Python Yamcs Client

Release 1.13.0

Yamcs
MISSION CONTROL

Space Applications Services, NV/SA
Leuvensesteenweg 325
1932 Sint-Stevens-Woluwe
Belgium
spaceapplications.com
yamcs.org

Aerospace Applications North America, Inc.
16850 Saturn Ln, Ste 100
Houston, TX 77058
United States of America
aerospaceapplications-na.com

Contents

Getting Started

Install Python 3.8 or higher.

Install `yamcs-client` from PyPI:

```
pip install --upgrade yamcs-client
```

Usage

Get domain-specific clients:

```
from yamcs.client import YamcsClient
client = YamcsClient('localhost:8090')

mdb = client.get_mdb(instance='simulator')
# ...

archive = client.get_archive(instance='simulator')
# ...

processor = client.get_processor(instance='simulator', processor='realtime')
# ...
```

Documentation

- *General Client*
- *Mission Database*
- *TM/TC Processing*
- *Archive*
- *Link Management*
- *Object Storage*
- *File Transfer*
- *Time Correlation (TCO)*
- *Timeline*

Examples

- *Examples*

1. General Client

1.1 Reference

1.1.1 YamcsClient

class yamcs.client.**YamcsClient**(*address: str*, ** (Keyword-only parameters separator (PEP 3102))*, *tls: bool = False*, *tls_verify: bool | str = True*, *credentials: Credentials | None = None*, *user_agent: str | None = None*, *keep_alive: bool = True*, *on_token_update: Callable[[Credentials], None] | None = None*)

Bases: `object`

Client for accessing core Yamcs resources.

Parameters

- **address** – The address of Yamcs in the format 'hostname:port'
- **tls** – Whether TLS encryption is expected
- **tls_verify** – Whether server certificate verification is enabled (only applicable if `tls=True`). As an alternative to a boolean value, this option may be set to a path containing the appropriate TLS CA certificate bundle.
- **credentials** – Credentials for when the server is secured
- **user_agent** – Optionally override the default user agent
- **keep_alive** – Automatically renew the client session. If disabled, the session will terminate after about 30 minutes of inactivity.

  This property is only considered when accessing a server that requires authentication.

close()

Close this client session

compact_rdb_column_family(*tablespace: str*, *cf: str*, *dbpath: str | None = None*)

Compact a RocksDB column family.

Parameters

- **tablespace** – RocksDB tablespace name
- **cf** – Column family name
- **dbpath** – Optional path under the tablespace root.

create_event_subscription(*instance: str*, *on_data: Callable[[Event], None]*, *filter: str | None = None*, *timeout: float = 60*) → *WebSocketSubscriptionFuture*

Create a new subscription for receiving events of an instance.

This method returns a future, then returns immediately. Stop the subscription by canceling the future.

Parameters

- **instance** – A Yamcs instance name
- **on_data** – Function that gets called on each *Event*.
- **filter** – Filter query applied to returned events. Allows for both text and field search.

  Added in version 1.10.1: Compatible with Yamcs v5.10.2 onwards
- **timeout** – The amount of seconds to wait for the request to complete.

Returns
Future that can be used to manage the background websocket subscription.

create_instance(*name: str*, *template: str*, *args: Mapping[str, Any] | None = None*, *labels: Mapping[str, str] | None = None*)

Create a new instance based on an existing template. This method blocks until the instance is fully started.

Parameters

- **instance** – A Yamcs instance name.
- **template** – The name of an existing template.

create_link_subscription(*instance: str*, *on_data: Callable[[List[*Link*]], None] | None = None*, *timeout: float = 60*) → *LinkSubscription*

Create a new subscription for receiving data link updates of an instance.

This method returns a future, then returns immediately. Stop the subscription by canceling the future.

Parameters

- **instance** – A Yamcs instance name.
- **on_data** – Function that gets called with *Link* updates. Each update contains state of all links.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns
Future that can be used to manage the background websocket subscription.

create_time_subscription(*instance: str*, *on_data: Callable[[datetime], None] | None = None*, *timeout: float = 60*) → *TimeSubscription*

Create a new subscription for receiving time updates of an instance. Time updates are emitted at 1Hz.

This method returns a future, then returns immediately. Stop the subscription by canceling the future.

Parameters

- **instance** – A Yamcs instance name
- **on_data** – Function that gets called with `datetime` updates.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns
Future that can be used to manage the background websocket subscription.

delete_processor(*instance: str*, *processor: str*)

Delete a processor.

Parameters

- **instance** – A Yamcs instance name.
- **processor** – A processor name within that instance.

static from_environment()

Create a *YamcsClient*, initialized from environment variables.

This recognizes the following environment variables:

YAMCS_URL
Yamcs server URL

YAMCS_API_KEY
Yamcs API key (currently only assigned to script activities)

get_archive(*instance: str*) → *ArchiveClient*

Return an object for working with the Archive of the specified instance.

Parameters
instance – A Yamcs instance name.

get_auth_info() → *AuthInfo*

Returns general authentication information. This operation does not require authenticating and is useful to test if a server requires authentication or not.

get_file_transfer_client(*instance: str*) → *FileTransferClient*

Return an object for working with file transfers on a specified instance.

Parameters
instance – A Yamcs instance name.

get_link(*instance: str*, *link: str*) → *LinkClient*

Return an object for working with a specific Yamcs link.

Parameters

- **instance** – A Yamcs instance name.
- **link** – A link name within that instance.

get_mdb(*instance: str*) → *MDBClient*

Return an object for working with the MDB of the specified instance.

Parameters
instance – A Yamcs instance name.

get_processor(*instance: str*, *processor: str*) → *ProcessorClient*

Return an object for working with a specific Yamcs processor.

Parameters

- **instance** – A Yamcs instance name.
- **processor** – A processor name within that instance.

get_server_info() → *ServerInfo*

Return general server info.

get_storage_client() → *StorageClient*

Return an object for working with object storage

get_tco_client(*instance: str*, *service: str*) → *TCOClient*

Return an object for Time Correlation API calls on a specified service.

Parameters

- **instance** – A Yamcs instance name.
- **service** – Target service name.

get_time(*instance*) → datetime | None

Return the current mission time for the specified instance.

get_timeline_client(*instance: str*) → *TimelineClient*

Return an object for working with Yamcs timeline items

Parameters
instance – A Yamcs instance name.

get_user_info() → *UserInfo*

Get information on the authenticated user.

list_instance_templates() → Iterable[*InstanceTemplate*]

List the available instance templates.

list_instances() → Iterable[*Instance*]

Lists the instances.

Instances are returned in lexicographical order.

list_links(*instance: str*) → Iterable[*Link*]

Lists the data links visible to this client.

Data links are returned in random order.

Parameters
instance – A Yamcs instance name.

list_processors(*instance: str | None = None*) → Iterable[*Processor*]

Lists the processors.

Processors are returned in lexicographical order.

Parameters
instance – A Yamcs instance name.

list_rdb_tablespaces() → Iterable[*RdbTablespace*]

Lists RocksDB tablespaces.

list_services(*instance: str*) → Iterable[*Service*]

List the services for an instance.

Parameters
instance – A Yamcs instance name.

load_parameter_values(*instance: str*, *data: Iterator[Mapping[str, 'ValueUpdate']]*, *stream: str = 'pp_dump'*, *chunk_size: int = 32768*) → *LoadParameterValuesResult*

Load an indefinite amount of parameter values into Yamcs.

Added in version 1.9.1: Compatible with Yamcs 5.8.8 onwards

Parameters

- **instance** – Yamcs instance name
- **data** – Generator that yields batches of parameter values, keyed by parameter name.
- **stream** – Stream where to send the parameters to.
- **chunk_size** – HTTP chunk size. Multiple updates are grouped in chunks of about this size.

restart_instance(*instance: str*)

Restarts a single instance.

Parameters
instance – A Yamcs instance name.

send_event(*instance: str*, *message: str*, *event_type: str | None = None*, *time: datetime | None = None*, *severity: str | None = 'info'*, *source: str | None = None*, *sequence_number: int | None = None*, *extra: Mapping[str, str] | None = None*)

Post a new event.

Parameters

- **instance** – A Yamcs instance name.
- **message** – Event message.
- **event_type** – Type of event.
- **severity** – The severity level of the event. One of `info`, `watch`, `warning`, `distress`, `critical` or `severe`. Defaults to `info`.
- **time** – Time of the event. If unspecified, defaults to mission time.
- **source** – Source of the event. Useful for grouping events in the archive. When unset this defaults to `User`.
- **event_type** – Event type.
- **sequence_number** – Sequence number of this event. This is used to determine unicity of events at the same time and coming from the same source. If not set Yamcs will automatically assign a sequential number as if every submitted event is unique.
- **extra** – Extra event properties.

start_instance(*instance: str*)

Starts a single instance.

Parameters

instance – A Yamcs instance name.

start_service(*instance: str*, *service: str*)

Starts a single service.

Parameters

- **instance** – A Yamcs instance name.
- **service** – The name of the service.

stop_instance(*instance: str*)

Stops a single instance.

Parameters

instance – A Yamcs instance name.

stop_service(*instance: str*, *service: str*)

Stops a single service.

Parameters

- **instance** – A Yamcs instance name.
- **service** – The name of the service.

class yamcs.client.**TimeSubscription**(*manager*)

Bases: *WebSocketSubscriptionFuture*

Local object providing access to time updates.

A subscription object stores the last time info.

Initializes the future. Should not be called by clients.

time: *datetime* | None

The last time info.

class yamcs.client.**LinkSubscription**(*manager*)

Bases: *WebSocketSubscriptionFuture*

Local object providing access to data link updates.

A subscription object stores the last link info for each link.

Initializes the future. Should not be called by clients.

get_link(*name: str*) → *Link* | None

Returns the latest link state.

Parameters

name – Identifying name of the data link

list_links() → List[*Link*]

Returns a snapshot of all links.

1.1.2 Model

class yamcs.client.**AuthInfo**(*proto*)

Bases: object

Authentication information

property require_authentication: *bool*

class yamcs.client.**Event**(*proto*)

Bases: object

A timetagged free-text message. Events work a lot like log messages in logging frameworks, but then targeted at operators.

property event_type: *str* | None

The event type. This is mission-specific and can be any string.

property extra: *Dict*[*str*, *str*]

Dict with extra event properties.

property generation_time: *datetime*

The time when the event was generated.

property message: *str* | None

Event message.

property reception_time: *datetime*

The time when the event was received by Yamcs.

property sequence_number: *int*

Sequence number. Usually this is assigned by the source of the event.

property severity: *str* | None

Severity level of the event. One of INFO, WATCH, WARNING, DISTRESS, CRITICAL or SEVERE.

property source: *str* | None

The event source. Can be any string.

class yamcs.client.**Instance**(*proto*)

Bases: object

property failure_cause: str | None

Failure message when `state == 'FAILED'`

property mission_time: datetime | None

Mission time of this instance's time service.

property name: str

Name of this instance.

property state: str

State of this instance. One of `OFFLINE`, `INITIALIZING`, `INITIALIZED`, `STARTING`, `RUNNING`, `STOPPING` or `FAILED`.

class yamcs.client.**InstanceTemplate**(*proto*)

Bases: object

A template for creating an instance.

property name: str

Name of this template.

class yamcs.client.**Link**(*proto*)

Bases: object

Represents a link with an external system. Depending on the semantics of the link, this may imply inbound data, outbound data or a combination of both.

property actions: List[*LinkAction*]

Custom actions.

property class_name: str

Name of this link's class.

property enabled: bool

If `True`, this link accepts or outputs data.

property extra: Dict[str, Any]

Custom info fields.

property in_count: int

packet count).

Type

The number of inbound data events (example

property instance: str

Name of the instance where this link is defined.

property name: str

Name of this link (unique per instance).

property out_count: int

command count).

Type

The number of outbound data events (example

property status: str

Short status.

class yamcs.client.**LinkAction**(*proto*)

Bases: object

property checked: bool

Whether the action is currently checked

property enabled: bool

Whether the action is currently enabled

property id: str

Action ID

property label: str

Label for the action

property style: str

Action style

class yamcs.client.**LoadParameterValuesResult**(*proto*)

Bases: object

Statistics returned when loading a stream of parameter values.

property max_generation_time: datetime | None

Maximum geneneration time of all loaded parameter values

property min_generation_time: datetime | None

Minimum geneneration time of all loaded parameter values

property value_count: str

Number of loaded parameter values.

class yamcs.client.**ObjectPrivilege**(*proto*)

Bases: object

property name: str

property objects: List[str]

class yamcs.client.**Processor**(*proto*)

Bases: object

property instance: str

Name of the instance where this processor is defined.

property mission_time: datetime | None

Mission time of this processor.

property name: str

Name of this processor.

property owner: str

User that owns this processor.

property persistent: bool

If `True`, this processor does not close if no clients are connected.

property protected: bool

If `True`, this processor can not be deleted.

property state: str

State of this processor.

property type: str

Type of this processor.

class yamcs.client.**RdbTablespace**(*proto*)

Bases: object

property data_dir: str

Data directory

property name: str

Tablespace name

class yamcs.client.**ServerInfo**(*proto*)

Bases: object

General server properties.

property default_yamcs_instance: str | None

Returns the default Yamcs instance.

property id: str

The Server ID.

property version: str

The version of Yamcs Server.

class yamcs.client.**Service**(*proto*)

Bases: object

A Yamcs service.

property class_name: str

Name of this service's class.

failure_cause() → str | None

Java stacktrace when state is FAILED

Added in version 1.9.0: Compatible with Yamcs 5.8.0 onwards

failure_message() → str | None

Short failure message when state is FAILED

Added in version 1.9.0: Compatible with Yamcs 5.8.0 onwards

property instance: str | None

Name of the instance where this service is defined.

property name: str

Name of this service.

property processor: str | None

Name of the processor where this service is defined.

property state: str

State of this service.

class yamcs.client.**UserInfo**(*proto*)

Bases: object

Info on a Yamcs User.

property object_privileges: List[*ObjectPrivilege*]

property superuser: bool

property system_privileges: List[str]

property username: str

1.1.3 Authentication

1.1.3.1 User Accounts

Yamcs Server can be configured for different authentication setups.

The common use case is to entrust Yamcs with validating user credentials (either by locally verifying passwords, or by delegating to an upstream server such as an LDAP tree).

To authenticate in such a scenario, do:

```
credentials = Credentials(username="admin", password="password")
client = YamcsClient("localhost:8090", credentials=credentials)
```

In the background this will convert your username/password credentials to an access token with limited lifetime, and a long-lived refresh token for automatically generating new access tokens.

Further HTTP requests do not use your username/password but instead use these tokens.

1.1.3.2 Service Accounts

Service accounts are useful in server-to-server scenarios. Support for service accounts will be available in future releases.

1.1.3.3 Types

class yamcs.client.**APIKeyCredentials**(*key: str*)

Bases: *Credentials*

before_request(*session: Session*, *auth_url: str*)

is_expired() → bool

login(**args*, ***kwargs*)

refresh()

class yamcs.client.**BasicAuthCredentials**(*username: str*, *password: str*)

Bases: *Credentials*

Data holder for Basic Auth credentials. This includes a username and a password which are passed in the HTTP Authorization header on each request.

before_request(*session: Session*, *auth_url: str*)

is_expired() → bool

login(**args*, ***kwargs*)

refresh()

class yamcs.client.**Credentials**(*username: str | None = None*, *password: str | None = None*, *access_token: str | None = None*, *refresh_token: str | None = None*, *expiry: datetime | None = None*, *client_id: str | None = None*, *client_secret: str | None = None*, *become: str | None = None*)

Bases: `object`

Data holder for different types of credentials. Currently this includes:

- Username/password credentials (fields `username` and `password`)
- Bearer tokens (fields `access_token` and optionally `refresh_token`)

access_token

Short-lived bearer token.

become

Name of the user to impersonate. Only service accounts with impersonation authority can use this feature.

before_request(*session: Session*, *auth_url: str*)

client_id

The client ID. Currently used only by service accounts.

client_secret

The client secret. Currently used only by service accounts.

expiry

When this token expires.

is_expired() → bool

login(*session: Session*, *auth_url: str*, *on_token_update: Callable[[*Credentials*], None] | None*) → *Credentials*

password

Clear-text password (consider TLS!).

refresh(*session: Session*, *auth_url: str*)

refresh_token

Refresh token used to request a new access token.

username

Username (only needed when using username/password credentials).

1.1.4 Exceptions

class yamcs.client.**ConnectionFailure**

Bases: *YamcsError*

Yamcs is not or no longer available.

class yamcs.client.**NotFound**

Bases: *YamcsError*

The resource was not found.

class yamcs.client.**TimeoutError**

Bases: *YamcsError*

The operation exceeded the given deadline.

class yamcs.client.**Unauthorized**

Bases: *YamcsError*

Unable to get access the resource.

class yamcs.client.**YamcsError**

Bases: Exception

Base class for raised exceptions.

1.1.5 Futures

class yamcs.client.**WebSocketSubscriptionFuture**(*manager: WebSocketSubscriptionManager*)

Bases: Future

Future for capturing the asynchronous execution of a WebSocket subscription.

Initializes the future. Should not be called by clients.

add_done_callback(*fn*)

Attaches a callable that will be called when the future finishes.

Parameters

fn – A callable that will be called with this future as its only argument when the future completes or is cancelled. The callable will always be called by a thread in the same process in which it was added. If the future has already completed or been cancelled then the callable will be called immediately. These callables are called in the order that they were added.

cancel()

Closes the websocket and shutdowns the background thread consuming messages.

cancelled()

Return True if the future was cancelled.

done()

Return True if the future was cancelled or finished executing.

exception(*timeout=None*)

Return the exception raised by the call that the future represents.

Parameters

timeout – The number of seconds to wait for the exception if the future isn't done. If None, then there is no limit on the wait time.

Returns

The exception raised by the call that the future represents or None if the call completed without raising.

Raises

- **CancelledError** – If the future was cancelled.
- ***TimeoutError*** – If the future didn't finish executing before the given timeout.

reply(*timeout=None*)

Returns the initial reply. This is emitted before any subscription data is emitted. This function raises an exception if the subscription attempt failed.

result(*timeout=None*)

Return the result of the call that the future represents.

Parameters

timeout – The number of seconds to wait for the result if the future isn't done. If None, then there is no limit on the wait time.

Returns

The result of the call that the future represents.

Raises

- **CancelledError** – If the future was cancelled.
- ***TimeoutError*** – If the future didn't finish executing before the given timeout.
- **Exception** – If the call raised then that exception will be raised.

running()

Return True if the future is currently executing.

set_exception(*exception*)

Sets the result of the future as being the given exception.

Should only be used by Executor implementations and unit tests.

set_result(*result*)

Sets the return value of work associated with the future.

Should only be used by Executor implementations and unit tests.

