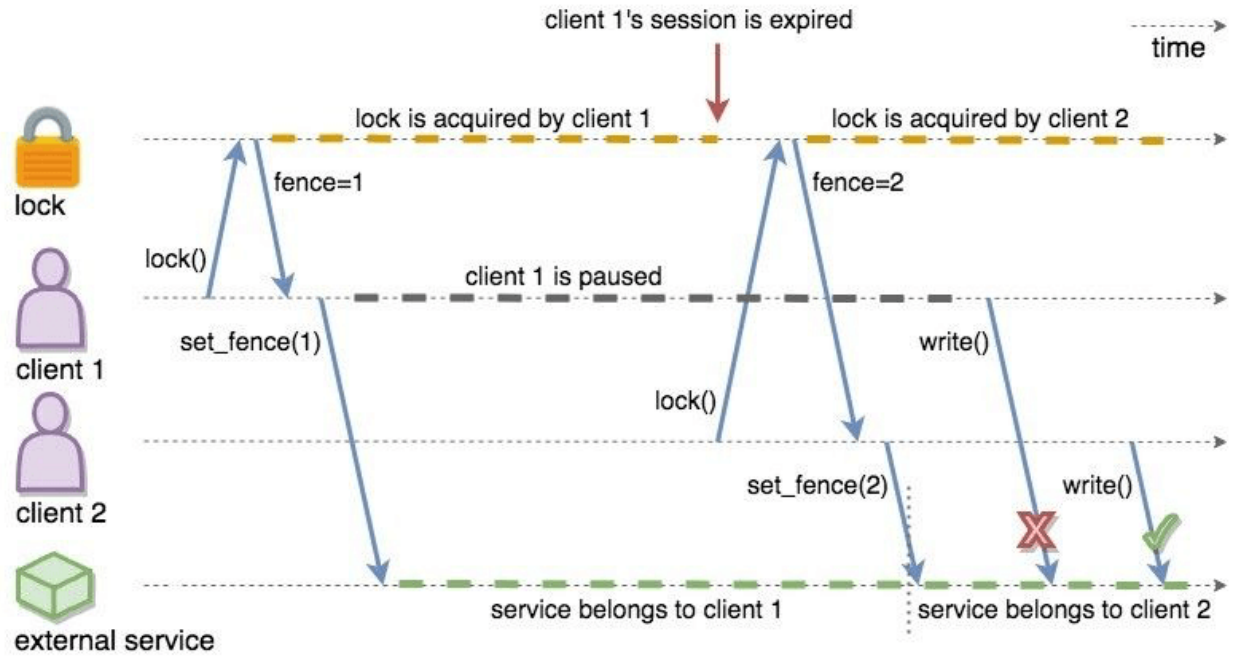

Fig. 1: CP Fenced Lock diagram

You can read more about the fencing token idea in Martin Kleppmann's "How to do distributed locking" blog post and Google's Chubby paper.

Using Semaphore

Hazelcast `Semaphore` is the distributed implementation of a linearizable and distributed semaphore. It offers multiple operations for acquiring the permits. This data structure is a part of CP Subsystem.

Semaphore is a cluster-wide counting semaphore. Conceptually, it maintains a set of permits. Each `acquire()` waits if necessary until a permit is available, and then takes it. Dually, each `release()` adds a permit, potentially releasing a waiting acquirer. However, no actual permit objects are used; the semaphore just keeps a count of the number available and acts accordingly.

A basic Semaphore usage example is shown below.

```
# Get a Semaphore called "my-semaphore"
semaphore = client.cp_subsystem.get_semaphore("my-semaphore").blocking()
# Try to initialize the semaphore
# (does nothing if the semaphore is already initialized)
semaphore.init(3)
# Acquire 3 permits out of 3
semaphore.acquire(3)
# Release 2 permits
semaphore.release(2)
# Check available permits
available = semaphore.available_permits()
print("Available:", available)
# Prints:
# Available: 2
```

Beware of the increased risk of indefinite postponement when using the multiple-permit acquire. If permits are released one by one, a caller waiting for one permit will acquire it before a caller waiting for multiple permits regardless of the call order. Correct usage of a semaphore is established by programming convention in the application.

As an alternative, potentially safer approach to the multiple-permit acquire, you can use the `try_acquire()` method of Semaphore. It tries to acquire the permits in optimistic manner and immediately returns with a `bool` operation result. It also accepts an optional `timeout` argument which specifies the timeout in seconds to acquire the permits before giving up.

```
# Try to acquire 2 permits
success = semaphore.try_acquire(2)
# Check for the result of the acquire request
if success:
    try:
        pass
        # Your guarded code goes here
    finally:
        # Make sure to release the permits
        semaphore.release(2)
```

Semaphore data structure has two variations:

- The default implementation is session-aware. In this one, when a caller makes its very first `acquire()` call, it starts a new CP session with the underlying CP group. Then, liveliness of the caller is tracked via this CP session. When the caller fails, permits acquired by this caller are automatically and safely released. However, the session-aware version comes with a limitation, that is, a Hazelcast client cannot release permits before acquiring them first. In other words, a client can release only the permits it has acquired earlier.
- The second implementation is sessionless. This one does not perform auto-cleanup of acquired permits on failures. Acquired permits are not bound to callers and permits can be released without acquiring first. However, you need to handle failed permit owners on your own. If a Hazelcast server or a client fails while holding some permits, they will not be automatically released. You can use the sessionless CP Semaphore implementation

by enabling JDK compatibility `jdk-compatible` server-side setting. Refer to Semaphore configuration documentation for more details.

Using CountDownLatch

Hazelcast `CountDownLatch` is the distributed implementation of a linearizable and distributed countdown latch. This data structure is a cluster-wide synchronization aid that allows one or more callers to wait until a set of operations being performed in other callers completes. This data structure is a part of CP Subsystem.

A basic CountDownLatch usage example is shown below.

```
# Get a CountDownLatch called "my-latch"
latch = client.cp_subsystem.get_count_down_latch("my-latch").blocking()
# Try to initialize the latch
# (does nothing if the count is not zero)
initialized = latch.try_set_count(1)
print("Initialized:", initialized)
# Check count
count = latch.get_count()
print("Count:", count)
# Prints:
# Count: 1

# Bring the count down to zero after 10ms
def run():
    time.sleep(0.01)
    latch.count_down()

t = Thread(target=run)
t.start()

# Wait up to 1 second for the count to become zero up
count_is_zero = latch.await(1)
print("Count is zero:", count_is_zero)
```

Note: CountDownLatch count can be reset with `try_set_count()` after a countdown has finished, but not during an active count.

