Yamcs Python Client Documentation

Release 1.5.3

Space Applications Services

Aug 19, 2020

CONTENTS

CHAPTER

ONE

GETTING STARTED

Install Python 3.5 or higher.

Install `yamcs-client` from PyPI:

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

CHAPTER

TWO

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')
# ...
```

2.1 Documentation

- *General Client*
- *Mission Database*
- *TM/TC Processing*
- *Archive*
- *Link Management*
- *Object Storage*
- *CFDP*

2.2 Examples

- *Examples*

2.2.1 General Client

Reference

YamcsClient

class yamcs.client.**YamcsClient**(*address*, ***kwargs*)
Bases: yamcs.core.client.BaseClient

Client for accessing core Yamcs resources.

The only state managed by this client is its connection info to Yamcs.

Parameters

- **address** (*str*) – The address of Yamcs in the format 'hostname:port'
- **tls** (*Optional[bool]*) – Whether TLS encryption is expected
- **tls_verify** (*Optional[bool]*) – Whether server certificate verification is enabled (only applicable if tls=True)
- **credentials** (*Optional[Credentials]*) – Credentials for when the server is secured
- **user_agent** (*Optional[str]*) – Optionally override the default user agent

create_data_link_subscription(*instance*, *on_data=None*, *timeout=60*)
Deprecated. Rename to create_link_subscription.

create_event_subscription(*instance*, *on_data*, *timeout=60*)
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** (*str*) – A Yamcs instance name
- **on_data** (*Optional[Callable[Event]]*) – Function that gets called on each *Event*.
- **timeout** (*Optional[float]*) – 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 *WebSocketSubscriptionFuture*

create_instance(*name*, *template*, *args=None*, *labels=None*)
Create a new instance based on an existing template. This method blocks until the instance is fully started.

Parameters

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

create_link_subscription(*instance*, *on_data=None*, *timeout=60*)
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** (*str*) – A Yamcs instance name.

- **on_data** (*Optional[Callable[LinkEvent]]*) – Function that gets called with *LinkEvent* updates.
- **timeout** (*Optional[float]*) – 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 *LinkSubscription*

create_time_subscription(*instance*, *on_data=None*, *timeout=60*)
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** (*str*) – A Yamcs instance name
- **on_data** (*Optional[Callable[datetime]]*) – Function that gets called with `datetime` updates.
- **timeout** (*Optional[float]j*) – 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 *TimeSubscription*

disable_data_link(*instance*, *link*)
Deprecated. Use `get_link(instance, link).disable_link()`.

enable_data_link(*instance*, *link*)
Deprecated. Use `get_link(instance, link).enable_link()`.

get_archive(*instance*)
Return an object for working with the Archive of the specified instance.

Parameters **instance** (*str*) – A Yamcs instance name.

Return type *ArchiveClient*

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

Return type *AuthInfo*

get_cfdp_client(*instance*)
Return an object for working with CFDP transfers on a specified instance.

Parameters **instance** (*str*) – A Yamcs instance name.

Return type *CFDPClient*

get_data_link(*instance*, *link*)
Deprecated. Use `get_link(instance, link).get_info()`.

get_link(*instance*, *link*)
Return an object for working with a specific Yamcs link.

Parameters

- **instance** (*str*) – A Yamcs instance name.
- **link** (*str*) – A link name within that instance.

Return type *LinkClient*

get_mdb(*instance*)
Return an object for working with the MDB of the specified instance.

Parameters **instance** (*str*) – A Yamcs instance name.

Return type *MDBClient*

get_processor(*instance*, *processor*)
Return an object for working with a specific Yamcs processor.

Parameters

- **instance** (*str*) – A Yamcs instance name.
- **processor** (*str*) – A processor name within that instance.

Return type *ProcessorClient*

get_server_info()
Return general server info.

Return type *ServerInfo*

get_storage_client(*instance='_global'*)
Return an object for working with object storage

Parameters **instance** (*str*) – The storage instance.

Return type *StorageClient*

get_time(*instance*)
Return the current mission time for the specified instance.

Return type datetime

get_user_info()
Get information on the authenticated user.

Return type *UserInfo*

list_data_links(*instance*)
Lists the data links visible to this client.

Data links are returned in random order.

Parameters **instance** (*str*) – A Yamcs instance name.

Return type Iterable[*Link*]

list_instance_templates()
List the available instance templates.

list_instances()
Lists the instances.

Instances are returned in lexicographical order.

Return type Iterable[*Instance*]

list_processors(*instance=None*)
Lists the processors.

Processors are returned in lexicographical order.

Parameters **instance** (*Optional[str]*) – A Yamcs instance name.

Return type Iterable[*Processor*]

list_services(*instance*)
List the services for an instance.

Parameters **instance** (*str*) – A Yamcs instance name.

Return type Iterable[*Service*]

restart_instance(*instance*)
Restarts a single instance.

Parameters **instance** (*str*) – A Yamcs instance name.

send_event(*instance*, *message*, *event_type=None*, *time=None*, *severity='info'*, *source=None*, *sequence_number=None*)
Post a new event.

Parameters

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

start_instance(*instance*)
Starts a single instance.

Parameters **instance** (*str*) – A Yamcs instance name.

start_service(*instance*, *service*)
Starts a single service.

Parameters

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

stop_instance(*instance*)
Stops a single instance.

Parameters **instance** (*str*) – A Yamcs instance name.

stop_service(*instance*, *service*)