1.2 Snippets

Create a *YamcsClient*:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
```

Provide credentials if Yamcs is secured:

```
from yamcs.client import Credentials, YamcsClient

credentials = Credentials(username='admin', password='password')
client = YamcsClient('localhost:8090', credentials=credentials)
```

1.2.1 Events

Receive *Event* callbacks:

```

def callback(event):
    print("Event:", event)

client.create_event_subscription(instance="simulator", on_data=callback)

sleep(5)  # Subscription is non-blocking
```

Send an event:

```
client.send_event(instance="simulator", message="hello world")
```

2. Mission Database

The Mission Database API provides methods for reading the entries in a Yamcs Mission Database (MDB). An MDB groups telemetry and command definitions for one or more *space systems*. The MDB is used to:

- Instruct Yamcs how to process incoming packets
- Describe items in Yamcs Archive
- Instruct Yamcs how to compose telecommands

Space systems form a hierarchical multi-rooted tree. Each level of the tree may contain any number of definitions. These break down in:

- parameters
- containers
- commands
- algorithms

Entries in the Space system are addressable via a qualified name that looks like a Unix file path. Each segment of the path contains the name of the space system node, the final path segment is the name of the entry itself.

For example, in an MDB that contains these parameter entries:

```
└── YSS
    └── SIMULATOR
        ├── BatteryVoltage1
        └── BatteryVoltage2
```

we find two space systems `/YSS` and `/YSS/SIMULATOR` and two parameter entries `/YSS/SIMULATOR/BatteryVoltage1` and `/YSS/SIMULATOR/BatteryVoltage2`.

Some MDB entries may also define an alias. An alias is a unique name to address the entry under a custom namespace (unrelated to XTCE space systems).

When it comes to addressing entries via this client, it is possible to provide either the fully-qualified XTCE name in the format `/YSS/SIMULATOR/BatteryVoltage1` or an alias in the format `NAMESPACE/NAME`.

2.1 Reference

2.1.1 Client

Note

`MDBClient` instances are usually created via *`YamcsClient.get_mdb()`*:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
mdb = client.get_mdb(instance='simulator')
# ...
```

class yamcs.client.**MDBClient**(*ctx: Context*, *instance: str*)

create_parameter(*name: str*, *data_source: str*, *short_description: str | None = None*, *long_description: str | None = None*, *aliases: Mapping[str, str] | None = None*, *parameter_type: str | None = None*) → *Parameter*

Create a parameter.

Added in version 1.9.1: Compatible with Yamcs 5.8.8 onwards

Parameters

- **name** – Fully qualified name of the parameter.

  Space systems that do not yet exist, will be added automatically.
- **data_source** – Where this parameter originated. One of `TELEMETERED`, `GROUND`, `DERIVED`, `CONSTANT`, `LOCAL`, `SYSTEM`, `COMMAND`, `COMMAND_HISTORY`.
- **short_description** – Short description (one line)
- **long_description** – Long description (Markdown)
- **aliases** – Aliases, keyed by namespace.
- **parameter_type** – Qualified name of a parameter type. This is optional, but recommended. It allows specifying units, alarms and so on.

create_parameter_type(*name: str*, *eng_type: str*, *short_description: str | None = None*, *long_description: str | None = None*, *aliases: Mapping[str, str] | None = None*, *unit: str | None = None*, *signed: bool | None = None*, *enum_values: Mapping[int, str] | None = None*, *one_string_value: str | None = None*, *zero_string_value: str | None = None*, *default_alarm:* RangeSet *| None = None*, *context_alarms: Mapping[str,* RangeSet*] | None = None*) → *ParameterType*

Create a parameter type.

Added in version 1.9.1: Compatible with Yamcs 5.8.8 onwards

Parameters

- **name** – Fully qualified name of the parameter type.

  Space systems that do not yet exist, will be added automatically.
- **eng_type** – Engineering type. One of `float`, `integer`, `boolean`, `binary`, `string` or `enumeration`.
- **short_description** – Short description (one line)
- **long_description** – Long description (Markdown)
- **aliases** – Aliases, keyed by namespace.
- **unit** – Engineering unit
- **signed** – Whether the engineering type supports signed representation. (only used with `integer` parameter types)
- **enum_values** – Enumeration labels, keyed by integer value (only used with `enumeration` parameter types)
- **one_string_value** – String representation of a boolean one. (only used with `boolean` parameter types)
- **zero_string_value** – String representation of a boolean zero. (only used with `boolean` parameter types)

- **default_alarm** – Default alarm, effective when no contextual alarm takes precedence.
- **context_alarms** – Contextual alarms, keyed by condition.

export_space_system(*name: str*, *, *version: str = '1.2'*) → str

Exports an XTCE description of a space system (XML format).

Added in version 1.9.0: Compatible with Yamcs 5.8.0 onwards

Parameters

- **name** – A fully-qualified XTCE name. Use `/` for root.
- **version** – XTCE version. One of `1.2` or `1.3`.

  Added in version 1.13.0: Compatible with Yamcs 5.12.6 onwards

get_algorithm(*name: str*) → *Algorithm*

Gets a single algorithm by its unique name.

Parameters
name – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.

get_command(*name: str*) → *Command*

Gets a single command by its unique name.

Parameters
name – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.

get_container(*name: str*) → *Container*

Gets a single container by its unique name.

Parameters
name – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.

get_parameter(*name: str*) → *Parameter*

Gets a single parameter by its name.

Parameters
name – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.

get_parameter_type(*name: str*) → *ParameterType*

Gets a single parameter type by its name.

Added in version 1.9.1: Compatible with Yamcs 5.8.8 onwards

Parameters
name – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.

get_space_system(*name: str*) → *SpaceSystem*

Gets a single space system by its unique name.

Parameters
name – A fully-qualified XTCE name. Use `/` for root.

list_algorithms(*page_size: int | None = None*) → Iterable[*Algorithm*]

Lists the algorithms visible to this client.

Algorithms are returned in lexicographical order.

list_commands(*page_size: int | None = None*) → Iterable[*Command*]

Lists the commands visible to this client.

Commands are returned in lexicographical order.

list_containers(*page_size: int | None = None*) → Iterable[*Container*]

Lists the containers visible to this client.

Containers are returned in lexicographical order.

list_parameter_types(*page_size: int | None = None*) → Iterable[*ParameterType*]

Lists the parameter types visible to this client.

Parameter types are returned in lexicographical order.

Added in version 1.9.1: Compatible with Yamcs 5.8.8 onwards

list_parameters(*parameter_type: str | None = None*, *page_size: int | None = None*) → Iterable[*Parameter*]

Lists the parameters visible to this client.

Parameters are returned in lexicographical order.

Parameters

parameter_type – The type of parameter

list_space_systems(*page_size: int | None = None*) → Iterable[*SpaceSystem*]

Lists the space systems visible to this client.

Space systems are returned in lexicographical order.

2.1.2 Model

class yamcs.client.**Algorithm**(*proto*)

Bases: *MissionDatabaseItem*

class yamcs.client.**Argument**(*proto*)

Bases: object

property description: str | None

Short description

property initial_value: str | None

Initial value

property name: str

Argument name

property type: *ArgumentType*

Argument type information

class yamcs.client.**ArgumentType**(*proto*)

Bases: object

property eng_type: str

Engineering type

property name: str

Type name

class yamcs.client.**ArrayType**(*proto*)

Bases: object

property array_type: *ArrayType* | None

In case the elements of an array of this type are also of type *array*, this returns type info of the elements' array type.

> **Note**
>
> This is an uncommon use case. Multi-dimensional arrays are more prevalent.

property dimensions: int | None

The number of dimensions in case of a multi-dimensional array.

property members: List[*Member*]

In case the elements of this array are of type *aggregate*, this returns an ordered list of its direct sub-members.

property name: str

Short name of this type.

class yamcs.client.**Command**(*proto*)

Bases: *MissionDatabaseItem*

property abstract: bool

Whether this is an abstract command. Abstract commands are intended for inheritance and cannot be issued directly.

property arguments: List[*Argument*]

property base_command: *Command* | None

property significance: *Significance* | None

class yamcs.client.**Container**(*proto*)

Bases: *MissionDatabaseItem*

class yamcs.client.**DataEncoding**(*proto*)

Bases: object

property bitlength: int | None

The size in bits

property encoding: str | None

Encoding detail

property little_endian: bool

True if little-endian

property type: str

Raw type

class yamcs.client.**EnumValue**(*proto*)

Bases: object

property description: str | None

State description

property label: str

String value

property value: int

Numeric value

class yamcs.client.**Member**(*proto*)

Bases: object

A member is a data structure for a specific field of a parent data type (either another member, or a parameter of type *aggregate*).

This is similar to C structs. The top-level of a member hierarchy is a parameter of type *aggregate*.

property array_type: *ArrayType* | None

In case this member is of type *array*, this returns array-specific type info.

property members: List[*Member*]

In case this member is of type *aggregate*, this returns an ordered list of its direct sub-members.

property name: str

Short name

property type: str

Engineering type.

class yamcs.client.**MissionDatabaseItem**(*proto*)

Bases: ABC

Superclass for MDB items. Implementations:

- *Algorithm*
- *Command*
- *Container*
- *Parameter*
- *ParameterType*
- *SpaceSystem*

property aliases: Dict[str, str]

Aliases, keyed by namespace

property aliases_dict: Dict[str, str]

Aliases, keyed by namespace

This method shall be deprecated in a future release. Use *aliases* instead.

property description: str | None

Short description.

property long_description: str | None

Long description.

property name: str

Short name

property qualified_name: str

Full name (incl. space system)

class yamcs.client.**Parameter**(*proto*)

Bases: *MissionDatabaseItem*

A Parameter is a description of something that can have a value. It is not the value itself.

property array_type: *ArrayType* | None

In case this parameter is of type *array*, this returns array-specific type info.

property data_encoding: *DataEncoding* | None

Information on the raw encoding of this parameter, if applicable.

property data_source: str | None

TELEMETERED)

Type

Specifies the source of this parameter (example

property enum_values: List[*EnumValue*]

In case this parameter is of type *enumeration*, this returns an ordered list of possible values.

property members: List[*Member*]

In case this parameter is of type *aggregate*, this returns an ordered list of its direct members.

property type: str | None

Engineering type.

property units: List[str]

Engineering unit(s)

class yamcs.client.**ParameterType**(*proto*)

Bases: *MissionDatabaseItem*

property array_type: *ArrayType* | None

In case this parameter type is of type *array*, this returns array-specific type info.

property data_encoding: *DataEncoding* | None

Information on the raw encoding of this parameter type, if applicable.

property enum_values: List[*EnumValue*]

In case this parameter type is of type *enumeration*, this returns an ordered list of possible values.

property members: List[*Member*]

In case this parameter type is of type *aggregate*, this returns an ordered list of its direct members.

property type: str

Engineering type.

property units: List[str]

Engineering unit(s)

class yamcs.client.**RangeSet**(*watch: Tuple[float | None, float | None] | None = None*, *warning: Tuple[float | None, float | None] | None = None*, *distress: Tuple[float | None, float | None] | None = None*, *critical: Tuple[float | None, float | None] | None = None*, *severe: Tuple[float | None, float | None] | None = None*, *min_violations: int = 1*)

Bases: object

A set of alarm ranges that apply in a specific context.

Parameters

- **watch** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **warning** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **distress** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **critical** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **severe** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **min_violations** – Minimum violations before an alarm is generated.

class yamcs.client.**Significance**(*proto*)

Bases: object

property consequence_level: *str*

One of `NONE`, `WATCH`, `WARNING`, `DISTRESS`, `CRITICAL` or `SEVERE`.

property reason: *str* | *None*

Message attached to this significance.

class yamcs.client.**SpaceSystem**(*proto*)

Bases: *MissionDatabaseItem*

From XTCE:

> A SpaceSystem is a collection of SpaceSystem(s) including space assets, ground assets, multi-satellite systems and sub-systems. A SpaceSystem is the root element for the set of data necessary to monitor and command an arbitrary space device - this includes the binary decomposition of the data streams going into and out of a device.

2.2 Snippets

Create an *MDBClient* for a specific instance:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
mdb = client.get_mdb(instance='simulator')
```

Print all space systems:

```
for space_system in mdb.list_space_systems():
    print(space_system)
```

Print all parameters of type `float`:

```
for parameter in mdb.list_parameters(parameter_type="float"):
    print(parameter)
```

Print all commands:

```
for command in mdb.list_commands():
    print(command)
```

Find a parameter by qualified name or alias:

```
p1 = mdb.get_parameter("/YSS/SIMULATOR/BatteryVoltage2")
print("Via qualified name:", p1)

p2 = mdb.get_parameter("MDB:OPS Name/SIMULATOR_BatteryVoltage2")
print("Via domain-specific alias:", p2)
```

3. TM/TC Processing

The TM/TC API provides methods that you can use to programmatically interact with a TM/TC processor.

3.1 Reference

3.1.1 Client

Note

`ProcessorClient` instances are usually created via *`YamcsClient.get_processor()`*:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
processor = client.get_processor(instance='simulator',
                                 processor='realtime')
# ...
```

class yamcs.client.**ProcessorClient**(*ctx: Context*, *instance: str*, *processor: str*)

Client object that groups operations linked to a specific processor.

acknowledge_alarm(*alarm: str*, *sequence_number: int*, *comment: str | None = None*)

Acknowledges a specific alarm.

Parameters

- **alarm** – Alarm name
- **sequence_number** – Sequence number
- **comment** – Optional comment to associate with the state change.

clear_alarm(*alarm: str*, *sequence_number: int*, *comment: str | None = None*)

Clear an alarm.

Note

If the reason that caused the alarm is still present, a new alarm instance will be generated.

Parameters

- **alarm** – Alarm name
- **sequence_number** – Sequence number
- **comment** – Optional comment to associate with the state change.

clear_alarm_ranges(*parameter: str*)

Removes all alarm limits for the specified parameter.

clear_calibrators(*parameter: str*)

Removes all calibrators for the specified parameter.

create_alarm_subscription(*on_data: Callable[[AlarmUpdate], None] | None = None*, *include_pending: bool = False*, *timeout: float = 60*) → *AlarmSubscription*

Create a new alarm subscription.

Parameters

- **on_data** – Function that gets called with *AlarmUpdate* updates.
- **timeout** – The amount of seconds to wait for the request to complete.
- **include_pending** – Whether to include alarms that are currently pending, because they have not reached the minimum violation count.

  Added in version 1.11.2: Compatible with Yamcs 5.11.0 onwards

Returns

A Future that can be used to manage the background websocket subscription.

create_command_connection(*on_data: Callable[[CommandHistory], None] | None = None*, *timeout: float = 60*) → *CommandConnection*

Creates a connection for issuing multiple commands and following up on their acknowledgment progress.

> **Note**
>
> This is a convenience method that merges the functionalities of *create_command_history_subscription()* with those of *issue_command()*.

Parameters

- **on_data** – Function that gets called with *CommandHistory* updates. Only commands issued from this connection are reported.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns

Future that can be used to manage the background websocket subscription

create_command_history_subscription(*on_data: Callable[[CommandHistory], None] | None = None*, *timeout: float = 60*) → *CommandHistorySubscription*

Create a new command history subscription.

Parameters

- **on_data** – Function that gets called with *CommandHistory* updates.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns

Future that can be used to manage the background websocket subscription.

create_container_subscription(*containers: str | List[str]*, *on_data: Callable[[ContainerData], None] | None = None*, *timeout: float = 60*) → *ContainerSubscription*

Create a new container subscription.

Parameters

- **containers** – Container names.
- **on_data** – Function that gets called with *ContainerData* updates.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns

A Future that can be used to manage the background websocket subscription.

create_packet_subscription(*on_data: Callable[[Packet], None]*, *stream: str | None = None*, *timeout: float = 60*) → *WebSocketSubscriptionFuture*

Create a new packet subscription.

Parameters

- **on_data** – Function that gets called with *Packet* updates.
- **stream** – Stream to subscribe to.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns

A Future that can be used to manage the background websocket subscription.

create_parameter_subscription(*parameters: str | List[str]*, *on_data: Callable[[ParameterData], None] | None = None*, *abort_on_invalid: bool = True*, *update_on_expiration: bool = False*, *send_from_cache: bool = True*, *timeout: float = 60*) → *ParameterSubscription*

Create a new parameter subscription.

Parameters

- **parameters** – Parameter names (or aliases).
- **on_data** – Function that gets called with *ParameterData* updates.
- **abort_on_invalid** – If `True` an error is generated when invalid parameters are specified.
- **update_on_expiration** – If `True` an update is received when a parameter value has become expired. This update holds the same value as the last known valid value, but with status set to `EXPIRED`.
- **send_from_cache** – If `True` the last processed parameter value is sent from parameter cache. When `False` only newly processed parameters are received.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns

A Future that can be used to manage the background websocket subscription.

get_parameter_value(*parameter: str*, *from_cache: bool = True*, *timeout: float = 10*) → *ParameterValue* | None

Retrieve the current value of the specified parameter.

Parameters

- **parameter** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **from_cache** – If `False` this call will block until a fresh value is received on the processor. If `True` the server returns the latest value instead (which may be `None`).
- **timeout** – The amount of seconds to wait for a fresh value. (ignored if `from_cache=True`).

get_parameter_values(*parameters: List[str]*, *from_cache: bool = True*, *timeout: float = 10*) → List[*ParameterValue* | None]

Retrieve the current value of the specified parameter.

Parameters

- **parameters** – List of parameter names. These may be fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **from_cache** – If `False` this call will block until fresh values are received on the processor. If `True` the server returns the latest value instead (which may be `None`).
- **timeout** – The amount of seconds to wait for a fresh values (ignored if `from_cache=True`).

Returns

A list that matches the length and order of the requested list of parameters. Each entry contains either the returned parameter value, or `None`.

issue_command(*command: str*, *args: Mapping[str, Any] | None = None*, *, *dry_run: bool = False*, *comment: str | None = None*, *verification: VerificationConfig | None = None*, *stream: str | None = None*, *extra: Mapping[str, Any] | None = None*, *sequence_number: int | None = None*) → *IssuedCommand*

Issue the given command

Parameters

- **command** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **args** – Named arguments (if the command requires any).
- **dry_run** – If `True` the command is not actually issued. This can be used to test if the server would generate errors when preparing the command (for example because an argument is missing).
- **comment** – Comment attached to the command.
- **verification** – Overrides to the default verification handling of this command.
- **stream** – Override the stream on which the command should be sent out.

  Added in version 1.9.6.

- **extra** – Extra command options for interpretation by non-core extensions (custom preprocessor, datalinks, command listeners). Note that Yamcs will refuse command options that it does now know about. Extensions should therefore register available options.
- **sequence_number** – Sequence number of this command. This is used to determine unicity of commands at the same time and coming from the same origin. If not set Yamcs will automatically assign a sequential number as if every submitted command is unique.

  Added in version 1.9.0.

Returns

An object providing access to properties of the newly issued command.

list_alarms(*start: datetime | None = None*, *stop: datetime | None = None*, *include_pending=False*) → Iterable[*Alarm*]

Lists the active alarms.

Remark that this does not query the archive. Only active alarms on the current processor are returned.

Parameters

- **start** – Minimum trigger time of the returned alarms (inclusive)
- **stop** – Maximum trigger time of the returned alarms (exclusive)
- **include_pending** – Whether to include alarms that are currently pending, because they have not reached the minimum violation count.