Using AtomicReference

Hazelcast `AtomicReference` is the distributed implementation of a linearizable object reference. It provides a set of atomic operations allowing to modify the value behind the reference. This data structure is a part of CP Subsystem.

A basic AtomicReference usage example is shown below.

```
# Get a AtomicReference called "my-ref"
my_ref = client.cp_subsystem.get_atomic_reference("my-ref").blocking()
# Set the value atomically
my_ref.set(42)
# Read the value
value = my_ref.get()
print("Value:", value)
# Prints:
```

(continues on next page)

(continued from previous page)

```
# Value: 42

# Try to replace the value with "value"
# with a compare-and-set atomic operation
result = my_ref.compare_and_set(42, "value")
print("CAS result:", result)
# Prints:
# CAS result: True
```

The following are some considerations you need to know when you use AtomicReference:

- AtomicReference works based on the byte-content and not on the object-reference. If you use the `compare_and_set()` method, do not change to the original value because its serialized content will then be different.
- All methods returning an object return a private copy. You can modify the private copy, but the rest of the world is shielded from your changes. If you want these changes to be visible to the rest of the world, you need to write the change back to the AtomicReference; but be careful about introducing a data-race.
- The in-memory format of an AtomicReference is `binary`. The receiving side does not need to have the class definition available unless it needs to be deserialized on the other side, e.g., because a method like `alter()` is executed. This deserialization is done for every call that needs to have the object instead of the binary content, so be careful with expensive object graphs that need to be deserialized.
- If you have an object with many fields or an object graph and you only need to calculate some information or need a subset of fields, you can use the `apply()` method. With the `apply()` method, the whole object does not need to be sent over the line; only the information that is relevant is sent.

AtomicReference does not offer exactly-once / effectively-once execution semantics. It goes with at-least-once execution semantics by default and can cause an API call to be committed multiple times in case of CP member failures. It can be tuned to offer at-most-once execution semantics. Please see fail-on-indeterminate-operation-state server-side setting.

2.9.5 Distributed Events

This chapter explains when various events are fired and describes how you can add event listeners on a Hazelcast Python client. These events can be categorized as cluster and distributed data structure events.

Cluster Events

You can add event listeners to a Hazelcast Python client. You can configure the following listeners to listen to the events on the client side:

- Membership Listener: Notifies when a member joins to/leaves the cluster.
- Lifecycle Listener: Notifies when the client is starting, started, connected, disconnected, shutting down and shutdown.

Listening for Member Events

You can add the following types of member events to the `ClusterService`.

- `member_added`: A new member is added to the cluster.
- `member_removed`: An existing member leaves the cluster.

The `ClusterService` class exposes an `add_listener()` method that allows one or more functions to be attached to the member events emitted by the class.

The following is a membership listener registration by using the `add_listener()` method.

```
def added_listener(member):
    print("Member Added: The address is", member.address)


def removed_listener(member):
    print("Member Removed. The address is", member.address)


client.cluster_service.add_listener(
    member_added=added_listener,
    member_removed=removed_listener,
    fire_for_existing=True
)
```

Also, you can set the `fire_for_existing` flag to `True` to receive the events for list of available members when the listener is registered.

Membership listeners can also be added during the client startup using the `membership_listeners` argument.

```
client = hazelcast.HazelcastClient(
    membership_listeners=[
        (added_listener, removed_listener)
    ]
)
```

Listening for Distributed Object Events

The events for distributed objects are invoked when they are created and destroyed in the cluster. When an event is received, listener function will be called. The parameter passed into the listener function will be of the type `DistributedObjectEvent`. A `DistributedObjectEvent` contains the following fields:

- `name`: Name of the distributed object.
- `service_name`: Service name of the distributed object.
- `event_type`: Type of the invoked event. It is either `CREATED` or `DESTROYED`.

The following is example of adding a distributed object listener to a client.

```
def distributed_object_listener(event):
    print("Distributed object event >>>", event.name, event.service_name, event.event_
↪type)


client.add_distributed_object_listener(
```

(continues on next page)

(continued from previous page)

```
    listener_func=distributed_object_listener
).result()

map_name = "test_map"

# This call causes a CREATED event
test_map = client.get_map(map_name).blocking()

# This causes no event because map was already created
test_map2 = client.get_map(map_name).blocking()

# This causes a DESTROYED event
test_map.destroy()
```

Output

```
Distributed object event >>> test_map hz:impl:mapService CREATED
Distributed object event >>> test_map hz:impl:mapService DESTROYED
```

Listening for Lifecycle Events

The lifecycle listener is notified for the following events:

- `STARTING`: The client is starting.
- `STARTED`: The client has started.
- `CONNECTED`: The client connected to a member.
- `SHUTTING_DOWN`: The client is shutting down.
- `DISCONNECTED`: The client disconnected from a member.
- `SHUTDOWN`: The client has shutdown.

The following is an example of the lifecycle listener that is added to client during startup and its output.

```
def lifecycle_listener(state):
    print("Lifecycle Event >>>", state)


client = hazelcast.HazelcastClient(
    lifecycle_listeners=[
        lifecycle_listener
    ]
)
```

Output:

```
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is STARTING
Lifecycle Event >>> STARTING
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is STARTED
Lifecycle Event >>> STARTED
INFO:hazelcast.connection:Trying to connect to Address(host=127.0.0.1, port=5701)
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is CONNECTED
Lifecycle Event >>> CONNECTED
INFO:hazelcast.connection:Authenticated with server Address(host=172.17.0.2,␣
↪port=5701):7682c357-3bec-4841-b330-6f9ae0c08253, server version: 4.0, local␣
↪address: Address(host=127.0.0.1, port=56732)
```

(continues on next page)

(continued from previous page)

```
INFO:hazelcast.cluster:

Members [1] {
    Member [172.17.0.2]:5701 - 7682c357-3bec-4841-b330-6f9ae0c08253
}

INFO:hazelcast.client:Client started
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is SHUTTING_DOWN
Lifecycle Event >>> SHUTTING_DOWN
INFO:hazelcast.connection:Removed connection to Address(host=127.0.0.1,␣
↪port=5701):7682c357-3bec-4841-b330-6f9ae0c08253, connection: Connection(id=0,␣
↪live=False, remote_address=Address(host=172.17.0.2, port=5701))
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is DISCONNECTED
Lifecycle Event >>> DISCONNECTED
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is SHUTDOWN
Lifecycle Event >>> SHUTDOWN
```

You can also add lifecycle listeners after client initialization using the `LifecycleService`.

```
client.lifecycle_service.add_listener(lifecycle_listener)
```

Distributed Data Structure Events

You can add event listeners to the distributed data structures.

Listening for Map Events

You can listen to map-wide or entry-based events by attaching functions to the `Map` objects using the `add_entry_listener()` method. You can listen the following events.