Stops a single service.

Parameters

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

class yamcs.client.**TimeSubscription**(*manager*)
Bases: *yamcs.core.futures.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 = None
The last time info.

Type datetime

class yamcs.client.**LinkSubscription**(*manager*)
Bases: *yamcs.core.futures.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_data_link(*name*)
Returns the latest link state.

Parameters **name** (*str*) – Identifying name of the data link

Return type *Link*

list_data_links()
Returns a snapshot of all instance links.

Return type *Link*[]

Model

class yamcs.model.**AuthInfo**(*proto*)
Bases: object

Authentication information

property require_authentication

class yamcs.model.**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
The event type. This is mission-specific and can be any string.

property generation_time
The time when the event was generated.

Type datetime

property message
Event message.

property reception_time
The time when the event was received by Yamcs.

Type datetime

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

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

property source
The event source. Can be any string.

class yamcs.model.**Instance**(*proto*)
Bases: `object`

property failure_cause
Failure message when `state == 'FAILED'`

property mission_time
Mission time of this instance's time service.

Type `datetime`

property name
Name of this instance.

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

class yamcs.model.**InstanceTemplate**(*proto*)
Bases: `object`

A template for creating an instance.

property name
Name of this template.

class yamcs.model.**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 class_name
Name of this link's class.

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

property in_count
The number of inbound data events (example: packet count).

property instance
Name of the instance where this link is defined.

property name
Name of this link (unique per instance).

property out_count
The number of outbound data events (example: command count).

property status
Short status.

class yamcs.model.**LinkEvent**(*proto*)
Bases: `object`

Data holder used in link subscriptions.

property event_type
The type of the event. One of REGISTERED, UNREGISTERED, or UPDATED.

property link
Link state at the time of this event.

Type *Link*

class yamcs.model.**ObjectPrivilege**(*proto*)
Bases: object

property name

property objects

class yamcs.model.**Processor**(*proto*)
Bases: object

property instance
Name of the instance where this processor is defined.

property mission_time
Mission time of this processor.

Type datetime

property name
Name of this processor.

property owner
User that owns this processor.

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

property state
State of this processor.

property type
Type of this processor.

class yamcs.model.**ServerInfo**(*proto*)
Bases: object

General server properties.

property default_yamcs_instance
Returns the default Yamcs instance.

property id
The Server ID.

property version
The version of Yamcs Server.

class yamcs.model.**Service**(*proto*)
Bases: object

A Yamcs service.

property class_name
Name of this service's class.

property instance
Name of the instance where this service is defined.

property name
Name of this service.

property processor
Name of the processor where this service is defined.

property state
State of this service.

class yamcs.model.**UserInfo**(*proto*)
Bases: object

Info on a Yamcs User.

property object_privileges

property superuser

property system_privileges

property username

Authentication

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 simply 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.

Service Accounts

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

Types

class yamcs.core.auth.**Credentials**(*username=None*, *password=None*, *access_token=None*, *refresh_token=None*, *expiry=None*, *client_id=None*, *client_secret=None*, *become=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 = None
Short-lived bearer token.

become = None
Name of the user to impersonate. Only service accounts with impersonation authority can use this feature.

before_request(*session*, *auth_url*)

client_id = None
The client ID. Currently used only by service accounts.

client_secret = None
The client secret. Currently used only by service accounts.

expiry = None
When this token expires.

is_expired()

login(*session*, *auth_url*, *on_token_update*)

password = None
Clear-text password (consider TLS!).

refresh(*session*, *auth_url*)

refresh_token = None
Refresh token used to request a new access token.

username = None
Username (only needed when using username/password credentials).

Exceptions

exception yamcs.core.exceptions.**ConnectionFailure**
Bases: *yamcs.core.exceptions.YamcsError*

Yamcs is not or no longer available.

exception yamcs.core.exceptions.**NotFound**
Bases: *yamcs.core.exceptions.YamcsError*

The resource was not found.

exception yamcs.core.exceptions.**TimeoutError**
Bases: *yamcs.core.exceptions.YamcsError*

The operation exceeded the given deadline.

exception yamcs.core.exceptions.**Unauthorized**
Bases: *yamcs.core.exceptions.YamcsError*

Unable to get access the resource.

exception yamcs.core.exceptions.**YamcsError**
Bases: Exception

Base class for raised exceptions.

Futures

class yamcs.core.futures.**WebSocketSubscriptionFuture**(*manager*)
Bases: concurrent.futures._base.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.

Args:

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 of the future was cancelled or finished executing.

exception(*timeout=None*)
Return the exception raised by the call that the future represents.

Args:

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.

Args:

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.

Snippets

Create a *YamcsClient*:

```
from yamcs.client import YamcsClient

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

Provide credentials if Yamcs is secured:

```
from yamcs.client import YamcsClient
from yamcs.core.auth import Credentials

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

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')
```

2.2.2 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`.

Reference

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.mdb.client.**MDBClient**(*client*, *instance*)

get_algorithm(*name*)
Gets a single algorithm by its unique name.

Parameters **name** (*str*) – Either a fully-qualified XTCE name or an alias in the format NAMESPACE/NAME.

Return type *Algorithm*

get_command(*name*)
Gets a single command by its unique name.

Parameters **name** (*str*) – Either a fully-qualified XTCE name or an alias in the format NAMESPACE/NAME.

Return type *Command*

get_container(*name*)
Gets a single container by its unique name.

Parameters **name** (*str*) – Either a fully-qualified XTCE name or an alias in the format NAMESPACE/NAME.

Return type *Container*

get_parameter(*name*)
Gets a single parameter by its name.

Parameters **name** (*str*) – Either a fully-qualified XTCE name or an alias in the format NAMESPACE/NAME.

Return type *Parameter*

get_space_system(*name*)
Gets a single space system by its unique name.

Parameters **name** (*str*) – A fully-qualified XTCE name

Return type *SpaceSystem*

list_algorithms(*page_size=None*)
Lists the algorithms visible to this client.

Algorithms are returned in lexicographical order.

Return type *Algorithm* iterator

list_commands(*page_size=None*)
Lists the commands visible to this client.

Commands are returned in lexicographical order.

Return type *Command* iterator

list_containers(*page_size=None*)
Lists the containers visible to this client.

Containers are returned in lexicographical order.

Return type *Container* iterator

list_parameters(*parameter_type=None*, *page_size=None*)
Lists the parameters visible to this client.

Parameters are returned in lexicographical order.

Parameters **parameter_type** (*str*) – The type of parameter

Return type *Parameter* iterator

list_space_systems(*page_size=None*)
Lists the space systems visible to this client.