  Added in version 1.11.2: Compatible with Yamcs 5.11.0 onwards

reset_alarm_ranges(*parameter: str*)

Reset all alarm limits for the specified parameter to their original MDB value.

reset_algorithm(*parameter: str*)

Reset the algorithm text to its original definition from MDB

Parameters

parameter – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.

reset_calibrators(*parameter: str*)

Reset all calibrators for the specified parameter to their original MDB value.

run_script(*script: str*, *args: str | List[str] | None = None*)

Run a script.

The script has access to the environment variables `YAMCS_URL`, `YAMCS_API_KEY`, `YAMCS_INSTANCE` and `YAMCS_PROCESSOR`.

Parameters

- **script** – This should be the relative path to an an executable file in one of the search locations. When unconfigured, the default search location is `etc/scripts/` relative to the Yamcs working directory.
- **args** – Optional script arguments, passed verbatim in the command line.

set_alarm_range_sets(*parameter: str*, *sets: List[*AlarmRangeSet*]*)

Apply an ordered list of alarm range sets for the specified parameter. This replaces existing alarm sets (if any).

Each AlarmRangeSet may have a context, which indicates when its effects may be applied. Only the first matching set is applied.

An AlarmRangeSet with context `None` represents the *default* set of alarm ranges. There can be only one such set, and it is always applied at the end when no other set of contextual ranges is applicable.

Parameters

- **parameter** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **sets** – List of range sets (either contextual or not)

set_algorithm(*parameter: str*, *text: str*)

Change an algorithm text. Can only be peformed on JavaScript or Python algorithms.

Parameters

- **text** – New algorithm text (as it would appear in excel or XTCE)
- **parameter** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.

set_calibrators(*parameter: str*, *calibrators: List[*Calibrator*]*)

Apply an ordered set of calibrators for the specified parameter. This replaces existing calibrators (if any).

Each calibrator may have a context, which indicates when it its effects may be applied. Only the first matching calibrator is applied.

A calibrator with context `None` is the *default* calibrator. There can be only one such calibrator, and is always applied at the end when no other contextual calibrator was applicable.

Parameters

- **parameter** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **calibrators** – List of calibrators (either contextual or not)

set_default_alarm_ranges(*parameter: str*, *watch: Tuple[float | None, float | None] | None = None*, *warning: Tuple[float | None, float | None] | None = None*, *distress: Tuple[float | None, float | None] | None = None*, *critical: Tuple[float | None, float | None] | None = None*, *severe: Tuple[float | None, float | None] | None = None*, *min_violations: int = 1*)

Generate out-of-limit alarms for a parameter using the specified alarm ranges.

This replaces any previous default alarms on this parameter.

> **Note**
>
> Contextual range sets take precedence over the default alarm ranges. See *set_alarm_range_sets()* for setting contextual range sets.

Parameters

- **parameter** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **watch** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **warning** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **distress** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **critical** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **severe** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **min_violations** – Minimum violations before an alarm is generated.

set_default_calibrator(*parameter: str*, *type: str | None*, *data*)

Apply a calibrator while processing raw values of the specified parameter. If there is already a default calibrator associated to this parameter, that calibrator gets replaced.

> **Note**
>
> Contextual calibrators take precedence over the default calibrator See *set_calibrators()* for setting contextual calibrators.

Two types of calibrators can be applied:

- Polynomial calibrators apply a polynomial expression of the form: *y = a + bx + cx^2 + ...*.

  The *data* argument must be an array of floats `[a, b, c, ...]`.

- Spline calibrators interpolate the raw value between a set of points which represent a linear curve.

  The *data* argument must be an array of `[x, y]` points.

  Parameters

  - **parameter** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
  - **type** – One of `polynomial` or `spline`.
  - **data** – Calibration definition for the selected type.

set_parameter_value(*parameter: str*, *value: Any*, *generation_time: datetime | None = None*, *expires_in: float | None = None*)

Sets the value of the specified parameter.

Parameters

- **parameter** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **value** – The value to set.
- **generation_time** – Generation time of the value. If unset, Yamcs will assign the generation time.
- **expires_in** – How long before this value expires (in fractional seconds). If unset, the value does not expire.

  Added in version 1.9.1: Compatible with Yamcs 5.8.8 onwards

set_parameter_values(*values: Mapping[str, Any]*, *generation_time: datetime | None = None*, *expires_in: float | None = None*)

Sets the value of multiple parameters.

Values are specified with their native Python types. If you need to customize individual value properties, use *`ValueUpdate`* instead.

The method arguments `generation_time` and `expires_in` can be used to specify a custom generation time for all values at once. This has lower priority than value-specific properties.

If no generation time is specified at all, Yamcs will determine one.

Parameters

- **values** – Values keyed by parameter name. This name can be either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **generation_time** – Generation time of the values.
- **expires_in** – How long before this value expires (in fractional seconds).

  Added in version 1.9.1: Compatible with Yamcs 5.8.8 onwards

shelve_alarm(*alarm: str*, *sequence_number: int*, *comment: str | None = None*)

Shelve an alarm.

Parameters

- **alarm** – Alarm name
- **sequence_number** – Sequence number
- **comment** – Optional comment to associate with the state change.

unshelve_alarm(*alarm: str*, *sequence_number: int*, *comment: str | None = None*)

Unshelve an alarm.

Parameters

- **alarm** – Alarm name
- **sequence_number** – Sequence number
- **comment** – Optional comment to associate with the state change.

class yamcs.client.**CommandConnection**(*manager*, *tmtc_client: ProcessorClient*)

Bases: *WebSocketSubscriptionFuture*

Local object providing access to the acknowledgment progress of command updates.

Only commands issued from this object are monitored.

Initializes the future. Should not be called by clients.

issue(*command: str*, *args: Mapping[str, Any] | None = None*, ***, *dry_run: bool = False*, *comment: str | None = None*, *verification: VerificationConfig | None = None*, *stream: str | None = None*, *extra: Mapping[str, Any] | None = None*, *sequence_number: int | None = None*) → *MonitoredCommand*

Issue the given command

Parameters

- **command** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **args** – Named arguments (if the command requires these).
- **dry_run** – If `True` the command is not actually issued. This can be used to test if the server would generate errors when preparing the command (for example because an argument is missing).
- **comment** – Comment attached to the command.
- **verification** – Overrides to the default verification handling of this command.
- **stream** – Override the stream on which the command should be sent out.

  Added in version 1.9.6.
- **extra** – Extra command options for interpretation by non-core extensions (custom preprocessor, datalinks, command listeners). Note that Yamcs will refuse command options that it does now know about. Extensions should therefore register available options.
- **sequence_number** – Sequence number of this command. This is used to determine unicity of commands at the same time and coming from the same origin. If not set Yamcs will automatically assign a sequential number as if every submitted command is unique.

  Added in version 1.9.0.

Returns

An object providing access to properties of the newly issued command and updated according to command history updates.

class yamcs.client.**AlarmSubscription**(*manager*)

Bases: *WebSocketSubscriptionFuture*

Local object representing an alarm subscription.

A subscription object stores the currently active alarms.

Initializes the future. Should not be called by clients.

get_alarm(*name: str*) → *Alarm*

Returns the alarm state associated with a specific named alarm from local cache.

Parameters
name – Fully-qualified name

list_alarms() → List[*Alarm*]

Returns a snapshot of all active alarms.

class yamcs.client.**CommandHistorySubscription**(*manager: WebSocketSubscriptionManager*)

Bases: *WebSocketSubscriptionFuture*

Local object providing access to command history updates.

This object buffers all received command history. This is needed to stitch together incremental command history events.

If you expect to receive a lot of command history updates you should periodically clear local cache via `clear_cache()`. In future work, we may add automated buffer management within configurable watermarks.

Warning

If command history updates are received for commands that are not currently in the local cache, the returned information may be incomplete.

Initializes the future. Should not be called by clients.

clear_cache()

Clears local command history cache.

get_command_history(*issued_command:* IssuedCommand) → *CommandHistory* | None

Gets locally cached CommandHistory for the specified command.

Parameters
issued_command – object representing a previously issued command.

class yamcs.client.**ContainerSubscription**(*manager: WebSocketSubscriptionManager*)

Bases: *WebSocketSubscriptionFuture*

Local object providing access to container updates

Initializes the future. Should not be called by clients.

get_container(*name: str*) → *ContainerData*

Returns the latest container of a specific name.

Parameters
name – Container name

list_containers() → List[*ContainerData*]

Returns the latest container for each name.

class yamcs.client.**ParameterSubscription**(*manager: WebSocketSubscriptionManager*)

Bases: *WebSocketSubscriptionFuture*

Local object representing a subscription of zero or more parameters.

A subscription object stores the last received value of each subscribed parameter.

Initializes the future. Should not be called by clients.

add(*parameters: str | List[str]*, *abort_on_invalid: bool = True*, *send_from_cache: bool = True*)

Add one or more parameters to this subscription.

Parameters

- **parameters** – Parameter(s) to be added
- **abort_on_invalid** – If `True` one invalid parameter means any other parameter in the request will also not be added to the subscription.
- **send_from_cache** – If `True` the last processed parameter value is sent from parameter cache. When `False` only newly processed parameters are received.

await_first_delivery(*timeout: float | None = None*)

Wait for the first update of parameter values.

Parameters

timeout – The amount of seconds to wait.

delivery_count: `int`

The number of parameter deliveries.

get_value(*parameter: str*)

Returns the last value of a specific parameter from local cache.

Parameters

parameter – Parameter name.

remove(*parameters: str | List[str]*)

Remove one or more parameters from this subscription.

Parameters

parameters – Parameter(s) to be removed

value_cache: `Dict[str, ParameterValue]`

Value cache keyed by parameter name.

3.1.2 Model

class yamcs.client.**Acknowledgment**(*name*, *time*, *status*, *message*)

Bases: `object`

is_terminated()

message

Message of this acknowledgment.

name

Name of this acknowledgment.

status

Status of this acknowlegment.

time

Last update time of this acknowledgment.

class yamcs.client.**Alarm**(*proto*)

Bases: `ABC`

property acknowledge_message: `str | None`

Comment provided when acknowledging the alarm.

property acknowledge_time: `datetime | None`

Processor time when the alarm was acknowledged.

property acknowledged_by: str | None

Username of the acknowledger.

property count: int

Total number of samples while this alarm is active.

property is_acknowledged: bool

True if this alarm has been acknowledged.

property is_latched: bool

True if this alarm is currently latched.

property is_latching: bool

True if this is a latching alarm. A latching alarm returns to normal only when the operator resets it

property is_ok: bool

True if this alarm is currently 'inactive'.

For a non-latching alarm this is identical to *is_process_ok()*. A latching alarm is only OK if it has returned to normal and was acknowledged.

property is_process_ok: bool

True if the process that caused this alarm is OK. For example: parameter back within limits.

For a non-latching alarm this is identical to *is_ok()*.

property is_shelved: bool

True if this alarm has been shelved.

property name: str

Fully-qualified name of the source of this alarm.

property sequence_number: int

Sequence number for this specific alarm instance. This allows ensuring that operations (such as acknowledgment) are done on the expected alarm instance.

property severity: str

property trigger_time: datetime

Processor time when the alarm was triggered.

property update_time: datetime

Processor time when the alarm was last updated.

property violation_count: int

Number of violating samples while this alarm is active.

class yamcs.client.**AlarmRangeSet**(*context: str*, *watch: Tuple[float | None, float | None] | None = None*, *warning: Tuple[float | None, float | None] | None = None*, *distress: Tuple[float | None, float | None] | None = None*, *critical: Tuple[float | None, float | None] | None = None*, *severe: Tuple[float | None, float | None] | None = None*, *min_violations: int = 1*)

Bases: object

A set of alarm ranges that apply in a specific context.

Parameters

- **context** – Condition under which this range set is applicable. The value None indicates the default range set which is only applicable if no contextual sets match.
- **watch** – Range expressed as a tuple (lo, hi) where lo and hi are assumed exclusive.

- **warning** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **distress** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **critical** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **severe** – Range expressed as a tuple `(lo, hi)` where lo and hi are assumed exclusive.
- **min_violations** – Minimum violations before an alarm is generated.

class yamcs.client.**AlarmUpdate**(*proto*)

Bases: object

Object received through callbacks when subscribing to alarm updates.

property alarm: *Alarm*

Latest alarm state.

property update_type: str

Type of update.

class yamcs.client.**Calibrator**(*context: str*, *type: str*, *data*)

Bases: object

A calibrator that may be applied to a numeric raw value.

Two types of calibrators can be applied:

- **Polynomial calibrators apply a polynomial expression of the form:**
  y = a + bx + cx^2 + ...

  The *data* argument must be an array of floats `[a, b, c, ...]`.
- **Spline calibrators interpolate the raw value between a set of points**
  which represent a linear curve.

  The *data* argument must be an array of `[x, y]` points.

Parameters

- **context** – Condition under which this calibrator may be applied. The value `None` indicates the default calibrator which is only applied if no contextual calibrators match.
- **type** – One of `polynomial` or `spline`.
- **data** – Calibration definition for the selected type.

POLYNOMIAL = 'polynomial'

SPLINE = 'spline'

class yamcs.client.**CommandHistory**(*proto*)

Bases: object

property acknowledgments: Dict[str, *Acknowledgment*]

All acknowledgments by name.

Returns
Acknowledgments keyed by name.

property aliases: Dict[str, str]

Aliases, keyed by namespace

Added in version 1.9.2: Compatible with Yamcs 5.8.4 onwards

property all_assignments: Dict[str, Any]

Argument assignments by name.

Added in version 1.9.2.

property assignments: Dict[str, Any]

Argument assignments by name.

This returns only explicit assignments set by the command issuer. To return all assignments, including ones that use their default values, use *all_assignments* instead.

Added in version 1.9.2.

property binary

Binary representation of the command.

property comment: str | None

Optional user comment attached when issuing the command.

property error: str | None

Error message in case the command failed.

generation_time: datetime

The generation time as set by Yamcs

property id: str

A unique identifier for this command.

is_complete() → bool

Returns whether this command is complete. A command can be completed, yet still failed.

is_failure() → bool

Returns True if the command has completed, but failed.

is_success() → bool

Returns True if the command has completed successfully.

property name: str

Name of the command.

property origin: str | None

The origin of this command. Usually the IP address of the issuer.

property sequence_number: int

The sequence number of this command. This is the sequence number assigned by the issuing client.

property source: str

String representation of the command.

property unprocessed_binary

Binary representation before postprocessing.

Added in version 1.9.2.

property username: str | None

Username of the issuer.

class yamcs.client.**ContainerData**(*proto*)

Bases: object

property binary

Raw binary

property generation_time: *datetime*

The time when this container's packet was generated (packet time).

property name: *str*

The name of the container.

property reception_time: *datetime*

The time when this container's packet was received by Yamcs.

class yamcs.client.**EventAlarm**(*proto*)

Bases: *Alarm*

An alarm triggered by an event.

property current_event: *Event*

Latest event for this alarm

property most_severe_event: *Event*

First event that invoked the highest severity level of this alarm

property trigger_event: *Event*

Event that originally triggered the alarm

class yamcs.client.**IssuedCommand**(*proto*)

Bases: object

property aliases: Dict[str, str]

Aliases, keyed by namespace

Added in version 1.9.2: Compatible with Yamcs 5.8.4 onwards

property all_assignments: Dict[str, Any]

Argument assignments by name.

Added in version 1.9.2.

property assignments: Dict[str, Any]

Argument assignments by name.

This returns only explicit assignments set by the command issuer. To return all assignments, including ones that use their default values, use *all_assignments* instead.

Added in version 1.9.2.

property binary

Binary representation of this command.

property generation_time: *datetime*

The generation time as set by Yamcs.

property hex

Hexadecimal string representation of this command.

property id: *str*

A unique identifier for this command.

property name: *str*

The fully-qualified name of this command.

property origin: *str* | None

The origin of this command. Usually the IP address of the issuer.

property queue: *str* | None

The name of the queue that this command was assigned to.

property sequence_number

The sequence number of this command. This is the sequence number assigned by the issuing client.

property source

String representation of this command.

property unprocessed_binary

Binary representation before postprocessing.

property username: str

The username of the issuer.

class yamcs.client.**MonitoredCommand**(*proto*)

Bases: *IssuedCommand*

Represent an instance of an issued command that is updated throughout the acknowledgment process.

Objects of this class are owned by a *CommandConnection* instance.

property acknowledgments: Dict[str, *Acknowledgment*]

All acknowledgments by name.

Returns

Acknowledgments keyed by name.

property attributes: Dict[str, Any]

await_acknowledgment(*name: str*, *timeout: float | None = None*) → *Acknowledgment*

Waits for the result of a specific acknowledgment.

Parameters

- **name** – The name of the acknowledgment. Standard names are `Acknowledge_Queued`, `Acknowledge_Released` and `Acknowledge_Sent`. Others depend on specific link types.
- **timeout** – The amount of seconds to wait.

await_complete(*timeout: float | None = None*)

Wait for the command to be *completed*. Afterwards use *is_success()* or *is_failure()* to determine whether the command was successful or not.

> **Note**
>
> Yamcs cannot determine completion unless the command has an appropriate verifier configured in the Yamcs MDB.
>
> When no such verifier is present, this method will never return (or timeout).

Parameters

timeout – The amount of seconds to wait.

property comment: str | None

Optional user comment attached when issuing the command.

property error: str | None

Error message in case the command failed.

This information is only available when the command has failed to complete. (*is_failure()* returns *True*).

is_complete() → bool

Returns whether this command is complete. A command can be completed, yet still failed.

Use *await_complete()* to wait until this information is available.

is_failure() → bool

Returns True if the command has completed, but failed.

Use *await_complete()* to wait until this information is available.

is_success() → bool

Returns True if this command was completed successfully.

Use *await_complete()* to wait until this information is available.

class yamcs.client.**Packet**(*proto*)

Bases: object

property binary: bytes

Raw binary of this packet

property earth_reception_time: datetime | None

When the signal was received on the ground.

Added in version 1.9.1.

property generation_time: datetime

The time when the packet was generated (packet time).

property link: str

Name of the Yamcs link where this packet was received from.

Added in version 1.9.1.

property name: str

The name of the packet. When using XTCE extraction this is the fully-qualified name of the first container in the hierarchy that this packet maps to.

property reception_time: datetime

The time when the packet was received by Yamcs.

property sequence_number: int

The sequence number of the packet. This is usually decoded from the packet.

property size: int

Size in bytes of this packet

class yamcs.client.**ParameterAlarm**(*proto*)

Bases: *Alarm*

An alarm triggered by a parameter that went out of limits.

property current_value: *ParameterValue*

Latest parameter value for this alarm.

property most_severe_value: *ParameterValue*

First parameter value that invoked the highest severity level of this alarm.

property trigger_value: *ParameterValue*

Parameter value that originally triggered the alarm

class yamcs.client.**ParameterData**(*proto*, *mapping=None*)

Bases: object

get_value(*parameter: str*) → *ParameterValue* | None

Returns the value of a specific parameter. Or `None` if the parameter is not included in this update.