- `added_func` : Function to be called when an entry is added to map.
- `removed_func` : Function to be called when an entry is removed from map.
- `updated_func` : Function to be called when an entry is updated.
- `evicted_func` : Function to be called when an entry is evicted from map.
- `evict_all_func` : Function to be called when entries are evicted from map.
- `clear_all_func` : Function to be called when entries are cleared from map.
- `merged_func` : Function to be called when WAN replicated entry is merged.
- `expired_func` : Function to be called when an entry's live time is expired.

You can also filter the events using `key` or `predicate`. There is also an option called `include_value`. When this option is set to true, event will also include the value.

An entry-based event is fired after the operations that affect a specific entry. For example, `map.put()`, `map.remove()` or `map.evict()`. An `EntryEvent` object is passed to the listener function.

See the following example.

```
def added(event):
    print("Entry Added: %s-%s" % (event.key, event.value))
```

(continues on next page)

(continued from previous page)

```
customer_map.add_entry_listener(include_value=True, added_func=added)
customer_map.put("4", "Jane Doe")
```

A map-wide event is fired as a result of a map-wide operation. For example, `map.clear()` or `map.evict_all()`. An `EntryEvent` object is passed to the listener function.

See the following example.

```
def cleared(event):
    print("Map Cleared:", event.number_of_affected_entries)


customer_map.add_entry_listener(include_value=True, clear_all_func=cleared)
customer_map.clear()
```

2.9.6 Distributed Computing

This chapter explains how you can use Hazelcast IMDG's entry processor implementation in the Python client.

Using EntryProcessor

Hazelcast supports entry processing. An entry processor is a function that executes your code on a map entry in an atomic way.

An entry processor is a good option if you perform bulk processing on a `Map`. Usually you perform a loop of keys – executing `Map.get(key)`, mutating the value, and finally putting the entry back in the map using `Map.put(key, value)`. If you perform this process from a client or from a member where the keys do not exist, you effectively perform two network hops for each update: the first to retrieve the data and the second to update the mutated value.

If you are doing the process described above, you should consider using entry processors. An entry processor executes a read and updates upon the member where the data resides. This eliminates the costly network hops described above.

Note: Entry processor is meant to process a single entry per call. Processing multiple entries and data structures in an entry processor is not supported as it may result in deadlocks on the server side.

Hazelcast sends the entry processor to each cluster member and these members apply it to the map entries. Therefore, if you add more members, your processing completes faster.

Processing Entries

The `Map` class provides the following methods for entry processing:

- `execute_on_key` processes an entry mapped by a key.
- `execute_on_keys` processes entries mapped by a list of keys.
- `execute_on_entries` can process all entries in a map with a defined predicate. Predicate is optional.

In the Python client, an `EntryProcessor` should be `IdentifiedDataSerializable` or `Portable` because the server should be able to deserialize it to process.

The following is an example for `EntryProcessor` which is an `IdentifiedDataSerializable`.

```
from hazelcast.serialization.api import IdentifiedDataSerializable

class IdentifiedEntryProcessor(IdentifiedDataSerializable):
    def __init__(self, value=None):
        self.value = value

    def read_data(self, object_data_input):
        self.value = object_data_input.read_utf()

    def write_data(self, object_data_output):
        object_data_output.write_utf(self.value)

    def get_factory_id(self):
        return 5

    def get_class_id(self):
        return 1
```

Now, you need to make sure that the Hazelcast member recognizes the entry processor. For this, you need to implement the Java equivalent of your entry processor and its factory, and create your own compiled class or JAR files. For adding your own compiled class or JAR files to the server's `CLASSPATH`, see the *Adding User Library to CLASSPATH* section.

The following is the Java equivalent of the entry processor in Python client given above:

```
import com.hazelcast.map.EntryProcessor;
import com.hazelcast.nio.ObjectDataInput;
import com.hazelcast.nio.ObjectDataOutput;
import com.hazelcast.nio.serialization.IdentifiedDataSerializable;

import java.io.IOException;
import java.util.Map;


public class IdentifiedEntryProcessor
        implements EntryProcessor<String, String, String>, IdentifiedDataSerializable
↪{

    static final int CLASS_ID = 1;
    private String value;

    public IdentifiedEntryProcessor() {
    }

    @Override
    public int getFactoryId() {
        return IdentifiedFactory.FACTORY_ID;
    }

    @Override
    public int getClassId() {
        return CLASS_ID;
    }

    @Override
    public void writeData(ObjectDataOutput out) throws IOException {
        out.writeUTF(value);
```

(continues on next page)

(continued from previous page)

```
    }

    @Override
    public void readData(ObjectDataInput in) throws IOException {
        value = in.readUTF();
    }

    @Override
    public String process(Map.Entry<String, String> entry) {
        entry.setValue(value);
        return value;
    }
}
```

You can implement the above processor's factory as follows:

```
import com.hazelcast.nio.serialization.DataSerializableFactory;
import com.hazelcast.nio.serialization.IdentifiedDataSerializable;

public class IdentifiedFactory implements DataSerializableFactory {
    public static final int FACTORY_ID = 5;

    @Override
    public IdentifiedDataSerializable create(int typeId) {
        if (typeId == IdentifiedEntryProcessor.CLASS_ID) {
            return new IdentifiedEntryProcessor();
        }
        return null;
    }
}
```

Now you need to configure the `hazelcast.xml` to add your factory as shown below.

```
<hazelcast>
    <serialization>
        <data-serializable-factories>
            <data-serializable-factory factory-id="5">
                IdentifiedFactory
            </data-serializable-factory>
        </data-serializable-factories>
    </serialization>
</hazelcast>
```

The code that runs on the entries is implemented in Java on the server side. The client side entry processor is used to specify which entry processor should be called. For more details about the Java implementation of the entry processor, see the Entry Processor section in the Hazelcast IMDG Reference Manual.

After the above implementations and configuration are done and you start the server where your library is added to its `CLASSPATH`, you can use the entry processor in the `Map` methods. See the following example.

```
distributed_map = client.get_map("my-distributed-map").blocking()

distributed_map.put("key", "not-processed")
distributed_map.execute_on_key("key", IdentifiedEntryProcessor("processed"))

print(distributed_map.get("key"))  # Outputs 'processed'
```

2.9.7 Distributed Query

Hazelcast partitions your data and spreads it across cluster of members. You can iterate over the map entries and look for certain entries (specified by predicates) you are interested in. However, this is not very efficient because you will have to bring the entire entry set and iterate locally. Instead, Hazelcast allows you to run distributed queries on your distributed map.

How Distributed Query Works

1. The requested predicate is sent to each member in the cluster.
2. Each member looks at its own local entries and filters them according to the predicate. At this stage, key-value pairs of the entries are deserialized and then passed to the predicate.
3. The predicate requester merges all the results coming from each member into a single set.