Parameters

parameter – Parameter name.

property parameters: List[*ParameterValue*]

class yamcs.client.**ParameterValue**(*proto*, *id=None*)

Bases: object

property eng_value

The engineering (calibrated) value.

property generation_time: datetime

The time when the parameter was generated. If the parameter was extracted from a packet, this usually returns the packet time.

property monitoring_result: str

property name: str

An identifying name for the parameter value. Typically this is the fully-qualified XTCE name, but it may also be an alias depending on how the parameter update was requested.

property range_condition: str | None

If the value is out of limits, this indicates `LOW` or `HIGH`.

property raw_value

The raw (uncalibrated) value.

property reception_time: datetime | None

The time when the parameter value was received by Yamcs.

property validity_duration: timedelta | None

How long this parameter value is valid.

property validity_status: str

class yamcs.client.**ValueUpdate**(*value: Any*, *generation_time: datetime | None = None*, *expires_in: float | None = None*)

Bases: object

Data holder for passing a value along with its generation time when updating a software parameter.

Parameters

- **value** – The value to set
- **generation_time** – Generation time of the value. If unset, Yamcs will assign the generation time.
- **expires_in** – How long before this value expires (in fractional seconds). If unset, the value does not expire.

  Added in version 1.9.1: Compatible with Yamcs 5.8.8 onwards

class yamcs.client.**VerificationConfig**

Bases: object

Contains overrides to the default verification handling of Yamcs.

disable(*verifier: str | None = None*)

Disable verification.

Parameters

verifier – Name of a specific verifier to disable. If unspecified all verifiers are disabled.

modify_check_window(*verifier: str*, *start: float | None = None*, *stop: float | None = None*)

Set or override the check window.

Depending on the Mission Database configuration, the time may be relative to either the command release or a preceding verifier.

Parameters

- **verifier** – Name of the verifier
- **start** – Window start time (relative, in seconds)
- **stop** – Window stop time (relative, in seconds)

3.2 Snippets

Create a *ProcessorClient* for a specific processor:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
processor = client.get_processor(instance='simulator', processor='realtime')
```

3.2.1 Read/Write Parameters

Read a single value. This returns the latest value from server cache.

```
pval = processor.get_parameter_value("/YSS/SIMULATOR/BatteryVoltage1")
print(pval)
```

Read a single value, but block until a fresh value could be processed:

```
pval = processor.get_parameter_value(
    "/YSS/SIMULATOR/BatteryVoltage2", from_cache=False, timeout=5
)
print(pval)
```

Read the latest value of multiple parameters at the same time:

```
pvals = processor.get_parameter_values(
    ["/YSS/SIMULATOR/BatteryVoltage1", "/YSS/SIMULATOR/BatteryVoltage2"]
)
print("battery1", pvals[0])
print("battery2", pvals[1])
```

Set the value of a parameter. Only some types of parameters can be written to. This includes software parameters (local to Yamcs) and parameters that are linked to an external system (such as a simulator).

```
processor.set_parameter_value("/YSS/SIMULATOR/AllowCriticalTC1", True)
```

Set the value of multiple parameters:

```
processor.set_parameter_values(
    {
        "/YSS/SIMULATOR/AllowCriticalTC1": False,
        "/YSS/SIMULATOR/AllowCriticalTC2": False,
    }
)
```

3.2.2 Parameter Subscription

Poll latest values from a subscription:

```
subscription = processor.create_parameter_subscription(
    ["/YSS/SIMULATOR/BatteryVoltage1"]
)

sleep(5)
print("Latest value:")
print(subscription.get_value("/YSS/SIMULATOR/BatteryVoltage1"))

sleep(5)
print("Latest value:")
print(subscription.get_value("/YSS/SIMULATOR/BatteryVoltage1"))
```

Receive *ParameterData* callbacks whenever one or more of the subscribed parameters have been updated:

```
def print_data(data):
    for parameter in data.parameters:
        print(parameter)

processor.create_parameter_subscription(
    "/YSS/SIMULATOR/BatteryVoltage1", on_data=print_data
)
sleep(5)  # Subscription is non-blocking
```

Create and modify a parameter subscription:

```
subscription = processor.create_parameter_subscription(
    ["/YSS/SIMULATOR/BatteryVoltage1"]
)

sleep(5)

print("Adding extra items to the existing subscription...")
subscription.add(
    [
        "/YSS/SIMULATOR/Alpha",
        "/YSS/SIMULATOR/BatteryVoltage2",
        "MDB:OPS Name/SIMULATOR_PrimBusVoltage1",
    ]
)

sleep(5)

print("Shrinking subscription...")
subscription.remove("/YSS/SIMULATOR/Alpha")

print("Cancelling the subscription...")
subscription.cancel()

print("Last values from cache:")
print(subscription.get_value("/YSS/SIMULATOR/BatteryVoltage1"))
print(subscription.get_value("/YSS/SIMULATOR/BatteryVoltage2"))
print(subscription.get_value("/YSS/SIMULATOR/Alpha"))
print(subscription.get_value("MDB:OPS Name/SIMULATOR_PrimBusVoltage1"))
```

3.2.3 Commanding

Issue a command (fire-and-forget):

```
command = processor.issue_command(
    "/YSS/SIMULATOR/SWITCH_VOLTAGE_OFF", args={"voltage_num": 1}
)
print("Issued", command)
```

To monitor acknowledgments, establish a command connection first. Commands issued from this connection are automatically updated with progress status:

```
conn = processor.create_command_connection()

command = conn.issue("/YSS/SIMULATOR/SWITCH_VOLTAGE_OFF", args={"voltage_num": 1})

ack = command.await_acknowledgment("Acknowledge_Sent")
print(ack.status)
```

The default Yamcs-local acknowledgments are:

- Acknowledge_Queued
- Acknowledge_Released
- Acknowledge_Sent

Custom telemetry verifiers or command links may cause additional acknowledgments to be generated.

If configured, command completion can also be monitored:

```
conn = processor.create_command_connection()

command1 = conn.issue("/YSS/SIMULATOR/SWITCH_VOLTAGE_OFF", args={"voltage_num": 1})

# Issue 2nd command only if the previous command was completed successfully.
command1.await_complete()
if command1.is_success():
    conn.issue("/YSS/SIMULATOR/SWITCH_VOLTAGE_ON", args={"voltage_num": 1})
else:
    print("Command 1 failed:", command1.error)
```

3.2.4 Alarm Monitoring

Receive *AlarmUpdate* callbacks:

```
def callback(alarm_update):
    print("Alarm Update:", alarm_update)

processor.create_alarm_subscription(callback)
```

Acknowledge all active alarms:

```
for alarm in processor.list_alarms():
    if not alarm.is_acknowledged:
        processor.acknowledge_alarm(
            alarm.name,
            alarm.sequence_number,
            comment="false alarm",
        )
```

3.2.5 Run a Script

Run a custom script. The script has to be known by Yamcs. By default, this is done by placing an executable file in the `etc/scripts/` directory of Yamcs.

Scripts have access to special environment variables `YAMCS_URL`, `YAMCS_API_KEY`, `YAMCS_INSTANCE` and `YAMCS_PROCESSOR`, which is used by *YamcsClient.from_environment()*

```
# Simulate LOS for 5 seconds
# (the run_script call does not block)
processor.run_script("simulate_los.py", "--duration 5")
```

etc/scripts/simulate_los.sh

```
#!/usr/bin/env -S python3 -u

import argparse
import os
import time

from yamcs.client import YamcsClient

parser = argparse.ArgumentParser()
parser.add_argument("-d", "--duration", type=float, default=60)
args = parser.parse_args()

client = YamcsClient.from_environment()
processor = client.get_processor(
    instance=os.environ["YAMCS_INSTANCE"],
    processor=os.environ["YAMCS_PROCESSOR"],
)

print("Starting LOS")
processor.issue_command("/TSE/simulator/start_los")
time.sleep(args.duration)

print("Stopping LOS")
processor.issue_command("/TSE/simulator/stop_los")
```

Notice the use of `-u` in the shebang. This disables Python output buffering, so that print statements are immediately seen.

4. Archive

The Archive API provides methods that you can use to programmatically retrieve the content of a Yamcs Archive.

4.1 Reference

4.1.1 Client

Note

`ArchiveClient` instances are usually created via *`YamcsClient.get_archive()`*:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
archive = client.get_archive(instance='simulator')
# ...
```

class yamcs.client.**ArchiveClient**(*ctx: Context*, *instance: str*)

create_stream_subscription(*stream: str*, *on_data: Callable[[StreamData], None]*, *timeout: float = 60*) → *WebSocketSubscriptionFuture*

Create a new stream subscription.

Parameters

- **stream** – The name of the stream.
- **on_data** – Function that gets called with *`StreamData`* updates.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns

Future that can be used to manage the background websocket subscription.

disable_parameter_archive_backfilling()

Disables the automatic backfilling (rebuilding) of the parameter archive.

If the backfilling is already disabled, this operation has no effect.

downsample_mean(*parameter: str*, *start: datetime | None = None*, *stop: datetime | None = None*, ***, *sample_count: int = 500*, *parameter_cache: str = 'realtime'*, *source: str = 'ParameterArchive'*) → List[*MeanSample*]

Downsample parameter values in the requested range using averaging (mean).

The query range is split in sample intervals of equal length. For each interval a *`MeanSample`* is returned which describes the min, max, count and avg during that interval.

Note that sample times are determined without considering the actual parameter values. Two separate queries with equal start/stop arguments will always return the same number of samples

with the same timestamps. This is done to ease merging of multiple sample series. You should always be explicit about the `start` and `stop` times when relying on this property.

Parameters

- **parameter** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **start** – Minimum generation time of the sampled parameter values (inclusive). If not set this defaults to one hour ago.
- **stop** – Maximum generation time of the sampled parameter values (exclusive). If not set this defaults to the current time.
- **sample_count** – The number of returned samples.
- **parameter_cache** – Specify the name of the processor who's parameter cache is merged with already archived values. To disable results from the parameter cache, set this to `None`.
- **source** – Specify how to retrieve parameter values. By default this uses the `ParameterArchive` which is optimized for retrieval. For Yamcs instances that do not enable the `ParameterArchive`, you can still get results by specifying `replay` as the source. Replay requests take longer to return because the data needs to be reprocessed.

dump_table(*table: str*, *query: str | None = None*, *chunk_size: int = 32768*)

enable_parameter_archive_backfilling()

Enables the automatic backfilling (rebuilding) of the parameter archive.

If the backfilling is already enabled, this operation has no effect.

execute_sql(*statement: str*) → *ResultSet*

Executes a single SQL statement.

Parameters
statement – SQL string

Returns
A result set for consuming rows

export_commands(*start: datetime | None = None*, *stop: datetime | None = None*, *chunk_size: int = 32768*) → Iterable

Export command history.

Commands are sorted by generation time, origin and sequence number.

Parameters

- **start** – Minimum generation time of the returned commands (inclusive)
- **stop** – Maximum generation time of the returned commands (exclusive)

Returns
An iterator over received chunks

export_packets(*name: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *chunk_size: int = 32768*) → Iterable

Export raw packets.

Packets are sorted by generation time and sequence number.

Parameters

- **name** – Archived name of the packet
- **start** – Minimum generation time of the returned packets (inclusive)

- **stop** – Maximum generation time of the returned packets (exclusive)

Returns
An iterator over received chunks

export_parameter_values(*parameters: List[str]*, ***, *namespace: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *interval: float | None = None*, *chunk_size: int = 32768*, *source: str = 'ParameterArchive'*) → Iterable

Export parameter values in CSV format.

Parameters

- **parameters** – List of parameter names. These may be fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **namespace** – Preferred namespace of the names in the returned CSV header
- **start** – Minimum generation time of the returned values (inclusive)
- **stop** – Maximum generation time of the returned values (exclusive)
- **interval** – If specified, only one value for each interval is returned. The interval is expressed in seconds.
- **source** – Specify how to retrieve parameter values. By default this uses the `ParameterArchive` which is optimized for parameter retrieval. For Yamcs instances that do not enable the `ParameterArchive`, you can still get results by specifying `replay` as the source. Replay requests take longer to return because the packet data needs to be reprocessed.

  Added in version 1.13.0: Compatible with Yamcs v5.12.6 onwards

Returns
An iterator over received chunks

get_packet(*generation_time: datetime*, *sequence_number: int*, *partition: str | None = None*) → *Packet*

Gets a single packet by its identifying key (partition, gentime, seqNum).

Parameters

- **generation_time** – When the packet was generated (packet time)
- **sequence_number** – Sequence number of the packet
- **partition** – Packet partition name. This property works only against recent versions of Yamcs,and will become required in the future (it was erroneously omitted).

get_stream(*stream: str*) → *Stream*

Gets a single stream.

Parameters
stream – The name of the stream.

get_table(*table: str*) → *Table*

Gets a single table.

Parameters
table – The name of the table.

list_alarms(*name: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *page_size: int = 500*, *descending: bool = False*) → Iterable[*Alarm*]

Reads alarm information between the specified start and stop time.

Alarms are sorted by trigger time, name and sequence number.

Parameters

- **name** – Filter by alarm name
- **start** – Minimum trigger time (inclusive)
- **stop** – Maximum trigger time (exclusive)
- **page_size** – Page size of underlying requests. Higher values imply less overhead, but risk hitting the maximum message size limit.
- **descending** – If set to `True` alarms are fetched in reverse order (most recent first).

list_command_histogram(*name: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *merge_time: float = 2*) → Iterable[*IndexGroup*]

Reads command-related index records between the specified start and stop time.

Each iteration returns a chunk of chronologically-sorted records.

Parameters

merge_time – Maximum gap in seconds before two consecutive index records are merged together.

list_command_history(*command: str | None = None*, *queue: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *page_size: int = 500*, *descending: bool = False*) → Iterable[*CommandHistory*]

Reads command history entries between the specified start and stop time.

> **Note**
>
> This method will send out multiple requests when more than `page_size` commands are queried. For large queries, consider using *`stream_command_history()`* instead, it uses server-streaming based on a single request.

Parameters

- **command** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **queue** – Name of the queue that the command was assigned to.
- **start** – Minimum generation time of the returned command history entries (inclusive)
- **stop** – Maximum generation time of the returned command history entries (exclusive)
- **page_size** – Page size of underlying requests. Higher values imply less overhead, but risk hitting the maximum message size limit.
- **descending** – If set to `True` results are fetched in reverse order (most recent first).

list_completeness_index(*start: datetime | None = None*, *stop: datetime | None = None*, *merge_time: float = 2*) → Iterable[*IndexGroup*]

Reads completeness index records between the specified start and stop time.

Each iteration returns a chunk of chronologically-sorted records.

Parameters

merge_time – Maximum gap in seconds before two consecutive index records are merged together.

list_event_histogram(*source: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *merge_time: float = 2*) → Iterable[*IndexGroup*]

Reads event-related index records between the specified start and stop time.

Each iteration returns a chunk of chronologically-sorted records.

Parameters

merge_time – Maximum gap in seconds before two consecutive index records are merged together.

list_event_sources() → Iterable[str]

Returns the existing event sources.

list_events(*source: str | None = None*, *severity: str | None = None*, *text_filter: str | None = None*, *filter: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *page_size: int = 500*, *descending: bool = False*) → Iterable[*Event*]

Reads events between the specified start and stop time.

Events are sorted by generation time, source, then sequence number.

Note

This method will send out multiple requests when more than `page_size` events are queried. For large queries, consider using *`stream_events()`* instead, it uses server-streaming based on a single request.

Parameters

- **source** – The source of the returned events.
- **severity** – The minimum severity level of the returned events. One of `INFO`, `WATCH`, `WARNING`, `DISTRESS`, `CRITICAL` or `SEVERE`.
- **text_filter** – Filter the text message of the returned events
- **filter** – Filter query, allows for both text and field search.

  Added in version 1.10.1: Compatible with Yamcs v5.10.2 onwards
- **start** – Minimum start date of the returned events (inclusive)
- **stop** – Maximum start date of the returned events (exclusive)
- **page_size** – Page size of underlying requests. Higher values imply less overhead, but risk hitting the maximum message size limit.
- **descending** – If set to `True` events are fetched in reverse order (most recent first).

list_packet_histogram(*name: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *merge_time: float = 2*) → Iterable[*IndexGroup*]

Reads packet-related index records between the specified start and stop time.

Each iteration returns a chunk of chronologically-sorted records.

Parameters

merge_time – Maximum gap in seconds before two consecutive index records are merged together.

list_packet_names() → Iterable[str]

Returns the existing packet names.

list_packets(*name: str | None = None*, *filter: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *page_size: int = 500*, *descending: bool = False*) → Iterable[*Packet*]

Reads packet information between the specified start and stop time.

Packets are sorted by generation time and sequence number.

Note

This method will send out multiple requests when more than `page_size` packets are queried. For large queries, consider using *`stream_packets()`* instead, it uses server-streaming based on a single request.

Parameters

- **`name`** – Archived name of the packet
- **`filter`** – Filter query, allows for both text and field search.

  Added in version 1.12.0: Compatible with Yamcs v5.12.0 onwards
- **`start`** – Minimum generation time of the returned packets (inclusive)
- **`stop`** – Maximum generation time of the returned packets (exclusive)
- **`page_size`** – Page size of underlying requests. Higher values imply less overhead, but risk hitting the maximum message size limit.
- **`descending`** – If set to `True` packets are fetched in reverse order (most recent first).

`list_parameter_ranges`(*parameter: str*, *start: datetime | None = None*, *stop: datetime | None = None*, *min_gap: float | None = None*, *max_gap: float | None = None*, *min_range: float | None = None*, *max_values: int = 100*, *parameter_cache: str = 'realtime'*) → List[*ParameterRange*]

Returns parameter ranges between the specified start and stop time.

Each range indicates an interval during which this parameter's value was uninterrupted and unchanged.

Ranges are a good fit for retrieving the value of a parameter that does not change frequently. For example an on/off indicator or some operational status. Querying ranges will then induce much less overhead than manually processing the output of *`list_parameter_values()`* would.

The maximum number of returned ranges is limited to 500.

Parameters

- **`parameter`** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **`start`** – Minimum generation time of the considered values (inclusive)
- **`stop`** – Maximum generation time of the considered values (exclusive)
- **`min_gap`** – Time in seconds. Any gap (detected based on parameter expiration) smaller than this will be ignored. However if the parameter changes value, the ranges will still be split.
- **`max_gap`** – Time in seconds. If the distance between two subsequent parameter values is bigger than this value (but smaller than the parameter expiration), then an artificial gap is created. This also applies if there is no expiration defined for the parameter.
- **`min_range`** – Time in seconds. Minimum duration of returned ranges. If multiple values occur within the range, the most frequent can be accessed using the `entries` property.
- **`max_values`** – Maximum number of unique values, tallied across the full requested range. Use this in combination with `min_range` to further optimize for transfer size. This value is limited to 100 at most.

- **parameter_cache** – Specify the name of the processor who's parameter cache is merged with already archived values. To disable results from the parameter cache, set this to `None`.

list_parameter_values(*parameter: str*, *start: datetime | None = None*, *stop: datetime | None = None*, *page_size: int = 500*, *descending: bool = False*, *parameter_cache: str = 'realtime'*, *source: str = 'ParameterArchive'*) → Iterable[*ParameterValue*]

Reads parameter values between the specified start and stop time.