Distributed query is highly scalable. If you add new members to the cluster, the partition count for each member is reduced and thus the time spent by each member on iterating its entries is reduced. In addition, the pool of partition threads evaluates the entries concurrently in each member, and the network traffic is also reduced since only filtered data is sent to the requester.

Predicate Module Operators

The `predicate` module offered by the Python client includes many operators for your query requirements. Some of them are explained below.

- `equal`: Checks if the result of an expression is equal to a given value.
- `not_equal`: Checks if the result of an expression is not equal to a given value.
- `instance_of`: Checks if the result of an expression has a certain type.
- `like`: Checks if the result of an expression matches some string pattern. `%` (percentage sign) is the placeholder for many characters, _ (underscore) is placeholder for only one character.
- `ilike`: Checks if the result of an expression matches some string pattern in a case-insensitive manner.
- `greater`: Checks if the result of an expression is greater than a certain value.
- `greater_or_equal`: Checks if the result of an expression is greater than or equal to a certain value.
- `less`: Checks if the result of an expression is less than a certain value.
- `less_or_equal`: Checks if the result of an expression is less than or equal to a certain value.
- `between`: Checks if the result of an expression is between two values (this is inclusive).
- `in_`: Checks if the result of an expression is an element of a certain list.
- `not_`: Checks if the result of an expression is false.
- `regex`: Checks if the result of an expression matches some regular expression.
- `true`: Creates an always true predicate that will pass all items.
- `false`: Creates an always false predicate that will filter out all items.

Hazelcast offers the following ways for distributed query purposes:

- Combining Predicates with AND, OR, NOT
- Distributed SQL Query

Employee Map Query Example

Assume that you have an `employee` map containing the instances of `Employee` class, as coded below.

```
from hazelcast.serialization.api import Portable

class Employee(Portable):
    def __init__(self, name=None, age=None, active=None, salary=None):
        self.name = name
        self.age = age
        self.active = active
        self.salary = salary

    def get_class_id(self):
        return 100

    def get_factory_id(self):
        return 1000

    def read_portable(self, reader):
        self.name = reader.read_utf("name")
        self.age = reader.read_int("age")
        self.active = reader.read_boolean("active")
        self.salary = reader.read_double("salary")

    def write_portable(self, writer):
        writer.write_utf("name", self.name)
        writer.write_int("age", self.age)
        writer.write_boolean("active", self.active)
        writer.write_double("salary", self.salary)
```

Note that `Employee` extends `Portable`. As portable types are not deserialized on the server side for querying, you don't need to implement its Java equivalent on the server side.

For types that are not portable, you need to implement its Java equivalent and its data serializable factory on the server side for server to reconstitute the objects from binary formats. In this case, you need to compile the `Employee` and related factory classes with server's `CLASSPATH` and add them to the `user-lib` directory in the extracted `hazelcast-<version>.zip` (or `tar`) before starting the server. See the *Adding User Library to CLASSPATH* section.

Note: Querying with `Portable` class is faster as compared to `IdentifiedDataSerializable`.

Querying by Combining Predicates with AND, OR, NOT

You can combine predicates by using the `and_`, `or_` and `not_` operators, as shown in the below example.

```
from hazelcast.predicate import and_, equal, less

employee_map = client.get_map("employee")

predicate = and_(equal('active', True), less('age', 30))

employees = employee_map.values(predicate).result()
```

In the above example code, `predicate` verifies whether the entry is active and its `age` value is less than 30. This `predicate` is applied to the `employee` map using the `Map.values` method. This method sends the predicate to all cluster members and merges the results coming from them.

Note: Predicates can also be applied to `key_set` and `entry_set` of the Hazelcast IMDG's distributed map.

Querying with SQL

`SqlPredicate` takes the regular SQL `where` clause. See the following example:

```
from hazelcast.predicate import sql

employee_map = client.get_map("employee")

employees = employee_map.values(sql("active AND age < 30")).result()
```

Supported SQL Syntax

AND/OR: `<expression> AND <expression> AND <expression>...`

- `active AND age > 30`
- `active = false OR age = 45 OR name = 'Joe'`
- `active AND ( age > 20 OR salary < 60000 )`

Equality: `=, !=, <, , >, >=`

- `<expression> = value`
- `age <= 30`
- `name = 'Joe'`
- `salary != 50000`

BETWEEN: `<attribute> [NOT] BETWEEN <value1> AND <value2>`

- `age BETWEEN 20 AND 33 ( same as age >= 20 AND age  33 )`
- `age NOT BETWEEN 30 AND 40 ( same as age < 30 OR age > 40 )`

IN: `<attribute> [NOT] IN (val1, val2,...)`

- `age IN ( 20, 30, 40 )`
- `age NOT IN ( 60, 70 )`
- `active AND ( salary >= 50000 OR ( age NOT BETWEEN 20 AND 30 ) )`
- `age IN ( 20, 30, 40 ) AND salary BETWEEN ( 50000, 80000 )`

LIKE: `<attribute> [NOT] LIKE 'expression'`

The `%` (percentage sign) is the placeholder for multiple characters, an _ (underscore) is the placeholder for only one character.

- `name LIKE 'Jo%'` (true for 'Joe', 'Josh', 'Joseph' etc.)
- `name LIKE 'Jo_'` (true for 'Joe'; false for 'Josh')

- `name NOT LIKE 'Jo_'` (true for 'Josh'; false for 'Joe')
- `name LIKE 'J_s%'` (true for 'Josh', 'Joseph'; false 'John', 'Joe')

ILIKE: `<attribute> [NOT] ILIKE 'expression'`

ILIKE is similar to the LIKE predicate but in a case-insensitive manner.

- `name ILIKE 'Jo%'` (true for 'Joe', 'joe', 'jOe','Josh','joSH', etc.)
- `name ILIKE 'Jo_'` (true for 'Joe' or 'jOE'; false for 'Josh')

REGEX: `<attribute> [NOT] REGEX 'expression'`

- `name REGEX 'abc-.*'` (true for 'abc-123'; false for 'abx-123')

Querying Examples with Predicates

You can use the `__key` attribute to perform a predicated search for the entry keys. See the following example:

```
from hazelcast.predicate import sql

person_map = client.get_map("persons").blocking()

person_map.put("John", 28)
person_map.put("Mary", 23)
person_map.put("Judy", 30)

predicate = sql("__key like M%")

persons = person_map.values(predicate)

print(persons[0]) # Outputs '23'
```

In this example, the code creates a list with the values whose keys start with the letter "M".