> **Note**
>
> This method will send out multiple requests when more than `page_size` values are queried. For large queries, consider using *`stream_parameter_values()`* instead, it uses server-streaming based on a single request, and supports downloading the values of multiple parameter at the same time.

Parameters

- **parameter** – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **start** – Minimum generation time of the returned values (inclusive)
- **stop** – Maximum generation time of the returned values (exclusive)
- **page_size** – Page size of underlying requests. Higher values imply less overhead, but risk hitting the maximum message size limit.
- **descending** – If set to `True` values are fetched in reverse order (most recent first).
- **parameter_cache** – Specify the name of the processor who's parameter cache is merged with already archived values. To disable results from the parameter cache, set this to `None`.
- **source** – Specify how to retrieve parameter values. By default this uses the `ParameterArchive` which is optimized for retrieval. For Yamcs instances that do not enable the `ParameterArchive`, you can still get results by specifying `replay` as the source. Replay requests take longer to return because the data needs to be reprocessed.

list_processed_parameter_group_histogram(*group: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *merge_time: float = 20*) → Iterable[*IndexGroup*]

Reads index records related to processed parameter groups between the specified start and stop time.

Each iteration returns a chunk of chronologically-sorted records.

Parameters

merge_time – Maximum gap in seconds before two consecutive index records are merged together.

list_processed_parameter_groups() → Iterable[str]

Returns the existing parameter groups.

list_streams() → Iterable[*Stream*]

Returns the existing streams.

Streams are returned in lexicographical order.

list_tables() → Iterable[*Table*]

Returns the existing tables.

Tables are returned in lexicographical order.

load_table(*table: str*, *data*, *chunk_size: int = 32768*) → int

purge_parameter_archive()

Removes all Parameter Archive data and related metadata.

Afterwards, use *rebuild_parameter_archive()* to rebuild the Parameter Archive.

rebuild_ccsds_index(*start: datetime | None = None*, *stop: datetime | None = None*)

Rebuilds the CCSDS index. This is only applicable to projects that use a CcsdsTmIndex service to calculate packet completeness.

> **Note**
>
> Index rebuilds run synchronously: this method will await the outcome.

Parameters

- **start** – Start time
- **stop** – Stop time

rebuild_histogram(*table: str*, *start: datetime | None = None*, *stop: datetime | None = None*)

Rebuilds the histogram for a table. This may be necessary for example after bulk loading data.

The rebuild may be constrained by using the `start` and `stop` parameters. When specified all partitions overlapping this range are reconsidered.

> **Note**
>
> Histogram rebuilds run synchronously: this method will await the outcome.

Parameters

- **table** – The name of the table
- **start** – Start time
- **stop** – Stop time

rebuild_parameter_archive(*start: datetime | None = None*, *stop: datetime | None = None*)

Rebuilds the Parameter Archive.

The rebuild may be constrained by using the `start` and `stop` parameters. These values are only hints to the Parameter Archive, which will extend the requested range based on archive segmentation.

> **Note**
>
> Rebuilds run as an asynchronous operation: this method will not await the outcome.

Parameters

- **start** – Start time. This argument is optional since Yamcs v5.11.4
- **stop** – Stop time. This argument is optional since Yamcs v5.11.4

sample_parameter_values(*parameter: str*, *start: datetime | None = None*, *stop: datetime | None = None*, *sample_count: int = 500*, *parameter_cache: str = 'realtime'*, *source: str = 'ParameterArchive'*) → List[*MeanSample*]

> **Warning**
>
> Deprecated. Since yamcs-client 1.12.0 this method was renamed to `downsample_mean()`.

stream_command_history(*name: str | List[str] | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *chunk_size: int = 32768*) → Iterable[*CommandHistory*]

Reads command history between the specified start and stop time.

Added in version 1.9.1.

Parameters

- **name** – If specified, return only commands with this name.
- **start** – Minimum generation time of the returned commands (inclusive)
- **stop** – Maximum generation time of the returned commands (exclusive)

stream_events(*source: str | List[str] | None = None*, *severity: str | None = None*, *text_filter: str | None = None*, *filter: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *chunk_size: int = 32768*) → Iterable[*Event*]

Reads events between the specified start and stop time.

Added in version 1.9.1.

Parameters

- **source** – If specified, return only events with this source.
- **severity** – The minimum severity level of the returned events. One of `INFO`, `WATCH`, `WARNING`, `DISTRESS`, `CRITICAL` or `SEVERE`.
- **text_filter** – Filter the text message of the returned events
- **filter** – Filter query applied to returned events. Allows for both text and field search.

  Added in version 1.10.1: Compatible with Yamcs v5.10.2 onwards
- **start** – Minimum generation time of the returned events (inclusive)
- **stop** – Maximum generation time of the returned events (exclusive)

stream_packets(*name: str | List[str] | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *chunk_size: int = 32768*) → Iterable[*Packet*]

Reads packet information between the specified start and stop time.

Added in version 1.9.1.

Parameters

- **name** – If specified, return only packets with this archived name.
- **start** – Minimum generation time of the returned packets (inclusive)
- **stop** – Maximum generation time of the returned packets (exclusive)

stream_parameter_values(*parameters: str | List[str]*, *start: datetime | None = None*, *stop: datetime | None = None*, *tm_links: str | List[str] | None = None*, *chunk_size: int = 32768*) → Iterable[*ParameterData*]

Reads parameter values between the specified start and stop time.

Value updates are emitted for each unique generation time within the queried range. If one of the parameters does not have a value for a specific generation time, it is not included in the update.

Parameters

- **parameters** – Parameter(s) to be queried.
- **start** – Minimum generation time of the returned values (inclusive)
- **stop** – Maximum generation time of the returned values (exclusive)
- **tm_links** – If set, include only values that were received through a specific link.

4.1.2 Model

class yamcs.client.**ColumnData**(*proto*)

Bases: object

property **name**: str

Column name.

property **value**: Any

Value for this column.

class yamcs.client.**IndexGroup**(*proto*)

Bases: object

Group of index records that represent the same type of underlying objects.

property **name**: str

Name associated with this group. The meaning is defined by the objects represented by this index. For example:

- In an index of events, index records are grouped by `source`.
- In an index of packets, index records are grouped by `packet name`.

property **records**: List[*IndexRecord*]

Index records within this group

class yamcs.client.**IndexRecord**(*proto*)

Bases: object

Represents a block of uninterrupted data (derived from the index definition for the type of underlying objects, in combination with the requested `merge_time`.

property **count**: int

Number of underlying objects this index record represents

property **start**: datetime

Start time of the record

property **stop**: datetime

Stop time of the record

class yamcs.client.**MeanSample**(*proto*)

Bases: object

Provides aggregation properties over a range of a parameter's values.

Added in version 1.12.0: In earlier versions this class was called `Sample`.

property avg: float | None

Average (mean) value.

property max: float | None

Maximum value.

property min: float | None

Minimum value.

property parameter_count: int

The number of parameter values this sample represents.

property time: datetime

Sample time.

class yamcs.client.**ParameterRange**(*proto*)

Bases: object

Indicates an interval during which a parameter's value was uninterrupted and unchanged.

property eng_value: Any | None

The engineering (calibrated) value within this range.

If the request was made using `min_range` option, this will be the most-frequent value only. Retrieve the complete distribution using the `entries` attribute.

property entries: List[*ParameterRangeEntry*]

Value distribution within this range.

Unless the request was made using `min_range` option, there should be only one entry only.

property parameter_count: int

The total number of parameter values within this range.

property start: datetime

Start time of this range (inclusive).

property stop: datetime

Stop time of this range (exclusive).

class yamcs.client.**ParameterRangeEntry**(*proto*)

Bases: object

Value holder for an engineering value and its number of appearances within a ParameterRange.

property eng_value: Any

The engineering (calibrated) value.

property parameter_count: int

The number of received parameter values during this range.

class yamcs.client.**ResultSet**(*response*)

Bases: object

Provides capability to consume the rows returned by a SQL query, or access related information.

property column_types: List[str] | None

Column types.

property columns: List[str] | None

Column names. This returns `None` as long as no row has been consumed yet.

class yamcs.client.**Stream**(*proto*)

Bases: object

property name: str

Stream name.

class yamcs.client.StreamData(*proto*)

Bases: object

property columns: List[*ColumnData*]

Tuple columns.

property stream: str

Stream name.

class yamcs.client.Table(*proto*)

Bases: object

property name: str

Table name.

4.2 Snippets

Create an *ArchiveClient* for a specific instance:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
archive = client.get_archive(instance='simulator')
```

4.2.1 Packet Retrieval

Print the last 10 packets:

```
for packet in islice(archive.list_packets(descending=True), 0, 10):
    print(packet)
```

Print available range of archived packets:

```
first_packet = next(iter(archive.list_packets()))
last_packet = next(iter(archive.list_packets(descending=True)))
print("First packet:", first_packet)
print("Last packet:", last_packet)

td = last_packet.generation_time - first_packet.generation_time
print("Timespan:", td)
```

Iterate a specific range of packets:

```
now = datetime.now(tz=timezone.utc)
start = now - timedelta(hours=1)

total = 0
for packet in archive.list_packets(start=start, stop=now):
    total += 1
    # print(packet)
print("Found", total, "packets in range")
```

Download raw packet binary to a file:

```
now = datetime.now(tz=timezone.utc)
start = now - timedelta(hours=1)

with open("/tmp/dump.raw", "wb") as f:
    for chunk in archive.export_packets(start=start, stop=now):
        f.write(chunk)
```

4.2.2 Parameter Retrieval

Retrieve the last 10 values of a parameter:

```
iterable = archive.list_parameter_values(
    "/YSS/SIMULATOR/BatteryVoltage1", descending=True
)
for pval in islice(iterable, 0, 10):
    print(pval)
```

Iterate a specific range of values:

```
now = datetime.now(tz=timezone.utc)
start = now - timedelta(hours=1)

total = 0
for pval in archive.list_parameter_values(
    "/YSS/SIMULATOR/BatteryVoltage1", start=start, stop=now
):
    total += 1
    # print(pval)
print("Found", total, "parameter values in range")
```

Iterate values of multiple parameters by unique generation time:

```
now = datetime.now(tz=timezone.utc)
start = now - timedelta(hours=1)

total = 0
for pdata in archive.stream_parameter_values(
    ["/YSS/SIMULATOR/BatteryVoltage1", "/YSS/SIMULATOR/BatteryVoltage2"],
    start=start,
    stop=now,
):
    total += 1
    # print(pdata)
print("Found", total, "updates in range")
```

4.2.3 Event Retrieval

Iterate a specific range of events:

```
now = datetime.now(tz=timezone.utc)
start = now - timedelta(hours=1)

total = 0
for event in archive.list_events(start=start, stop=now):
    total += 1
    # print(event)
print("Found", total, "events in range")
```

4.2.4 Command Retrieval

Retrieve the last 10 issued commands:

```
iterable = archive.list_command_history(descending=True)
for entry in islice(iterable, 0, 10):
    print(entry)
```

4.2.5 Histogram Retrieval

Print the number of packets grouped by packet name:

```
for name in archive.list_packet_names():
    packet_count = 0
    for group in archive.list_packet_histogram(name):
```

(continues on next page)

(continued from previous page)

```
        for rec in group.records:
            packet_count += rec.count
    print(f"  {name: <40} {packet_count: >20}")
```

Print the number of events grouped by source:

```
for source in archive.list_event_sources():
    event_count = 0
    for group in archive.list_event_histogram(source):
        for rec in group.records:
            event_count += rec.count
    print(f"  {source: <40} {event_count: >20}")
```

Print the number of processed parameter frames grouped by group name:

```
for group in archive.list_processed_parameter_groups():
    frame_count = 0
    for pp_group in archive.list_processed_parameter_group_histogram(group):
        for rec in pp_group.records:
            frame_count += rec.count
    print(f"  {group: <40} {frame_count: >20}")
```

Print the number of commands grouped by name:

```
mdb = client.get_mdb(instance="simulator")
for command in mdb.list_commands():
    total = 0
    for group in archive.list_command_histogram(command.qualified_name):
        for rec in group.records:
            total += rec.count
    print(f"  {command.qualified_name: <40} {total: >20}")
```

5. Link Management

The Link API provides methods that you can use to programmatically interact with a Yamcs link.

5.1 Reference

5.1.1 Client

Note

`LinkClient` instances are usually created via *YamcsClient.get_link()*:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
link = client.get_link(instance='simulator', link='udp-in')
# ...
```

class yamcs.client.**LinkClient**(*ctx: Context*, *instance: str*, *link: str*)

Client object that groups operations for a specific link.

create_cop1_subscription(*on_data: Callable[[Cop1Status], None]*, *timeout: float = 60*) → *Cop1Subscription*

Create a new subscription for receiving status of the COP1 link.

This method returns a future, then returns immediately. Stop the subscription by canceling the future.

Parameters

- **on_data** – Function that gets called on each *Cop1Status*.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns

Future that can be used to manage the background websocket subscription.

disable_cop1(*bypass_all: bool = True*)

Disable COP1 for a data link.

Parameters

bypass_all – All frames have bypass activated (i.e. they will be BD frames)

disable_link()

Disables this link.

enable_link()

Enables this link.

get_cop1_config() → *Cop1Config*

Gets the COP1 configuration for a data link.

get_cop1_status() → *Cop1Status*

Retrieve the COP1 status.

get_info() → *Link*

Get info on this link.

initialize_cop1(*type: str*, *clcw_wait_timeout: int | None = None*, *v_r: int | None = None*)

Initialize COP1.

Parameters

- **type** – One of `WITH_CLCW_CHECK`, `WITHOUT_CLCW_CHECK`, `UNLOCK` or `SET_VR`
- **clcw_wait_timeout** – timeout in seconds used for the reception of CLCW. Required if type is `WITH_CLCW_CHECK`
- **v_r** – value of v(R) if type is set to `SET_VR`

resume_cop1()

Resume COP1.

run_action(*action: str*, *message: Mapping[str, Any] | None = None*) → Dict[str, Any]

Runs the given action for this link

Parameters

- **action** – action identifier
- **message** – action message

Returns

Action result (if the action returns anything)

sdls_get_ctr(*spi: int*) → int

Get the sequence counter associated with a given *spi* (Security Parameter Index) on this link.

sdls_set_ctr(*spi: int*, *new_seq: int*)

Set the sequence counter associated with a given *spi* (Security Parameter Index) on this link to *new_seq*.

sdls_set_key(*spi: int*, *key: bytes*)

Update the *key* associated with a given *spi* (Security Parameter Index) on this link.

sdls_set_spi(*vc_id: int*, *spi: int*)

Update the *spi* used for SDLS on the VC *vc_id* on this link.

sdls_set_spis(*vc_id: int*, *spis: List[int]*)

Update the *spis* used for SDLS on the VC *vc_id* on this link. Only valid for incoming links (i.e., downlink).

update_cop1_config(*window_width: int | None = None*, *timeout_type: str | None = None*, *tx_limit: int | None = None*, *t1: float | None = None*) → *Cop1Config*

Sets the COP1 configuration for a data link.

class yamcs.client.**Cop1Subscription**(*manager*)

Bases: *WebSocketSubscriptionFuture*

Local object providing access to COP1 status updates.

Initializes the future. Should not be called by clients.

property bypass_all: bool | None

property cop1_active: bool | None

get_status() → *Cop1Status* | None

Returns the latest known COP1 status.

property nn_r: int | None

property state: str | None

property v_s: int | None

5.1.2 Model

class yamcs.client.**Cop1Config**(*proto*)

Bases: object

COP1 configuration

property t1: float

property timeout_type: str

property tx_limit: int

property vc_id: int

Virtual Channel ID.

property window_width: int

class yamcs.client.**Cop1Status**(*proto*)

Bases: object

COP1 status

property bypass_all: bool | None

property cop1_active: bool

property nn_r: int | None

property state: str | None

property v_s: int | None

5.2 Snippets

Create a *LinkClient* for a specific link:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
link = client.get_link(instance='simulator', link='udp-in')
```

Enable a link:

```
link.enable_link()
```

Run a link action:

```
link.run_action(action_id, message)
```

6. Object Storage

The Storage API provides methods that you can use to programmatically work with Yamcs buckets and objects.

6.1 Reference

6.1.1 Client

class yamcs.client.**StorageClient**(*ctx: Context*)

Client for working with buckets and objects managed by Yamcs.

create_bucket(*bucket_name: str*)

Create a new bucket.

Parameters

bucket_name – The name of the bucket.

download_object(*bucket_name: str*, *object_name: str*) → bytes

Download an object.

Parameters

- **bucket_name** – The name of the bucket.
- **object_name** – The object to fetch.

get_bucket(*name: str*, *create=False*) → *Bucket*

Get a specific bucket.

Parameters

- **name** – The bucket name.
- **create** – If specified, create the bucket if it does not yet exist.

list_buckets() → Iterable[*Bucket*]

List the buckets.

list_objects(*bucket_name: str*, *prefix: str | None = None*, *delimiter: str | None = None*) → *ObjectListing*

List the objects for a bucket.

Parameters

- **bucket_name** – The name of the bucket.
- **prefix** – If specified, only objects that start with this prefix are listed.
- **delimiter** – If specified, return only objects whose name do not contain the delimiter after the prefix. For the other objects, the response contains (in the prefix response parameter) the name truncated after the delimiter. Duplicates are omitted.

remove_bucket(*bucket_name: str*)

Remove a bucket.

Parameters

bucket_name – The name of the bucket.

remove_object(*bucket_name: str*, *object_name: str*)

Remove an object from a bucket.

Parameters

- **bucket_name** – The name of the bucket.
- **object_name** – The object to remove.

upload_object(*bucket_name: str*, *object_name: str*, *file_obj: str | IO*, *content_type: str | None = None*, *metadata: Mapping[str, str] | None = None*)

Upload an object to a bucket.

Parameters

- **bucket_name** – The name of the bucket.
- **object_name** – The target name of the object.
- **file_obj** – The file (or file-like object) to upload.
- **content_type** – The content type associated to this object. This is mainly useful when accessing an object directly via a web browser. If unspecified, a content type *may* be automatically derived from the specified `file_obj`.
- **metadata** – Optional metadata associated to this object.

6.1.2 Model

class yamcs.client.**Bucket**(*proto*, *storage_client: StorageClient*)

Bases: `object`

property created: datetime

When this bucket was created.

delete()

Remove this bucket in its entirety.

delete_object(*object_name: str*)

Remove an object from this bucket.

Parameters

object_name – The object to remove.

property directory: str | None

Bucket root directory. This field is only set when the bucket is mapped to the file system. Therefore it is not set for buckets that store objects in RocksDB.

download_object(*object_name: str*) → bytes

Download an object.

Parameters

object_name – The object to fetch.

list_objects(*prefix: str | None = None*, *delimiter: str | None = None*)

List the objects for this bucket.

Parameters

- **prefix** – If specified, only objects that start with this prefix are listed.

- **delimiter** – If specified, return only objects whose name do not contain the delimiter after the prefix. For the other objects, the response contains (in the prefix response parameter) the name truncated after the delimiter. Duplicates are omitted.

property max_object_count: int | None

Maximum allowed number of objects.

property max_size: int | None

Maximum allowed total size of all objects.

property name: str

Name of this bucket.

property object_count: int

Number of objects in this bucket.

property size: int

Total size in bytes of this bucket (excluding metadata).

upload_object(*object_name: str*, *file_obj: str | IO*, *content_type: str | None = None*, *metadata: Mapping[str, str] | None = None*)

Upload an object to this bucket.

Parameters

- **object_name** – The target name of the object.
- **file_obj** – The file (or file-like object) to upload.
- **content_type** – The content type associated to this object. This is mainly useful when accessing an object directly via a web browser. If unspecified, a content type *may* be automatically derived from the specified `file_obj`.
- **metadata** – Optional metadata associated to this object.

class yamcs.client.**ObjectInfo**(*proto*, *bucket*, *storage_client*)

Bases: object

property created: datetime

Return when this object was created (or re-created).

delete()

Remove this object.

download()

Download this object.

property name: str

The name of this object.

property size: int

Size in bytes of this object (excluding metadata).

upload(*file_obj: str | IO*, *metadata: Mapping[str, str] | None = None*)

Replace the content of this object.

Parameters

- **file_obj** – The file (or file-like object) to upload.
- **metadata** – Optional metadata associated to this object.

class yamcs.client.**ObjectListing**(*proto*, *bucket*, *storage_client*)

Bases: object

property objects: List[*ObjectInfo*]

The objects in this listing.

property prefixes: List[str]

The prefixes in this listing.

6.2 Snippets

Create a *StorageClient*:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
storage = client.get_storage_client()
```

7. File Transfer

The File Transfer API provides methods that you can use to programmatically work with file transfers such as CFDP.

7.1 Reference

7.1.1 Client

class yamcs.client.**FileTransferClient**(*ctx: Context*, *instance: str*)

Client for working with file transfers (e.g. CFDP) managed by Yamcs.

get_service(*name: str*) → *FileTransferService*

Get a specific File Transfer service.

Parameters
name – The service name.

list_services() → Iterable[*FileTransferService*]

List the services.

class yamcs.client.**FileTransferServiceClient**(*ctx: Context*, *proto*)

cancel_transfer(*id: str*) → None

create_filelist_subscription(*on_data: Callable[[RemoteFileListing], None] | None = None*, *timeout: float = 60*) → *FileListSubscription*

create_transfer_subscription(*on_data: Callable[[Transfer], None] | None = None*, *timeout: float = 60*) → *TransferSubscription*

download(*bucket_name: str*, *remote_path: str*, *object_name: str | None*, *source_entity: str | None*, *destination_entity: str | None*, *options: Mapping[str, Any] | None*) → *Transfer*

fetch_filelist(*remote_path: str*, *source_entity: str | None*, *destination_entity: str | None*, *options: Mapping[str, Any] | None*)

get_filelist(*remote_path: str*, *source_entity: str | None*, *destination_entity: str | None*, *options: Mapping[str, Any] | None*) → *RemoteFileListing*

pause_transfer(*id: str*) → None

resume_transfer(*id: str*) → None

run_file_action(*file: str*, *action: str*, *message: Mapping[str, Any] | None = None*) → Dict[str, Any]

upload(*bucket_name: str*, *object_name: str*, *remote_path: str*, *source_entity: str | None*, *destination_entity: str | None*, *options: Mapping[str, Any] | None*) → *Transfer*

class yamcs.client.**FileListSubscription**(*manager*)

Local object providing access to filelist updates.

A subscription object stores the last filelist info for each remotepath and destination.

Initializes the future. Should not be called by clients.

get_filelist(*remote_path: str*, *destination: str*) → *RemoteFileListing* | None

Get the latest cached filelist for the given remote path and destination

Parameters

- **remote_path** – path on the remote destination
- **destination** – remote entity name

class yamcs.client.**TransferSubscription**(*manager*, *service_client: FileTransferServiceClient*)

Local object providing access to transfer updates.

A subscription object stores the last transfer info for each transfer.

Initializes the future. Should not be called by clients.

get_transfer(*id: str*) → *Transfer* | None

Returns the latest transfer state.

Parameters

id – Transfer identifier

list_completed() → List[*Transfer*]

Returns all completed transfers (successful or not).

list_ongoing() → List[*Transfer*]

Returns all ongoing transfers.

list_transfers() → List[*Transfer*]

Returns a snapshot of all transfer info.

7.1.2 Model

class yamcs.client.**EntityInfo**(*proto*)

Bases: object

property id: str

Entity ID

property name: str

Name of the entity

class yamcs.client.**FileTransferCapabilities**(*proto*)

Bases: object

property download: bool

property filelist: bool

property has_transfer_type: bool

property pause_resume: bool

property remote_path: bool

property upload: bool

class yamcs.client.**FileTransferOption**(*proto*)

Bases: object

property allow_custom_option: bool

Whether using different values from the pre-set ones is allowed

property associated_text: str

Text associated with the option

property default: Any

Default value for the option

property description: str

Description for the option

property name: str

Name of the option

property type: str

Type of the option

property values

List of possible values for the option

class yamcs.client.**RemoteFile**(*proto*)

Bases: object

Represents a file on a remote entity.

property display_name: str | None

Optionally, a preferred displayed name of the file.

property is_directory: bool

Whether the file is a directory

property modified: datetime | None

Latest modification time of the file

property name: str

Identifying name of the file

property size: int

File size in bytes

class yamcs.client.**RemoteFileListing**(*proto*)

Bases: object

Represents a list of files from a remote.

property destination: str

Remote destination of the file list

property files: List[*RemoteFile*]

List of files

property list_time: datetime

Time the file list was made

property remote_path: str

Remote directory of the file list

class yamcs.client.**FileTransferService**(*proto*, *service_client:* FileTransferServiceClient)

Bases: object

cancel_transfer(*id: str*) → None

Cancel a transfer

property capabilities: *FileTransferCapabilities*

Transfer capabilities

create_filelist_subscription(*on_data: Callable[[*RemoteFileListing*], None] | None = None*, *timeout: float = 60*) → *FileListSubscription*

Create a new filelist subscription.

Parameters

- **on_data** – Function that gets called with *RemoteFileListing* updates.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns

Future that can be used to manage the background websocket subscription

Return type

yamcs.client.filetransfer.client.FileListSubscription

create_transfer_subscription(*on_data: Callable[[*Transfer*], None] | None = None*, *timeout: float = 60*) → *TransferSubscription*

Create a new transfer subscription.

Parameters

- **on_data** – Function that gets called with *Transfer* updates.
- **timeout** – The amount of seconds to wait for the request to complete.

Returns

Future that can be used to manage the background websocket subscription

Return type

yamcs.client.filetransfer.client.TransferSubscription

download(*bucket_name: str*, *remote_path: str*, *object_name: str | None = None*, *source_entity: str | None = None*, *destination_entity: str | None = None*, *options: Mapping[str, Any] | None = None*) → *Transfer*

Downloads a file from the source to a bucket.

Parameters

- **bucket_name** – Name of the bucket to receive the file.
- **object_name** – Name of the file received in the bucket.
- **remote_path** – Name of the file to be downloaded from the source.
- **source_entity** – Use a specific source entity. (useful in case of multiples)
- **destination_entity** – Use a specific destination entity. (useful in case of multiples)
- **options** – File transfer options.

fetch_filelist(*remote_path: str*, *source_entity: str | None = None*, *destination_entity: str | None = None*, *options: Mapping[str, Any] | None = None*)

Sends a request to fetch the directory listing from the remote (destination).

Parameters

- **remote_path** – path on the remote destination to get the file list
- **source_entity** – source entity requesting the file list
- **destination_entity** – destination entity from which the file list is needed

- **options** – option dictionary

get_filelist(*remote_path: str*, *source_entity: str | None = None*, *destination_entity: str | None = None*, *options: Mapping[str, Any] | None = None*) → *RemoteFileListing*

Returns the latest directory listing for the given destination.

Parameters

- **remote_path** – path on the remote destination to get the file list
- **source_entity** – source entity requesting the file list
- **destination_entity** – destination entity from which the file list is needed
- **options** – option dictionary

property instance: str

Instance of the service

property local_entities: List[*EntityInfo*]

List of local entities

property name: str

Name of this service.

pause_transfer(*id: str*) → None

Pauses a transfer

property remote_entities: List[*EntityInfo*]

List of remote entities

resume_transfer(*id: str*) → None

Resume a transfer

run_file_action(*file: str*, *action: str*, *message: Mapping[str, Any] | None = None*) → Dict[str, Any]

Run an action on a remote file.

Available actions depend on the specific file transfer implementation that is in use use.

Added in version 1.9.6.

Parameters

- **file** – Remote file identifier
- **action** – Action identifier
- **message** – Action message

Returns

Action result (if the action returns anything)

property transfer_options: List[*FileTransferOption*]

List of possible transfer options

upload(*bucket_name: str*, *object_name: str*, *remote_path: str*, *source_entity: str | None = None*, *destination_entity: str | None = None*, *options: Mapping[str, Any] | None = None*) → *Transfer*

Uploads a file located in a bucket to a remote destination path.

Parameters

- **bucket_name** – Name of the bucket containing the source object.
- **object_name** – Name of the source object.
- **remote_path** – Remote destination.
- **source_entity** – Use a specific source entity. (useful in case of multiples)

- **destination_entity** – Use a specific destination entity. (useful in case of multiples)
- **options** – file transfer options

class yamcs.client.**Transfer**(*proto*, *service_client*)

Bases: object

Represents a file transfer.

await_complete(*timeout: float | None = None*)

Wait for the transfer to be completed.

Parameters
timeout – The amount of seconds to wait.

property bucket: str

cancel()

Cancel this transfer

property error: str | None

Error message in case the transfer failed.

property id: str

Yamcs-local transfer identifier.

is_complete() → bool

Returns whether this transfer is complete. A transfer can be completed, yet still failed.

is_success() → bool

Returns true if this transfer was completed successfully.

property object_name: str

pause()

Pause this transfer

property reliable: bool

True if this is a Class 2 CFDP transfer.

property remote_path: str

resume()

Resume this transfer

property size: int

Total bytes to transfer.

property state: str | None

Current transfer state.

property time: datetime | None

Time when the transfer was started.

property transferred_size: int

Total bytes already transferred.

7.2 Snippets

Create a *FileTransferClient* for a specific instance:

```python
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
cfdp = client.get_file_transfer_client(instance='cfdp')

# Operations are grouped by service.
# Here: take the first available
service = next(cfdp.list_services())
```

Upload a file to the specified location on the remote entity:

```python
# Transfer myfile from bucket to spacecraft
upload = service.upload(out_bucket.name, "myfile", "/CF:/mytarget")
upload.await_complete(timeout=10)

if not upload.is_success():
    print("Upload failure:", upload.error)
else:
    print(f"Successfully uploaded {upload.remote_path} ({upload.size} bytes)")
```

Start file download:

```python
# Download todownload from the remote
download = service.download(in_bucket.name, "todownload")
```

Initiate transfer with non-default parameters:

```python
# Transfer myfile, but use an alternative destination entity
upload = service.upload(
    out_bucket.name, "myfile", "/CF:/mytarget", destination_entity="target2"
)
```

Initiate transfer with extra transfer options:

```python
upload = service.upload(
    out_bucket.name, "myfile", "/CF:/mytarget", options={"reliable": True}
)
```

Subscribe to file list changes:

```python
subscription = service.create_filelist_subscription(on_data=filelist_callback)
```

Fetch file list from remote directory:

```python
filelist_request = service.fetch_filelist("/")
```

Get the latest saved remote file list for the given directory, and display it:

```python
filelist = service.get_filelist("/")

# Display file list
if filelist:
    print("File list received:")
    if not filelist.files:
        print("\tEmpty file list")
    for file in filelist.files:
        print(
            f"\t{file.name + ('/' if file.is_directory else ''):<12}\
        t{str(file.size) + ' bytes':>12}\tLast Modified: {file.modified}"
        )
else:
    print("No filelist found")
```

8. Time Correlation (TCO)

The Time Correlation API provides methods that you can use to programmatically interact with a Yamcs TCO service.

8.1 Reference

8.1.1 Client

class yamcs.client.**TCOClient**(*ctx: Context*, *instance: str*, *service: str*)

Client for interacting with a Time Correlation service managed by Yamcs.

add_tof_interval(*start: datetime*, *stop: datetime*, *polynomial: List[float]*)

Defines a ToF interval for the ERT range `[start, stop]`, specifying a polynomial function of the form: *tof = a + bx + cx^2 + ...* where *x* is ERT minus the provided start date.

Parameters

- **start** – ERT start
- **stop** – ERT stop
- **polynomial** – Coefficients in the order `[a, b, c, ...]`

add_tof_intervals(*intervals: List[TofInterval]*)

Adds multiple ToF intervals at once.

Parameters

intervals – List of ToF intervals.

get_status() → *TCOStatus*

Retrieve the TCO status.

override_coefficients(*utc: datetime*, *obt: int*, *gradient: float = 0*, *offset: float = 0*)

Manually override the assocation between UTC and onboard time.

> **Note**
>
> If later on you want to revert to automatically computed coefficients, use *reset_coefficients()*.

Parameters

- **utc** – UTC
- **obt** – Onboard time
- **gradient** – Gradient
- **offset** – Offset

reconfigure(*accuracy: float | None = None*, *validity: float | None = None*, *ob_delay: float | None = None*, *default_tof: float | None = None*)

Updates one or more TCO options

Parameters

- **accuracy** – Accuracy in seconds.
- **validity** – Validity in seconds.
- **ob_delay** – Onboard delay in seconds.
- **default_tof** – Default ToF in seconds. This value is used if the ToF estimator does not find a matching interval.

remove_tof_intervals(*start: datetime*, *stop: datetime*)

Removes previously registered ToF intervals whose start date falls in the specified range `[start, stop]`.

Parameters

- **start** – ERT start
- **stop** – ERT stop

reset_coefficients()

Resets current TCO coefficients, as well as any collected samples.

8.1.2 Model

class yamcs.client.**TCOCoefficients**(*proto*)

Bases: object

TCO Coefficients

property gradient: float | None

property obt: int | None

property offset: float | None

property utc: datetime | None

class yamcs.client.**TCOSample**(*proto*)

Bases: object

property obt: int | None

property utc: datetime | None

class yamcs.client.**TCOStatus**(*proto*)

Bases: object

TCO Status

property coefficients: *TCOCoefficients* | None

Current coefficients. Or None if the synchronization is not yet established.

property coefficients_time: datetime | None

Time when the coefficients have been computed

property deviation: float | None

Last computed deviation

property samples: List[*TCOSample*]

The last accumulated samples

class yamcs.client.**TofInterval**(*start: datetime*, *stop: datetime*, *polynomial: List[float]*)

Bases: object

ToF interval for the ERT range [start, stop], specifying a polynomial function of the form: *tof = a + bx + cx^2 + ...* where *x* is ERT minus the provided start date.

Parameters

- **start** – ERT start
- **stop** – ERT stop
- **polynomial** – Coefficients in the order [a, b, c, ...]

8.2 Snippets

Create a *TCOClient* for a specific instance:

```
from yamcs.client import YamcsClient

client = YamcsClient('localhost:8090')
tco = client.get_tco_client(instance='pus', service='tco0')
```

9. Timeline

The Timeline API provides methods that you can use to programmatically work with Yamcs bands and items.

9.1 Reference

9.1.1 Client

class yamcs.client.**TimelineClient**(*ctx: Context*, *instance: str*)

Client for working with Yamcs timeline.

delete_band(*band: str*)

Delete a band.

Parameters
band – Band identifier.

delete_item(*item: str*)

Delete an item.

Parameters
item – Item identifier.

delete_view(*view: str*)

Delete a view.

Parameters
view – View identifier.

get_band(*id: str*) → *Band*

Fetch a band by its identifier.

Parameters
id – Band identifier

get_item(*id: str*) → *Item*

Fetch an item by its identifier.

Parameters
id – Item identifier

get_view(*id: str*) → *View*

Fetch a view by its identifier.

Parameters
id – View identifier

list_bands() → Iterable[*Band*]

List the bands.

list_items(*band: str | None = None*, *start: datetime | None = None*, *stop: datetime | None = None*, *page_size: int = 500*) → Iterable[*Item*]

List the items.

Parameters

- **band** – Return only items matching the specified band
- **start** – Minimum stop time of the returned items (exclusive)
- **stop** – Maximum start time of the returned items (exclusive)
- **page_size** – Page size of underlying requests. Higher values imply less overhead, but risk hitting the maximum message size limit.

list_views() → Iterable[*View*]

List the views.

save_band(*band:* Band)

Save or update a band.

Parameters
band – Band object

save_item(*item:* Item)

Save or update an item.

Parameters
item – Item object

save_view(*view:* View)

Save or update a view.

Parameters
view – View object

9.1.2 Model

class yamcs.client.**Band**(*proto*)

Bases: ABC

Superclass for bands. Implementations:

- *TimeRuler*
- *ItemBand*
- *ParameterPlot*
- *ParameterStateBand*
- *Spacer*
- *CommandBand*

property band_type: str

Type of band.

property description: str

Description of this band.

property id: str

Band identifier.

property name: str

Name of this band.

class yamcs.client.**CommandBand**(*proto=None*)

Bases: *Band*

Display issued commands.

class yamcs.client.**Item**(*proto=None*)

Bases: object

property activity: Activity | None

Activity definition.

property background_color: str | None

CSS color string.

property border_color: str | None

CSS color string.

property border_width: int | None

property corner_radius: int | None

property duration: timedelta

Item duration.

property id: str

Item identifier.

property item_type: str

Type of item.

property margin_left: int | None

property name: str

Name of this item.

property start: datetime

Item start time.

property tags: List[str]

Item tags. Used by bands to filter what is visible.

property text_color: str | None

CSS color string.

property text_size: int | None

class yamcs.client.**ItemBand**(*proto=None*)

Bases: *Band*

Show a selection of timeline items.

property frozen: bool

Fix this line to the top of the view. Frozen bands are always rendered above other bands.

property item_background_color: str

CSS color string.

property item_border_color: str

CSS color string.

property item_border_width: int

property item_corner_radius: int

property item_height: int

property item_margin_left: int

property item_text_color: str

CSS color string.

property item_text_overflow: str

One of `show`, `clip`, or `hide`.

property item_text_size: int

property margin_bottom: int

property margin_top: int

property multiline: bool

Draw items on multiple lines if otherwise there would be collisions.

property space_between_items: int

In case of multilining, this indicates the minimum horizontal space between items. If an item does not meet this treshold, it gets rendered on a different line.

property space_between_lines: int

In case of multilining, this indicates the vertical space between lines.

property tags: List[str]

Item tags that this band filters on.

class yamcs.client.**ParameterPlot**(*proto=None*)

Bases: *Band*

Plot the values of a numeric parameter.