You can use the `this` attribute to perform a predicated search for the entry values. See the following example:

```
from hazelcast.predicate import greater_or_equal

person_map = client.get_map("persons").blocking()

person_map.put("John", 28)
person_map.put("Mary", 23)
person_map.put("Judy", 30)

predicate = greater_or_equal("this", 27)

persons = person_map.values(predicate)

print(persons[0], persons[1]) # Outputs '28 30'
```

In this example, the code creates a list with the values greater than or equal to "27".

Querying with JSON Strings

You can query JSON strings stored inside your Hazelcast clusters. To query the JSON string, you first need to create a `HazelcastJsonValue` from the JSON string or JSON serializable object. You can use `HazelcastJsonValues` both as keys and values in the distributed data structures. Then, it is possible to query these objects using the Hazelcast query methods explained in this section.

```
person1 = "{ \"name\": \"John\", \"age\": 35 }"
person2 = "{ \"name\": \"Jane\", \"age\": 24 }"
person3 = {"name": "Trey", "age": 17}

id_person_map = client.get_map("json-values").blocking()

# From JSON string
id_person_map.put(1, HazelcastJsonValue(person1))
id_person_map.put(2, HazelcastJsonValue(person2))

# From JSON serializable object
id_person_map.put(3, HazelcastJsonValue(person3))

people_under_21 = id_person_map.values(less("age", 21))
```

When running the queries, Hazelcast treats values extracted from the JSON documents as Java types so they can be compared with the query attribute. JSON specification defines five primitive types to be used in the JSON documents: `number`,`string`, `true`, `false` and `null`. The `string`, `true/false` and `null` types are treated as `String`, `boolean` and `null`, respectively. We treat the extracted `number` values as `longs` if they can be represented by a `long`. Otherwise, `numbers` are treated as `doubles`.

It is possible to query nested attributes and arrays in the JSON documents. The query syntax is the same as querying other Hazelcast objects using the `Predicates`.

```
# Sample JSON object
# {
#     "departmentId": 1,
#     "room": "alpha",
#     "people": [
#         {
#             "name": "Peter",
#             "age": 26,
#             "salary": 50000
#         },
#         {
#             "name": "Jonah",
#             "age": 50,
#             "salary": 140000
#         }
#     ]
# }
# The following query finds all the departments that have a person named "Peter"␣
↪working in them.

department_with_peter = departments.values(equal("people[any].name", "Peter"))
```

`HazelcastJsonValue` is a lightweight wrapper around your JSON strings. It is used merely as a way to indicate that the contained string should be treated as a valid JSON value. Hazelcast does not check the validity of JSON strings put into to the maps. Putting an invalid JSON string into a map is permissible. However, in that case whether such an entry is going to be returned or not from a query is not defined.

Metadata Creation for JSON Querying

Hazelcast stores a metadata object per JSON serialized object stored. This metadata object is created every time a JSON serialized object is put into an `Map`. Metadata is later used to speed up the query operations. Metadata creation is on by default. Depending on your application's needs, you may want to turn off the metadata creation to decrease the put latency and increase the throughput.

You can configure this using `metadata-policy` element for the map configuration on the member side as follows:

```
<hazelcast>
    ...
    <map name="map-a">
        <!--
        valid values for metadata-policy are:
          - OFF
          - CREATE_ON_UPDATE (default)
        -->
        <metadata-policy>OFF</metadata-policy>
    </map>
    ...
</hazelcast>
```

Filtering with Paging Predicates

Hazelcast Python client provides paging for defined predicates. With its `PagingPredicate`, you can get a collection of keys, values, or entries page by page by filtering them with predicates and giving the size of the pages. Also, you can sort the entries by specifying comparators. In this case, the comparator should be either `Portable` or `IdentifiedDataSerializable` and the serialization factory implementations should be registered on the member side. Please note that, paging is done on the cluster members. Hence, client only sends a marker comparator to indicate members which comparator to use. The comparision logic must be defined on the member side by implementing the `java.util.Comparator<Map.Entry>` interface.

Paging predicates require the objects to be deserialized on the member side from which the collection is retrieved. Therefore, you need to register the serialization factories you use on all the members on which the paging predicates are used. See the *Adding User Library to CLASSPATH* section for more details.

In the example code below:

- The `greater_or_equal` predicate gets values from the `students` map. This predicate has a filter to retrieve the objects with an `age` greater than or equal to `18`.
- Then a `PagingPredicate` is constructed in which the page size is `5`, so that there are five objects in each page. The first time the `values()` method is called, the first page is fetched.
- Finally, the subsequent page is fetched by calling the `next_page()` method of `PagingPredicate` and querying the map again with the updated `PagingPredicate`.

```
from hazelcast.predicate import paging, greater_or_equal

...

m = client.get_map("students").blocking()
predicate = paging(greater_or_equal("age", 18), 5)

# Retrieve the first page
values = m.values(predicate)
```

(continues on next page)

(continued from previous page)

```
...

# Set up next page
predicate.next_page()

# Retrieve next page
values = m.values(predicate)
```

If a comparator is not specified for `PagingPredicate`, but you want to get a collection of keys or values page by page, keys or values must implement the `java.lang.Comparable` interface on the member side. Otherwise, paging fails with an exception from the server. Luckily, a lot of types implement the `Comparable` interface by default, including the primitive types, so, you may use values of types `int`, `float`, `str` etc. in paging without specifying a comparator on the Python client.

You can also access a specific page more easily by setting the `predicate.page` attribute before making the remote call. This way, if you make a query for the hundredth page, for example, it gets all `100` pages at once instead of reaching the hundredth page one by one using the `next_page()` method.

Note: `PagingPredicate`, also known as Order & Limit, is not supported in Transactional Context.

2.9.8 Performance

Near Cache

Map entries in Hazelcast are partitioned across the cluster members. Hazelcast clients do not have local data at all. Suppose you read the key `k` a number of times from a Hazelcast client and `k` is owned by a member in your cluster. Then each `map.get(k)` will be a remote operation, which creates a lot of network trips. If you have a map that is mostly read, then you should consider creating a local Near Cache, so that reads are sped up and less network traffic is created.

These benefits do not come for free, please consider the following trade-offs:

- Clients with a Near Cache will have to hold the extra cached data, which increases memory consumption.
- If invalidation is enabled and entries are updated frequently, then invalidations will be costly.
- Near Cache breaks the strong consistency guarantees; you might be reading stale data.

Near Cache is highly recommended for maps that are mostly read.