Added in version 1.11.2: Compatible with Yamcs 5.11.2 onwards

property frozen: bool

Fix this line to the top of the view. Frozen bands are always rendered above other bands.

property height: int

Band height

property maximum: float | None

Maximum value to show on Y-axis. Set to `None` for fitting actual data

property maximum_fraction_digits: int

Maximum fraction digits

property minimum: float | None

Minimum value to show on Y-axis. Set to `None` for fitting actual data

property minimum_fraction_digits: int

Minimum fraction digits

traces: List[*Trace*]

Plot lines.

property zero_line_color: str

Color of the zero line

property zero_line_width: int

Thickness of the zero line. 0 is invisible

class yamcs.client.**ParameterStateBand**(*proto=None*)

Bases: *Band*

Show state transitions of a parameter

Added in version 1.11.2: Compatible with Yamcs 5.11.2 onwards

property frozen: bool

Fix this line to the top of the view. Frozen bands are always rendered above other bands.

property height: int

Band height

mappings: List[*ValueMapping* | *RangeMapping*]

Map engineering values to a label and/or color. Mappings are applied in order.

property parameter: str

Qualified parameter name

class yamcs.client.**RangeMapping**(*start: float*, *end: float*, *label: str | None = None*, *color: str | None = None*)

Bases: object

Maps a value to a label and/or color.

color: str | None = None

If specified, show states of this value (or mapped label) in this color

end: float

Match engineering value lesser or equal than the provided end value

label: str | None = None

If specified, map the provided value to this label

start: float

Match engineering value greater or equal than the provided start value

class yamcs.client.**Spacer**(*proto=None*)

Bases: *Band*

Insert empty vertical space.

property height: int

Spacer height

class yamcs.client.**TimeRuler**(*proto=None*)

Bases: *Band*

Displays absolute time, formatted in a timezone of choice.

property timezone: str

IANA timezone name.

Corresponds with the third column of the following table: https://data.iana.org/time-zones/data/zone1970.tab

In addition, the name *UTC* is supported.

class yamcs.client.**Trace**(*parameter: str*, *line_color: str*, *visible: bool = True*, *line_width: int = 1*, *fill: bool = False*, *fill_color: str = '#dddddd'*, *min_max: bool = False*, *min_max_opacity: float = 0.17*)

Bases: object

A trace on a *ParameterPlot*.

fill: *bool* = False

fill_color: *str* = '#dddddd'

line_color: *str*

line_width: *int* = 1

min_max: *bool* = False

min_max_opacity: *float* = 0.17

parameter: *str*

visible: *bool* = True

class yamcs.client.**ValueMapping**(*value: Any*, *label: str | None = None*, *color: str | None = None*)

Bases: object

Maps a value to a label and/or color

color: *str | None* = None

If specified, show states of this value (or mapped label) in this color

label: *str | None* = None

If specified, map the provided value to this label

value: *Any*

Engineering value to match

class yamcs.client.**View**(*proto=None*)

Bases: object

property bands: *List[Band]*

Bands included in this view.

property description: *str*

Description of this view.

property id: *str*

View identifier.

property name: *str*

Name of this view.

9.2 Snippets

Create a *TimelineClient* for a specific instance:

```
from yamcs.client import YamcsClient

client = YamcsClient("localhost:8090")
timeline = client.get_timeline_client(instance="simulator")
```

Create a few *Band* objects:

```
from yamcs.client import ItemBand, TimeRuler

utc_time = TimeRuler()
utc_time.name = "UTC"
timeline.save_band(utc_time)

local_time = TimeRuler()
local_time.name = "Local"
```

(continues on next page)

(continued from previous page)

```
local_time.timezone = "Europe/Brussels"
timeline.save_band(local_time)

group_a = ItemBand()
group_a.name = "Group A"
group_a.tags = ["group-a"]
timeline.save_band(group_a)

group_b = ItemBand()
group_b.name = "Group B"
group_b.tags = ["group-b"]
group_b.item_border_color = "#ff4500"
group_b.item_background_color = "#ffa500"
timeline.save_band(group_b)
```

Create some *Item* objects. Bands of type *ItemBand* will display items with matching tags:

```
from datetime import datetime, timedelta, timezone

from yamcs.client import Item

now = datetime.now(tz=timezone.utc)

for i in range(10):
    item = Item()
    item.name = f"A {i + 1}"
    item.start = now + timedelta(seconds=i * 7200)
    item.duration = timedelta(seconds=3600)
    item.tags = ["group-a"]
    timeline.save_item(item)

    item = Item()
    item.name = f"B {i + 1}"
    item.start = now + timedelta(seconds=3600 + (i * 7200))
    item.duration = timedelta(seconds=3600)
    item.tags = ["group-b"]
    timeline.save_item(item)
```

Create a *View* showing all bands:

```
from yamcs.client import View

view = View()
view.name = "Two groups"
view.bands = [utc_time, local_time, group_a, group_b]
timeline.save_view(view)
```

To update a *Band*, *Item* or *View* use the same save methods as for inserting. When saving or fetching these objects they are assigned a server identifier that is used to detect whether further saves require an insert or update.

```
group_a.description = "A few random items"
timeline.save_band(group_a)
```

```
for item in timeline.list_items():
    item.tags.append("example")
    timeline.save_item(item)
```

Create a view with *ParameterStateBand* and *ParameterPlot* bands:

```
from yamcs.client import (
    ParameterPlot,
    ParameterStateBand,
    RangeMapping,
    Trace,
    ValueMapping,
    View,
)
```

(continues on next page)

(continued from previous page)

```
states = ParameterStateBand()
states.name = "State example"
states.parameter = "/YSS/SIMULATOR/BatteryVoltage2"
states.mappings.append(RangeMapping(40, 56, label="LOW", color="#ff0000"))
states.mappings.append(ValueMapping(57, label="OK", color="#00ff00"))
states.mappings.append(RangeMapping(58, 70, label="HIGH", color="#ff0000"))
timeline.save_band(states)

plot = ParameterPlot()
plot.name = "Plot example"
plot.maximum = 65
trace = Trace(
    parameter="/YSS/SIMULATOR/BatteryVoltage2",
    line_color="#ffff00",
    fill=True,
)
plot.traces.append(trace)
timeline.save_band(plot)

view = View()
view.name = "Parameter examples"
view.bands = [states, plot]
timeline.save_view(view)
```

10. Examples

All of these examples run against a simulation that is used to develop and demo Yamcs. This setup is linked to a simple simulator which emits TM in the form of CCSDS packets and accepts a few telecommands as well.

Most of the telemetered parameters are in a space system called `/YSS/SIMULATOR`.

10.1 alarms.py

```python
from time import sleep

from yamcs.client import YamcsClient


def receive_callbacks():
    """Registers an alarm callback."""

    def callback(alarm_update):
        print("Alarm Update:", alarm_update)

    processor.create_alarm_subscription(callback)


def acknowledge_all():
    """Acknowledges all active alarms."""
    for alarm in processor.list_alarms():
        if not alarm.is_acknowledged:
            processor.acknowledge_alarm(
                alarm.name,
                alarm.sequence_number,
                comment="false alarm",
            )


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    processor = client.get_processor(instance="simulator", processor="realtime")
    receive_callbacks()
    sleep(10)

    print("Acknowledging all...")
    acknowledge_all()

    # If a parameter remains out of limits, a new alarm instance is created
    # on the next value update.  So you would keep receiving callbacks on
    # the subscription.

    # The subscription is non-blocking. Prevent the main
    # thread from exiting
    while True:
        sleep(10)
```

10.2 archive_breakdown.py

```
from yamcs.client import YamcsClient


def print_packet_count():
    """Print the number of packets grouped by packet name."""
    for name in archive.list_packet_names():
        packet_count = 0
        for group in archive.list_packet_histogram(name):
            for rec in group.records:
                packet_count += rec.count
        print(f"  {name: <40} {packet_count: >20}")


def print_pp_groups():
    """Print the number of processed parameter frames by group name."""
    for group in archive.list_processed_parameter_groups():
        frame_count = 0
        for pp_group in archive.list_processed_parameter_group_histogram(group):
            for rec in pp_group.records:
                frame_count += rec.count
        print(f"  {group: <40} {frame_count: >20}")


def print_event_count():
    """Print the number of events grouped by source."""
    for source in archive.list_event_sources():
        event_count = 0
        for group in archive.list_event_histogram(source):
            for rec in group.records:
                event_count += rec.count
        print(f"  {source: <40} {event_count: >20}")


def print_command_count():
    """Print the number of commands grouped by name."""
    mdb = client.get_mdb(instance="simulator")
    for command in mdb.list_commands():
        total = 0
        for group in archive.list_command_histogram(command.qualified_name):
            for rec in group.records:
                total += rec.count
        print(f"  {command.qualified_name: <40} {total: >20}")


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    archive = client.get_archive(instance="simulator")

    print("Packets:")
    print_packet_count()

    print("\nProcessed Parameter Groups:")
    print_pp_groups()

    print("\nEvents:")
    print_event_count()

    print("\nCommands:")
    print_command_count()
```

10.3 archive_retrieval.py

```
from datetime import datetime, timedelta, timezone
from itertools import islice

from yamcs.client import YamcsClient
```

(continues on next page)

(continued from previous page)

```
def print_last_packets():
    """Print the last 10 packets."""
    for packet in islice(archive.list_packets(descending=True), 0, 10):
        print(packet)


def print_packet_range():
    """Print the range of archived packets."""
    first_packet = next(iter(archive.list_packets()))
    last_packet = next(iter(archive.list_packets(descending=True)))
    print("First packet:", first_packet)
    print("Last packet:", last_packet)

    td = last_packet.generation_time - first_packet.generation_time
    print("Timespan:", td)


def iterate_specific_packet_range():
    """Count the number of packets in a specific range."""
    now = datetime.now(tz=timezone.utc)
    start = now - timedelta(hours=1)

    total = 0
    for packet in archive.list_packets(start=start, stop=now):
        total += 1
        # print(packet)
    print("Found", total, "packets in range")


def export_raw_packets():
    """Export raw packet binary."""
    now = datetime.now(tz=timezone.utc)
    start = now - timedelta(hours=1)

    with open("/tmp/dump.raw", "wb") as f:
        for chunk in archive.export_packets(start=start, stop=now):
            f.write(chunk)


def iterate_specific_event_range():
    """Count the number of events in a specific range."""
    now = datetime.now(tz=timezone.utc)
    start = now - timedelta(hours=1)

    total = 0
    for event in archive.list_events(start=start, stop=now):
        total += 1
        # print(event)
    print("Found", total, "events in range")


def print_last_values():
    """Print the last 10 values."""
    iterable = archive.list_parameter_values(
        "/YSS/SIMULATOR/BatteryVoltage1", descending=True
    )
    for pval in islice(iterable, 0, 10):
        print(pval)


def iterate_specific_parameter_range():
    """Count the number of parameter values in a specific range."""
    now = datetime.now(tz=timezone.utc)
    start = now - timedelta(hours=1)

    total = 0
    for pval in archive.list_parameter_values(
        "/YSS/SIMULATOR/BatteryVoltage1", start=start, stop=now
    ):
        total += 1
```

(continues on next page)

(continued from previous page)

```
        # print(pval)
    print("Found", total, "parameter values in range")


def stream_specific_parameter_range():
    """Stream one or more parameters in a specific range."""
    now = datetime.now(tz=timezone.utc)
    start = now - timedelta(hours=1)

    total = 0
    for pdata in archive.stream_parameter_values(
        ["/YSS/SIMULATOR/BatteryVoltage1", "/YSS/SIMULATOR/BatteryVoltage2"],
        start=start,
        stop=now,
    ):
        total += 1
        # print(pdata)
    print("Found", total, "updates in range")


def print_last_commands():
    """Print the last 10 commands."""
    iterable = archive.list_command_history(descending=True)
    for entry in islice(iterable, 0, 10):
        print(entry)


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    archive = client.get_archive(instance="simulator")

    print("Last 10 packets:")
    print_last_packets()

    print("\nPacket range:")
    print_packet_range()

    print("\nIterate specific packet range:")
    iterate_specific_packet_range()

    print("\nIterate specific event range:")
    iterate_specific_event_range()

    print("\nLast 10 parameter values:")
    print_last_values()

    print("\nIterate specific parameter range:")
    iterate_specific_parameter_range()

    print("\nStream specific parameter range:")
    stream_specific_parameter_range()

    print("\nLast 10 commands:")
    print_last_commands()
```

10.4 authenticate.py

```
from time import sleep

from yamcs.client import Credentials, YamcsClient

# For this example to work, enable security in Yamcs by
# configuring appropriate authentication modules.


def authenticate_with_username_password():
    """Authenticate by directly providing username/password to Yamcs."""
    credentials = Credentials(username="admin", password="password")
```

(continues on next page)

(continued from previous page)

```
    client = YamcsClient("localhost:8090", credentials=credentials)

    for link in client.list_links("simulator"):
        print(link)


def authenticate_with_access_token(access_token):
    """Authenticate using an existing access token."""
    credentials = Credentials(access_token=access_token)
    client = YamcsClient("localhost:8090", credentials=credentials)

    for link in client.list_links("simulator"):
        print(link)


def impersonate_with_client_credentials():
    credentials = Credentials(
        client_id="cf79cfbd-ed01-4ae2-93e1-c606a2ebc36f",
        client_secret="!#?hgbu1*3",
        become="admin",
    )
    client = YamcsClient("localhost:8090", credentials=credentials)
    print("have", client.get_user_info().username)
    while True:
        print(client.get_time("simulator"))
        sleep(1)


def authenticate_with_basic_auth():
    """Authenticate by providing username/password on each request to Yamcs."""
    from yamcs.client import BasicAuthCredentials

    credentials = BasicAuthCredentials(username="admin", password="password")
    client = YamcsClient("localhost:8090", credentials=credentials)

    for link in client.list_links("simulator"):
        print(link)


if __name__ == "__main__":
    print("Authenticate with username/password")
    authenticate_with_username_password()

    print("\nImpersonate with client credentials")
    impersonate_with_client_credentials()
```

10.5 ccsds_completeness.py

> **Note**
>
> CCSDS Completeness is a concept specific to CCSDS-style packets. If you store a different type of packet in Yamcs, then this code example is not applicable.

```
from datetime import datetime, timedelta, timezone

from yamcs.client import YamcsClient


def print_latest():
    """
    Prints completeness records for the last two days
    """
    now = datetime.now(tz=timezone.utc)
    start = now - timedelta(days=2)
```

(continues on next page)

(continued from previous page)

```
    # Results are returned in records per 'group'.
    # A completeness group matches with a CCSDS APID.

    # We may receive multiple groups with the same APID
    # depending on how much data there is for the selected
    # range.

    # Combine all returned pages by APID
    records_by_apid = {}
    for group in archive.list_completeness_index(start=start, stop=now):
        if group.name not in records_by_apid:
            records_by_apid[group.name] = []
        records_by_apid[group.name].extend(group.records)

    for apid, records in records_by_apid.items():
        print("APID:", apid)

        total = 0
        for rec in records:
            print("  -", rec)
            total += rec.count
        print(f"  --> Total packets for {apid}: {total}")


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    archive = client.get_archive(instance="simulator")
    print_latest()
```

10.6 commanding.py

```
from time import sleep

from yamcs.client import VerificationConfig, YamcsClient


def issue_command():
    """Issue a command to turn battery 1 off."""
    command = processor.issue_command(
        "/YSS/SIMULATOR/SWITCH_VOLTAGE_OFF", args={"voltage_num": 1}
    )
    print("Issued", command)


def issue_command_modify_verification():
    """Issue a command with changed verification."""
    verification = VerificationConfig()
    verification.disable("Started")
    verification.modify_check_window("Queued", 1, 5)

    command = processor.issue_command(
        "/YSS/SIMULATOR/SWITCH_VOLTAGE_OFF",
        args={"voltage_num": 1},
        verification=verification,
    )

    print("Issued", command)


def issue_command_no_verification():
    """Issue a command with no verification."""
    verification = VerificationConfig()
    verification.disable()

    command = processor.issue_command(
        "/YSS/SIMULATOR/SWITCH_VOLTAGE_OFF",
        args={"voltage_num": 1},
        verification=verification,
```

(continues on next page)

(continued from previous page)

```
    )

    print("Issued", command)


def monitor_command():
    """Monitor command completion."""
    conn = processor.create_command_connection()

    command1 = conn.issue("/YSS/SIMULATOR/SWITCH_VOLTAGE_OFF", args={"voltage_num": 1})

    # Issue 2nd command only if the previous command was completed successfully.
    command1.await_complete()
    if command1.is_success():
        conn.issue("/YSS/SIMULATOR/SWITCH_VOLTAGE_ON", args={"voltage_num": 1})
    else:
        print("Command 1 failed:", command1.error)


def monitor_acknowledgment():
    """Monitor command acknowledgment."""
    conn = processor.create_command_connection()

    command = conn.issue("/YSS/SIMULATOR/SWITCH_VOLTAGE_OFF", args={"voltage_num": 1})

    ack = command.await_acknowledgment("Acknowledge_Sent")
    print(ack.status)


def listen_to_command_history():
    """Receive updates on command history updates."""

    def tc_callback(rec):
        print("TC:", rec)

    processor.create_command_history_subscription(on_data=tc_callback)


def tm_callback(delivery):
    for parameter in delivery.parameters:
        print("TM:", parameter)


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    processor = client.get_processor("simulator", "realtime")

    print("Start to listen to command history")
    listen_to_command_history()

    issue_command_no_verification()
    issue_command_modify_verification()

    print("Issue a command")
    issue_command()

    # Monitor the voltage parameter to confirm that it is 0
    subscription = processor.create_parameter_subscription(
        ["/YSS/SIMULATOR/BatteryVoltage1"], on_data=tm_callback
    )

    # Subscription is non-blocking
    sleep(20)
```

10.7 cop1.py

```
from time import sleep
```

(continues on next page)

(continued from previous page)

```
from yamcs.client import YamcsClient


def callback(status):
    print("<callback> status:", status)


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    link = client.get_link("simulator", link="UDP_FRAME_OUT.vc0")

    config = link.get_cop1_config()
    print(config)

    print("Changing COP1 configuration")
    link.update_cop1_config(t1=3.1, tx_limit=4)

    monitor = link.create_cop1_subscription(on_data=callback)
    print("COP1 status subscribed.")

    sleep(5)

    print("Disabling COP1....")
    link.disable_cop1()

    sleep(3)
    print("Initializing COP1 with CLCW_CHECK")
    print("  (if no CLCW is received, COP1 will be suspended in 3 seconds)")
    link.initialize_cop1("WITH_CLCW_CHECK", clcw_wait_timeout=3)

    sleep(5)

    if monitor.state == "SUSPENDED":
        print("Resuming COP1")
        link.resume_cop1()

    sleep(3)

    print("Disabling COP1....")
    link.disable_cop1()

    sleep(3)

    print("Initializing COP1 with set v(R)=200")
    print("  (if no CLCW is received, COP1 will be suspended in 3 seconds)")
    link.initialize_cop1("SET_VR", v_r=200)

    sleep(2)
```