Configuring Near Cache

The following snippet show how a Near Cache is configured in the Python client using the `near_caches` argument, presenting all available values for each element. When an element is missing from the configuration, its default value is used.

```
from hazelcast.config import InMemoryFormat, EvictionPolicy

client = hazelcast.HazelcastClient(
    near_caches={
        "mostly-read-map": {
            "invalidate_on_change": True,
```

(continues on next page)

(continued from previous page)

```
            "time_to_live": 60,
            "max_idle": 30,
            # You can also set these to "OBJECT"
            # and "LRU" without importing anything.
            "in_memory_format": InMemoryFormat.OBJECT,
            "eviction_policy": EvictionPolicy.LRU,
            "eviction_max_size": 100,
            "eviction_sampling_count": 8,
            "eviction_sampling_pool_size": 16
        }
    }
)
```

Following are the descriptions of all configuration elements:

- `in_memory_format`: Specifies in which format data will be stored in your Near Cache. Note that a map's in-memory format can be different from that of its Near Cache. Available values are as follows:
  - `BINARY`: Data will be stored in serialized binary format (default value).
  - `OBJECT`: Data will be stored in deserialized format.
- `invalidate_on_change`: Specifies whether the cached entries are evicted when the entries are updated or removed. Its default value is `True`.
- `time_to_live`: Maximum number of seconds for each entry to stay in the Near Cache. Entries that are older than this period are automatically evicted from the Near Cache. Regardless of the eviction policy used, `time_to_live_seconds` still applies. Any non-negative number can be assigned. Its default value is `None`. `None` means infinite.
- `max_idle`: Maximum number of seconds each entry can stay in the Near Cache as untouched (not read). Entries that are not read more than this period are removed from the Near Cache. Any non-negative number can be assigned. Its default value is `None`. `None` means infinite.
- `eviction_policy`: Eviction policy configuration. Available values are as follows:
  - `LRU`: Least Recently Used (default value).
  - `LFU`: Least Frequently Used.
  - `NONE`: No items are evicted and the `eviction_max_size` property is ignored. You still can combine it with `time_to_live` and `max_idle` to evict items from the Near Cache.
  - `RANDOM`: A random item is evicted.
- `eviction_max_size`: Maximum number of entries kept in the memory before eviction kicks in.
- `eviction_sampling_count`: Number of random entries that are evaluated to see if some of them are already expired. If there are expired entries, those are removed and there is no need for eviction.
- `eviction_sampling_pool_size`: Size of the pool for eviction candidates. The pool is kept sorted according to eviction policy. The entry with the highest score is evicted.

Near Cache Example for Map

The following is an example configuration for a Near Cache defined in the `mostly-read-map` map. According to this configuration, the entries are stored as `OBJECT`'s in this Near Cache and eviction starts when the count of entries reaches `5000`; entries are evicted based on the `LRU` (Least Recently Used) policy. In addition, when an entry is updated or removed on the member side, it is eventually evicted on the client side.

```
client = hazelcast.HazelcastClient(
    near_caches={
        "mostly-read-map": {
            "invalidate_on_change": True,
            "in_memory_format": InMemoryFormat.OBJECT,
            "eviction_policy": EvictionPolicy.LRU,
            "eviction_max_size": 5000,
        }
    }
)
```

Near Cache Eviction

In the scope of Near Cache, eviction means evicting (clearing) the entries selected according to the given `eviction_policy` when the specified `eviction_max_size` has been reached.

The `eviction_max_size` defines the entry count when the Near Cache is full and determines whether the eviction should be triggered.

Once the eviction is triggered, the configured `eviction_policy` determines which, if any, entries must be evicted.

Near Cache Expiration

Expiration means the eviction of expired records. A record is expired:

- If it is not touched (accessed/read) for `max_idle` seconds
- `time_to_live` seconds passed since it is put to Near Cache

The actual expiration is performed when a record is accessed: it is checked if the record is expired or not. If it is expired, it is evicted and `KeyError` is raised to the caller.

Near Cache Invalidation

Invalidation is the process of removing an entry from the Near Cache when its value is updated or it is removed from the original map (to prevent stale reads). See the Near Cache Invalidation section in the Hazelcast IMDG Reference Manual.

2.9.9 Monitoring and Logging

Enabling Client Statistics

You can monitor your clients using Hazelcast Management Center.

As a prerequisite, you need to enable the client statistics before starting your clients. There are two arguments of `HazelcastClient` related to client statistics:

- `statistics_enabled`: If set to `True`, it enables collecting the client statistics and sending them to the cluster. When it is `True` you can monitor the clients that are connected to your Hazelcast cluster, using Hazelcast Management Center. Its default value is `False`.
- `statistics_period`: Period in seconds the client statistics are collected and sent to the cluster. Its default value is `3`.

You can enable client statistics and set a non-default period in seconds as follows:

```
client = hazelcast.HazelcastClient(
    statistics_enabled=True,
    statistics_period=4
)
```

Hazelcast Python client can collect statistics related to the client and Near Caches without an extra dependency. However, to get the statistics about the runtime and operating system, psutil is used as an extra dependency.

If the `psutil` is installed, runtime and operating system statistics will be sent to cluster along with statistics related to the client and Near Caches. If not, only the client and Near Cache statistics will be sent.

`psutil` can be installed independently or with the Hazelcast Python client as follows:

From PyPI

```
pip install hazelcast-python-client[stats]
```

From source

```
pip install -e .[stats]
```

After enabling the client statistics, you can monitor your clients using Hazelcast Management Center. Please refer to the Monitoring Clients section in the Hazelcast Management Center Reference Manual for more information on the client statistics.

Note: Statistics sent by Hazelcast Python client 4.0 are compatible with Management Center 4.0. Management Center 4.2020.08 and newer versions will be supported in version 4.1 of the client.

Logging Configuration

Hazelcast Python client uses Python's builtin `logging` package to perform logging.

All the loggers used throughout the client are identified by their module names. Hence, one may configure the `hazelcast` parent logger and use the same configuration for the child loggers such as `hazelcast.lifecycle` without an extra effort.

Below is an example of the logging configuration with `INFO` log level and a `StreamHandler` with a custom format, and its output.

```
import logging
import hazelcast

logger = logging.getLogger("hazelcast")
logger.setLevel(logging.INFO)

handler = logging.StreamHandler()
formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
handler.setFormatter(formatter)
logger.addHandler(handler)

client = hazelcast.HazelcastClient()

client.shutdown()
```

Output

```
2020-10-16 13:31:35,605 - hazelcast.lifecycle - INFO - HazelcastClient 4.0.0 is␣
↪STARTING
2020-10-16 13:31:35,605 - hazelcast.lifecycle - INFO - HazelcastClient 4.0.0 is␣
↪STARTED
2020-10-16 13:31:35,605 - hazelcast.connection - INFO - Trying to connect to␣
↪Address(host=127.0.0.1, port=5701)
2020-10-16 13:31:35,622 - hazelcast.lifecycle - INFO - HazelcastClient 4.0.0 is␣
↪CONNECTED
2020-10-16 13:31:35,622 - hazelcast.connection - INFO - Authenticated with server␣
↪Address(host=172.17.0.2, port=5701):7682c357-3bec-4841-b330-6f9ae0c08253, server␣
↪version: 4.0, local address: Address(host=127.0.0.1, port=56752)
2020-10-16 13:31:35,623 - hazelcast.cluster - INFO -

Members [1] {
    Member [172.17.0.2]:5701 - 7682c357-3bec-4841-b330-6f9ae0c08253
}

2020-10-16 13:31:35,624 - hazelcast.client - INFO - Client started
2020-10-16 13:31:35,624 - hazelcast.lifecycle - INFO - HazelcastClient 4.0.0 is␣
↪SHUTTING_DOWN
2020-10-16 13:31:35,624 - hazelcast.connection - INFO - Removed connection to␣
↪Address(host=127.0.0.1, port=5701):7682c357-3bec-4841-b330-6f9ae0c08253,␣
↪connection: Connection(id=0, live=False, remote_address=Address(host=172.17.0.2,␣
↪port=5701))
2020-10-16 13:31:35,624 - hazelcast.lifecycle - INFO - HazelcastClient 4.0.0 is␣
↪DISCONNECTED
2020-10-16 13:31:35,634 - hazelcast.lifecycle - INFO - HazelcastClient 4.0.0 is␣
↪SHUTDOWN
```

A handy alternative to above example would be configuring the root logger using the `logging.basicConfig()` utility method. Beware that, every logger is the child of the root logger in Python. Hence, configuring the root logger may have application level impact. Nonetheless, it is useful for the testing or development purposes.

```
import logging
import hazelcast

logging.basicConfig(level=logging.INFO)

client = hazelcast.HazelcastClient()
```

(continues on next page)

(continued from previous page)

```
client.shutdown()
```

Output

```
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is STARTING
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is STARTED
INFO:hazelcast.connection:Trying to connect to Address(host=127.0.0.1, port=5701)
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is CONNECTED
INFO:hazelcast.connection:Authenticated with server Address(host=172.17.0.2,␣
↪port=5701):7682c357-3bec-4841-b330-6f9ae0c08253, server version: 4.0, local␣
↪address: Address(host=127.0.0.1, port=56758)
INFO:hazelcast.cluster:

Members [1] {
    Member [172.17.0.2]:5701 - 7682c357-3bec-4841-b330-6f9ae0c08253
}

INFO:hazelcast.client:Client started
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is SHUTTING_DOWN
INFO:hazelcast.connection:Removed connection to Address(host=127.0.0.1,␣
↪port=5701):7682c357-3bec-4841-b330-6f9ae0c08253, connection: Connection(id=0,␣
↪live=False, remote_address=Address(host=172.17.0.2, port=5701))
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is DISCONNECTED
INFO:hazelcast.lifecycle:HazelcastClient 4.0.0 is SHUTDOWN
```

To learn more about the `logging` package and its capabilities, please see the logging cookbook and documentation of the `logging` package.

2.9.10 Defining Client Labels

Through the client labels, you can assign special roles for your clients and use these roles to perform some actions specific to those client connections.

You can also group your clients using the client labels. These client groups can be blacklisted in Hazelcast Management Center so that they can be prevented from connecting to a cluster. See the related section in the Hazelcast Management Center Reference Manual for more information on this topic.

You can define the client labels using the `labels` config option. See the below example.

```
client = hazelcast.HazelcastClient(
    labels=[
        "role admin",
        "region foo"
    ]
)
```

2.9.11 Defining Client Name

Each client has a name associated with it. By default, it is set to `hz.client_${CLIENT_ID}`. Here `CLIENT_ID` starts from `0` and it is incremented by `1` for each new client. This id is incremented and set by the client, so it may not be unique between different clients used by different applications.

You can set the client name using the `client_name` configuration element.

```
client = hazelcast.HazelcastClient(
    client_name="blue_client_0"
)
```

2.9.12 Configuring Load Balancer

Load Balancer configuration allows you to specify which cluster member to send next operation when queried.

If it is a *Smart Client*, only the operations that are not key-based are routed to the member that is returned by the `LoadBalancer`. If it is not a smart client, `LoadBalancer` is ignored.

By default, client uses round robin load balancer which picks each cluster member in turn. Also, the client provides random load balancer which picks the next member randomly as the name suggests. You can use one of them by setting the `load_balancer` config option.

The following are example configurations.

```
from hazelcast.util import RandomLB

client = hazelcast.HazelcastClient(
    load_balancer=RandomLB()
)
```

You can also provide a custom load balancer implementation to use different load balancing policies. To do so, you should provide a class that implements the `LoadBalancers` interface or extend the `AbstractLoadBalancer` class for that purpose and provide the load balancer object into the `load_balancer` config option.

2.10 Securing Client Connection

This chapter describes the security features of Hazelcast Python client. These include using TLS/SSL for connections between members and between clients and members, and mutual authentication. These security features require **Hazelcast IMDG Enterprise** edition.

2.10.1 TLS/SSL

One of the offers of Hazelcast is the TLS/SSL protocol which you can use to establish an encrypted communication across your cluster with key stores and trust stores.

- A Java `keyStore` is a file that includes a private key and a public certificate. The equivalent of a key store is the combination of `keyfile` and `certfile` at the Python client side.
- A Java `trustStore` is a file that includes a list of certificates trusted by your application which is named certificate authority. The equivalent of a trust store is a `cafile` at the Python client side.

You should set `keyStore` and `trustStore` before starting the members. See the next section on how to set `keyStore` and `trustStore` on the server side.

TLS/SSL for Hazelcast Members

Hazelcast allows you to encrypt socket level communication between Hazelcast members and between Hazelcast clients and members, for end to end encryption. To use it, see the TLS/SSL for Hazelcast Members section.

TLS/SSL for Hazelcast Python Clients

TLS/SSL for the Hazelcast Python client can be configured using the `SSLConfig` class. Let's first give an example of a sample configuration and then go over the configuration options one by one:

```
from hazelcast.config import SSLProtocol

client = hazelcast.HazelcastClient(
    ssl_enabled=True,
    ssl_cafile="/home/hazelcast/cafile.pem",
    ssl_certfile="/home/hazelcast/certfile.pem",
    ssl_keyfile="/home/hazelcast/keyfile.pem",
    ssl_password="keyfile-password",
    # You can also set this to "TLSv1_3"
    # without importing anything.
    ssl_protocol=SSLProtocol.TLSv1_3,
    ssl_ciphers="DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA"
)
```

Enabling TLS/SSL

TLS/SSL for the Hazelcast Python client can be enabled/disabled using the `ssl_enabled` option. When this option is set to `True`, TLS/SSL will be configured with respect to the other SSL options. Setting this option to `False` will result in discarding the other SSL options.