10.8 file_transfer.py

```
import io
import time

from yamcs.client import YamcsClient


def upload_file():
    """Snippet used in docs to initiate upload."""
    # Transfer myfile from bucket to spacecraft
    upload = service.upload(out_bucket.name, "myfile", "/CF:/mytarget")
    upload.await_complete(timeout=10)

    if not upload.is_success():
        print("Upload failure:", upload.error)
    else:
        print(f"Successfully uploaded {upload.remote_path} ({upload.size} bytes)")
```

(continues on next page)

(continued from previous page)

```
def upload_file_extra():
    """Snippet used in docs to initiate upload with extra parameters."""
    # Transfer myfile, but use an alternative destination entity
    upload = service.upload(
        out_bucket.name, "myfile", "/CF:/mytarget", destination_entity="target2"
    )
    upload.await_complete(timeout=10)

    if not upload.is_success():
        print("Upload failure:", upload.error)
    else:
        print(f"Successfully uploaded {upload.remote_path} ({upload.size} bytes)")


def upload_file_options():
    """Snippet used in docs to initiate upload with extra options."""
    upload = service.upload(
        out_bucket.name, "myfile", "/CF:/mytarget", options={"reliable": True}
    )
    upload.await_complete(timeout=60)

    if not upload.is_success():
        print("Upload failure:", upload.error)
    else:
        print(f"Successfully uploaded {upload.remote_path} ({upload.size} bytes)")


def download_file():
    """Snippet used in docs to initiate download."""
    # Download todownload from the remote
    download = service.download(in_bucket.name, "todownload")
    download.await_complete(timeout=10)

    if not download.is_success():
        print("Download failure:", download.error)
    else:
        print(f"Successfully downloaded {download.remote_path} ({download.size} bytes)")


def subscribe_filelist():
    """Snippet used in docs to subscribe to filelist updates."""
    subscription = service.create_filelist_subscription(on_data=filelist_callback)
    return subscription


def filelist_callback(filelist):
    global updated
    updated = True
    print(f"Filelist updated with {len(filelist.files)} files or directories")


def fetch_filelist():
    """Snippet used in docs to fetch a remote filelist."""
    filelist_request = service.fetch_filelist("/")
    return filelist_request


def get_filelist():
    """Snippet used in docs to get a saved filelist and display it."""
    filelist = service.get_filelist("/")

    # Display file list
    if filelist:
        print("File list received:")
        if not filelist.files:
            print("\tEmpty file list")
        for file in filelist.files:
            print(
                f"\t{file.name + ('/' if file.is_directory else ''):<12}\
            t{str(file.size) + ' bytes':>12}\tLast Modified: {file.modified}"
```

(continues on next page)

(continued from previous page)

```
            )
    else:
        print("No filelist found")


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    storage = client.get_storage_client()
    cfdp = client.get_file_transfer_client(instance="cfdp")

    # Use pre-existing buckets, one for each direction
    out_bucket = storage.get_bucket("cfdpUp")
    in_bucket = storage.get_bucket("cfdpDown")

    # Prepare a sample file
    file_like = io.StringIO("Sample file content")
    out_bucket.upload_object("myfile", file_like)

    # Assume only one CFDP service
    service = next(iter(cfdp.list_services()))

    # Show transfer service capabilities
    print(service.capabilities)

    # Show possible transfer options
    for option in service.transfer_options:
        print(option)

    if service.capabilities.upload:
        upload_file()

        upload_file_extra()

        upload_file_options()

    if service.capabilities.download:
        download_file()

    # DIRECTORY LISTING
    if service.capabilities.filelist:
        updated = False

        subscribe_filelist()

        fetch_filelist()

        start = time.time()
        while not updated and time.time() - start < 20:
            time.sleep(0.1)

        # The saved filelist can either be retrieved from the subscription callback or
        # via the get_filelist function
        get_filelist()
```

10.9 links.py

```
from time import sleep

from yamcs.client import YamcsClient


def enable_link(link):
    """Enable a link."""
    link.enable_link()


def run_action(link, action_id, message=None):
    """Run an action."""
```

(continues on next page)

(continued from previous page)

```
    link.run_action(action_id, message)


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    link = client.get_link("simulator", link="tm_dump")

    print("Enabling link")
    enable_link(link)

    subscription = client.create_link_subscription("simulator")

    sleep(10)

    print("-----")
    # Retrieve the latest data link state from local cache:
    print("Last values from cache:")
    for link in subscription.list_links():
        print(link)
```

10.10 events.py

```
from time import sleep

from yamcs.client import YamcsClient


def listen_to_event_updates():
    """Subscribe to events."""

    def callback(event):
        print("Event:", event)

    client.create_event_subscription(instance="simulator", on_data=callback)

    sleep(5)  # Subscription is non-blocking


def send_event():
    """Post an event."""
    client.send_event(instance="simulator", message="hello world")


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    listen_to_event_updates()

    print("Sending an event:")
    send_event()

    sleep(5)
```

10.11 mission_time.py

```
from time import sleep

from yamcs.client import YamcsClient


def callback(dt):
    print("Mission time:", dt)


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
```

(continues on next page)

(continued from previous page)

```
    subscription = client.create_time_subscription("simulator", callback)

    sleep(6)

    print("-----")
    # You don't have to use the on_data callback. You could also
    # directly retrieve the latest state from a local cache:
    print("Last time from cache:", subscription.time)

    # But then maybe you don't need a subscription, so do simply:
    time = client.get_time("simulator")
    print("Mission time (fresh from server)", time)
```

10.12 packet_subscription.py

```
from binascii import hexlify
from time import sleep

from yamcs.client import YamcsClient


def receive_callbacks():
    """Shows how to receive callbacks on packet updates."""

    def print_data(packet):
        hexpacket = hexlify(packet.binary).decode("ascii")
        print(packet.generation_time, ":", hexpacket)

    processor.create_container_subscription(
        containers=["/YSS/SIMULATOR/FlightData", "/YSS/SIMULATOR/Power"],
        on_data=print_data,
    )


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    processor = client.get_processor("simulator", "realtime")

    print("\nReceive callbacks")
    receive_callbacks()

    sleep(5)  # Subscription is non-blocking
```

10.13 parameter_subscription.py

```
from time import sleep

from yamcs.client import YamcsClient


def poll_values():
    """Shows how to poll values from the subscription."""
    subscription = processor.create_parameter_subscription(
        ["/YSS/SIMULATOR/BatteryVoltage1"]
    )

    sleep(5)
    print("Latest value:")
    print(subscription.get_value("/YSS/SIMULATOR/BatteryVoltage1"))

    sleep(5)
    print("Latest value:")
    print(subscription.get_value("/YSS/SIMULATOR/BatteryVoltage1"))
```

(continues on next page)

(continued from previous page)

```
def receive_callbacks():
    """Shows how to receive callbacks on value updates."""

    def print_data(data):
        for parameter in data.parameters:
            print(parameter)

    processor.create_parameter_subscription(
        "/YSS/SIMULATOR/BatteryVoltage1", on_data=print_data
    )
    sleep(5)  # Subscription is non-blocking


def manage_subscription():
    """Shows how to interact with a parameter subscription."""
    subscription = processor.create_parameter_subscription(
        ["/YSS/SIMULATOR/BatteryVoltage1"]
    )

    sleep(5)

    print("Adding extra items to the existing subscription...")
    subscription.add(
        [
            "/YSS/SIMULATOR/Alpha",
            "/YSS/SIMULATOR/BatteryVoltage2",
            "MDB:OPS Name/SIMULATOR_PrimBusVoltage1",
        ]
    )

    sleep(5)

    print("Shrinking subscription...")
    subscription.remove("/YSS/SIMULATOR/Alpha")

    print("Cancelling the subscription...")
    subscription.cancel()

    print("Last values from cache:")
    print(subscription.get_value("/YSS/SIMULATOR/BatteryVoltage1"))
    print(subscription.get_value("/YSS/SIMULATOR/BatteryVoltage2"))
    print(subscription.get_value("/YSS/SIMULATOR/Alpha"))
    print(subscription.get_value("MDB:OPS Name/SIMULATOR_PrimBusVoltage1"))


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    processor = client.get_processor("simulator", "realtime")

    print("Poll value cache")
    poll_values()

    print("\nReceive callbacks")
    receive_callbacks()

    print("\nModify the subscription")
    manage_subscription()
```

10.14 plot_with_matplotlib.py

Note

To run this example, install matplotlib:

```
pip install matplotlib
```

```python
from datetime import datetime, timedelta, timezone

import matplotlib.dates as mdates
import matplotlib.pyplot as plt
from yamcs.client import YamcsClient

if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    archive = client.get_archive(instance="simulator")

    stop = datetime.now(tz=timezone.utc)
    start = stop - timedelta(hours=1)

    samples = archive.downsample_mean("/YSS/SIMULATOR/Altitude", start=start, stop=stop)
    x = [s.time for s in samples]
    y = [s.avg for s in samples]

    mx = [mdates.date2num(t) for t in x]
    my = [val if val is not None else float("nan") for val in y]

    plt.subplot(2, 1, 1)
    plt.title("Sampled at " + str(stop))
    plt.plot(mx, my)
    plt.xlim(start, stop)
    plt.ylabel("Altitude")
    plt.grid()
    plt.xticks(rotation=45, ha="right")

    samples = archive.downsample_mean("/YSS/SIMULATOR/SinkRate", start=start, stop=stop)
    x = [s.time for s in samples]
    y = [s.avg for s in samples]

    mx = [mdates.date2num(t) for t in x]
    my = [val if val is not None else float("nan") for val in y]

    plt.subplot(2, 1, 2)
    plt.plot(mx, my)
    plt.xlim(start, stop)
    plt.xlabel("UTC")
    plt.ylabel("Sink Rate")
    plt.grid()
    plt.xticks(rotation=45, ha="right")

    plt.tight_layout()
    plt.show()
```

10.15 query_mdb.py

```python
from yamcs.client import YamcsClient


def print_space_systems():
    """Print all space systems."""
    for space_system in mdb.list_space_systems():
        print(space_system)


def print_parameters():
    """Print all float parameters."""
    for parameter in mdb.list_parameters(parameter_type="float"):
        print(parameter)


def print_commands():
    """Print all commands."""
    for command in mdb.list_commands():
        print(command)
```

(continues on next page)

(continued from previous page)

```
def find_parameter():
    """Find one parameter."""
    p1 = mdb.get_parameter("/YSS/SIMULATOR/BatteryVoltage2")
    print("Via qualified name:", p1)

    p2 = mdb.get_parameter("MDB:OPS Name/SIMULATOR_BatteryVoltage2")
    print("Via domain-specific alias:", p2)


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    mdb = client.get_mdb(instance="simulator")

    print("\nSpace systems:")
    print_space_systems()

    print("\nParameters:")
    print_parameters()

    print("\nCommands:")
    print_commands()

    print("\nFind a specific parameter using different names")
    find_parameter()
```

10.16 read_write_parameters.py

```
from yamcs.client import YamcsClient


def print_cached_value():
    """Print a single value from server cache."""
    pval = processor.get_parameter_value("/YSS/SIMULATOR/BatteryVoltage1")
    print(pval)


def print_realtime_value():
    """Print a newly processed value."""
    pval = processor.get_parameter_value(
        "/YSS/SIMULATOR/BatteryVoltage2", from_cache=False, timeout=5
    )
    print(pval)


def print_current_values():
    """Print multiple parameters from server cache."""
    pvals = processor.get_parameter_values(
        ["/YSS/SIMULATOR/BatteryVoltage1", "/YSS/SIMULATOR/BatteryVoltage2"]
    )
    print("battery1", pvals[0])
    print("battery2", pvals[1])


def write_value():
    """Writes to a software parameter."""
    processor.set_parameter_value("/YSS/SIMULATOR/AllowCriticalTC1", True)


def write_values():
    """Writes multiple software parameters."""
    processor.set_parameter_values(
        {
            "/YSS/SIMULATOR/AllowCriticalTC1": False,
            "/YSS/SIMULATOR/AllowCriticalTC2": False,
        }
    )
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    processor = client.get_processor(instance="simulator", processor="realtime")

    print("Fetch parameter value from cache")
    print_cached_value()

    print("\nFetch newly processed parameter value")
    print_realtime_value()

    print("\nFetch multiple parameters at the same time")
    print_current_values()

    print("\nWrite to a software parameter")
    write_value()

    print("\nWrite multiple software parameters at once")
    write_values()
```

10.17 reconnection.py

```
from time import sleep

from yamcs.client import ConnectionFailure, YamcsClient

"""
Subscription types are modeled as Python futures which cover
the lifetime of the underlying WebSocket call.

Therefore we can use methods like ``result()`` or
``add_done_callback(fn)`` to observe connection failures.

The following example uses ``result()`` to trigger
a delayed reconnection attempt.
"""


def print_data(data):
    for parameter in data.parameters:
        print(parameter)


while True:
    try:
        client = YamcsClient("localhost:8090")
        processor = client.get_processor("simulator", "realtime")
        subscription = processor.create_parameter_subscription(
            "/YSS/SIMULATOR/BatteryVoltage1", on_data=print_data
        )

        # Wait until WebSocket close
        subscription.result()
    except ConnectionFailure:
        print("Retrying in 5 seconds...")
        sleep(5)
```

10.18 timeline.py

```
from yamcs.client import YamcsClient


def create_bands():
    """Snippet used in docs to create a few bands."""
    global utc_time, local_time, group_a, group_b
    from yamcs.client import ItemBand, TimeRuler
```

(continues on next page)

(continued from previous page)

```
    utc_time = TimeRuler()
    utc_time.name = "UTC"
    timeline.save_band(utc_time)

    local_time = TimeRuler()
    local_time.name = "Local"
    local_time.timezone = "Europe/Brussels"
    timeline.save_band(local_time)

    group_a = ItemBand()
    group_a.name = "Group A"
    group_a.tags = ["group-a"]
    timeline.save_band(group_a)

    group_b = ItemBand()
    group_b.name = "Group B"
    group_b.tags = ["group-b"]
    group_b.item_border_color = "#ff4500"
    group_b.item_background_color = "#ffa500"
    timeline.save_band(group_b)


def create_items():
    """Snippet used in docs to create a few items."""
    from datetime import datetime, timedelta, timezone

    from yamcs.client import Item

    now = datetime.now(tz=timezone.utc)

    for i in range(10):
        item = Item()
        item.name = f"A {i + 1}"
        item.start = now + timedelta(seconds=i * 7200)
        item.duration = timedelta(seconds=3600)
        item.tags = ["group-a"]
        timeline.save_item(item)

        item = Item()
        item.name = f"B {i + 1}"
        item.start = now + timedelta(seconds=3600 + (i * 7200))
        item.duration = timedelta(seconds=3600)
        item.tags = ["group-b"]
        timeline.save_item(item)


def create_view():
    """Snippet used in docs to create a view."""
    from yamcs.client import View

    view = View()
    view.name = "Two groups"
    view.bands = [utc_time, local_time, group_a, group_b]
    timeline.save_view(view)


def edit_band():
    """Snippet used in docs to edit a band."""
    global group_a
    group_a.description = "A few random items"
    timeline.save_band(group_a)


def edit_fetched_items():
    """Snippet used in docs to edit a fetched band."""
    for item in timeline.list_items():
        item.tags.append("example")
        timeline.save_item(item)
```

(continues on next page)

(continued from previous page)

```
def create_parameter_bands():
    """Snippet used in docs to create parameter-based bands."""
    from yamcs.client import (
        ParameterPlot,
        ParameterStateBand,
        RangeMapping,
        Trace,
        ValueMapping,
        View,
    )

    states = ParameterStateBand()
    states.name = "State example"
    states.parameter = "/YSS/SIMULATOR/BatteryVoltage2"
    states.mappings.append(RangeMapping(40, 56, label="LOW", color="#ff0000"))
    states.mappings.append(ValueMapping(57, label="OK", color="#00ff00"))
    states.mappings.append(RangeMapping(58, 70, label="HIGH", color="#ff0000"))
    timeline.save_band(states)

    plot = ParameterPlot()
    plot.name = "Plot example"
    plot.maximum = 65
    trace = Trace(
        parameter="/YSS/SIMULATOR/BatteryVoltage2",
        line_color="#ffff00",
        fill=True,
    )
    plot.traces.append(trace)
    timeline.save_band(plot)

    view = View()
    view.name = "Parameter examples"
    view.bands = [states, plot]
    timeline.save_view(view)


if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    timeline = client.get_timeline_client("simulator")

    # Delete all
    for view in timeline.list_views():
        timeline.delete_view(view.id)
    for band in timeline.list_bands():
        timeline.delete_band(band.id)
    for item in timeline.list_items():
        timeline.delete_item(item.id)

    create_bands()
    create_items()
    create_view()
    edit_band()
    edit_fetched_items()
    create_parameter_bands()
```

10.19 write_mdb.py

If a system is marked as *writable* in the loader tree, Yamcs becomes responsible for updating the file, and the HTTP API can then be used to update it. This capability exists only for XTCE files.

Note that this functionality is experimental, and does not offer the same degree of customizations as when loading the MDB from a predefined XTCE file.

To demonstrate this capability, add a writable subsystem `test` to the Yamcs Quickstart example:

mdb/yamcs.myproject.yaml

```
mdb:
  - type: xtce
    args:
      file: mdb/xtce.xml
    subLoaders:
      - type: xtce
        writable: true
        args:
          file: /path/to/test.xml
```

Because the quickstart example works on a copy of the files (in `target/yamcs/`), we specify `test.xml` as an absolute rather than relative reference.

The initial content can just be an empty XTCE SpaceSystem, specifying its name:

mdb/test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<SpaceSystem xmlns="http://www.omg.org/spec/XTCE/20180204" name="test">
</SpaceSystem>
```

Then you can add parameter types and values like this:

```
from yamcs.client import NotFound, YamcsClient

"""
The following example requires a _writable_ system.

In this example the system ``/myproject/test`` is writable, whereas
the top-level system ``/myproject`` is populated based on a predefined
XTCE file (not managed by Yamcs itself).
"""

if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    mdb = client.get_mdb(instance="myproject")

    system = "/myproject/test"
    try:
        mdb.get_parameter_type(f"{system}/float_t")
    except NotFound:
        mdb.create_parameter_type(f"{system}/float_t", eng_type="float")

    try:
        mdb.get_parameter(f"{system}/testparam")
    except NotFound:
        mdb.create_parameter(
            f"{system}/testparam",
            data_source="LOCAL",
            parameter_type=f"{system}/float_t",
        )

    # MDB is present, now publish a parameter value
    processor = client.get_processor("myproject", "realtime")
    processor.set_parameter_value(f"{system}/testparam", 123.4)
```

Afterwards you should see that `test.xml` has been updated automatically:

mdb/test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<SpaceSystem xmlns="http://www.omg.org/spec/XTCE/20180204" name="test">
  <TelemetryMetaData>
    <ParameterTypeSet>
      <FloatParameterType sizeInBits="32" name="float_t">
        <UnitSet/>
      </FloatParameterType>
```

(continues on next page)

(continued from previous page)

```
    </ParameterTypeSet>
    <ParameterSet>
      <Parameter parameterTypeRef="float_t" name="testparam">
        <ParameterProperties dataSource="local" persistence="false"/>
      </Parameter>
    </ParameterSet>
  </TelemetryMetaData>
</SpaceSystem>
```

Index

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z