The following is an example configuration:

```
client = hazelcast.HazelcastClient(
    ssl_enabled=True
)
```

Default value is `False` (disabled).

Setting CA File

Certificates of the Hazelcast members can be validated against `ssl_cafile`. This option should point to the absolute path of the concatenated CA certificates in PEM format. When SSL is enabled and `ssl_cafile` is not set, a set of default CA certificates from default locations will be used.

The following is an example configuration:

```
client = hazelcast.HazelcastClient(
    ssl_cafile="/home/hazelcast/cafile.pem"
)
```

Setting Client Certificate

When mutual authentication is enabled on the member side, clients or other members should also provide a certificate file that identifies themselves. Then, Hazelcast members can use these certificates to validate the identity of their peers.

Client certificate can be set using the `ssl_certfile`. This option should point to the absolute path of the client certificate in PEM format.

The following is an example configuration:

```
client = hazelcast.HazelcastClient(
    ssl_certfile="/home/hazelcast/certfile.pem"
)
```

Setting Private Key

Private key of the `ssl_certfile` can be set using the `ssl_keyfile`. This option should point to the absolute path of the private key file for the client certificate in the PEM format.

If this option is not set, private key will be taken from `ssl_certfile`. In this case, `ssl_certfile` should be in the following format.

```
-----BEGIN RSA PRIVATE KEY-----
... (private key in base64 encoding) ...
-----END RSA PRIVATE KEY-----
-----BEGIN CERTIFICATE-----
... (certificate in base64 PEM encoding) ...
-----END CERTIFICATE-----
```

The following is an example configuration:

```
client = hazelcast.HazelcastClient(
    ssl_keyfile="/home/hazelcast/keyfile.pem"
)
```

Setting Password of the Private Key

If the private key is encrypted using a password, `ssl_password` will be used to decrypt it. The `ssl_password` may be a function to call to get the password. In that case, it will be called with no arguments, and it should return a string, bytes or bytearray. If the return value is a string it will be encoded as UTF-8 before using it to decrypt the key.

Alternatively a string, `bytes` or `bytearray` value may be supplied directly as the password.

The following is an example configuration:

```
client = hazelcast.HazelcastClient(
    ssl_password="keyfile-password"
)
```

Setting the Protocol

`ssl_protocol` can be used to select the protocol that will be used in the TLS/SSL communication. Hazelcast Python client offers the following protocols:

- **SSLv2** : SSL 2.0 Protocol. *RFC 6176 prohibits the usage of SSL 2.0.*
- **SSLv3** : SSL 3.0 Protocol. *RFC 7568 prohibits the usage of SSL 3.0.*
- **TLSv1** : TLS 1.0 Protocol described in RFC 2246
- **TLSv1_1** : TLS 1.1 Protocol described in RFC 4346
- **TLSv1_2** : TLS 1.2 Protocol described in RFC 5246
- **TLSv1_3** : TLS 1.3 Protocol described in RFC 8446

  Note that TLSv1+ requires at least Python 2.7.9 or Python 3.4 built with OpenSSL 1.0.1+, and TLSv1_3 requires at least Python 2.7.15 or Python 3.7 built with OpenSSL 1.1.1+.

These protocol versions can be selected using the `ssl_protocol` as follows:

```
from hazelcast.config import SSLProtocol

client = hazelcast.HazelcastClient(
    ssl_protocol=SSLProtocol.TLSv1_3
)
```

Note that the Hazelcast Python client and the Hazelcast members should have the same protocol version in order for TLS/SSL to work. In case of the protocol mismatch, connection attempts will be refused.

Default value is `SSLProtocol.TLSv1_2`.

Setting Cipher Suites

Cipher suites that will be used in the TLS/SSL communication can be set using the `ssl_ciphers` option. Cipher suites should be in the OpenSSL cipher list format. More than one cipher suite can be set by separating them with a colon.

TLS/SSL implementation will honor the cipher suite order. So, Hazelcast Python client will offer the ciphers to the Hazelcast members with the given order.

Note that, when this option is not set, all the available ciphers will be offered to the Hazelcast members with their default order.

The following is an example configuration:

```
client = hazelcast.HazelcastClient(
    ssl_ciphers="DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA"
)
```

Mutual Authentication

As explained above, Hazelcast members have key stores used to identify themselves (to other members) and Hazelcast clients have trust stores used to define which members they can trust.

Using mutual authentication, the clients also have their key stores and members have their trust stores so that the members can know which clients they can trust.

To enable mutual authentication, firstly, you need to set the following property on the server side in the `hazelcast.xml` file:

```
<network>
    <ssl enabled="true">
        <properties>
            <property name="javax.net.ssl.mutualAuthentication">REQUIRED</property>
        </properties>
    </ssl>
</network>
```

You can see the details of setting mutual authentication on the server side in the Mutual Authentication section of the Hazelcast IMDG Reference Manual.

On the client side, you have to provide `ssl_cafile`, `ssl_certfile` and `ssl_keyfile` on top of the other TLS/SSL configurations. See the *TLS/SSL for Hazelcast Python Clients* section for the details of these options.

2.11 Development and Testing

If you want to help with bug fixes, develop new features or tweak the implementation to your application's needs, you can follow the steps in this section.

2.11.1 Building and Using Client From Sources

Follow the below steps to build and install Hazelcast Python client from its source:

1. Clone the GitHub repository (https://github.com/hazelcast/hazelcast-python-client.git).
2. Run `python setup.py install` to install the Python client.

If you are planning to contribute:

1. Run `pip install -r requirements-dev.txt` to install development dependencies.
2. Use black to reformat the code by running the `black --config black.toml .` command.
3. Make sure that tests are passing by following the steps described in the *Testing* section.

2.11.2 Testing

In order to test Hazelcast Python client locally, you will need the following:

- Java 8 or newer
- Maven

Following commands starts the tests according to your operating system:

```
bash run-tests.sh
```

or

```
.\run-tests.ps1
```

Test script automatically downloads `hazelcast-remote-controller` and Hazelcast IMDG. The script uses Maven to download those.

2.12 Getting Help

You can use the following channels for your questions and development/usage issues:

- Github Repository
- Slack
- Google Groups
- Stack Overflow

2.13 Contributing

Besides your development contributions as explained in the *Development and Testing* section, you can always open a pull request on this repository for your other requests.

2.14 License

Apache 2 License.

2.15 Copyright

Visit www.hazelcast.com for more information.

PYTHON MODULE INDEX

h

INDEX

D

E

F

G

H

I

K

L

M

N

S

T

U

V

W

X