Space systems are returned in lexicographical order.

Return type *SpaceSystem* iterator

Model

class yamcs.mdb.model.**Algorithm**(*proto*)

property aliases
List of (namespace, name) pairs, as 2-tuples

property description
Short description.

property long_description
Long description.

property name
Short name

property qualified_name
Full name (incl. space system)

class yamcs.mdb.model.**ArrayType**(*proto*)

property arrayType
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.

Type *ArrayType*

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

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

Type List[*Member*]

property name
Short name of this type.

class yamcs.mdb.model.**Command**(*proto*)

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

property aliases
List of (namespace, name) pairs, as 2-tuples

property base_command

property description
Short description.

property long_description
Long description.

property name
Short name

property qualified_name
Full name (incl. space system)

property significance

class yamcs.mdb.model.**Container**(*proto*)

property aliases
List of (namespace, name) pairs, as 2-tuples

property description
Short description.

property long_description
Long description.

property name
Short name

property qualified_name
Full name (incl. space system)

class yamcs.mdb.model.**Member**(*proto*)
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 arrayType
In case this member is of type *array*, this returns array-specific type info.

Type *ArrayType*

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

Type List[*Member*]

property name
Short name

property type
Engineering type.

Type str

class yamcs.mdb.model.**Parameter**(*proto*)
From XTCE:

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

property aliases
List of (namespace, name) pairs, as 2-tuples

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

Type *ArrayType*

property data_source
Specifies how this parameter originated (example: `TELEMETERED`)

Type str

property description
Short description.

property long_description
Long description.

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

Type List[*Member*]

property name
Short name

property qualified_name
Full name (incl. space system)

property type
Engineering type.

Type str

class yamcs.mdb.model.**Significance**(*proto*)

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

property reason
Message attached to this significance.

class yamcs.mdb.model.**SpaceSystem**(*proto*)
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.

property aliases
List of (namespace, name) pairs, as 2-tuples

property description
Short description.

property long_description
Long description.

property name
Short name

property qualified_name
Full name (incl. space system)

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)
```

2.2.3 TM/TC Processing

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

Reference

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.tmtc.client.**ProcessorClient**(*client*, *instance*, *processor*)

Client object that groups operations linked to a specific processor.

acknowledge_alarm(*alarm*, *comment=None*)
Acknowledges a specific alarm.

Parameters

- **alarm** (*Alarm*) – Alarm instance
- **comment** (*str*) – Optional comment to associate with the state change.

clear_alarm(*alarm*, *comment=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*) – Alarm instance
- **comment** (*str*) – Optional comment to associate with the state change.

clear_alarm_ranges(*parameter*)
Removes all alarm limits for the specified parameter.

clear_calibrators(*parameter*)
Removes all calibrators for the specified parameter.

create_alarm_subscription(*on_data=None*, *timeout=60*)
Create a new alarm subscription.

Parameters

- **on_data** – (Optional) Function that gets called with *AlarmUpdate* updates.
- **timeout** (*float*) – The amount of seconds to wait for the request to complete.

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

Return type *AlarmSubscription*

create_command_connection(*on_data=None*, *timeout=60*)
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** (*float*) – 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 *CommandConnection*

create_command_history_subscription(*on_data=None*, *timeout=60*)
Create a new command history subscription.

Parameters

- **on_data** – (Optional) Function that gets called with *CommandHistory* updates.
- **timeout** (*float*) – 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 *CommandHistorySubscription*

create_parameter_subscription(*parameters*, *on_data=None*, *abort_on_invalid=True*, *update_on_expiration=False*, *send_from_cache=True*, *timeout=60*)

Create a new parameter subscription.

Parameters

- **parameters** (*str[]*) – Parameter names (or aliases).
- **on_data** – (Optional) Function that gets called with *ParameterData* updates.
- **abort_on_invalid** (*bool*) – If `True` an error is generated when invalid parameters are specified.
- **update_on_expiration** (*bool*) – 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** (*bool*) – If `True` the last processed parameter value is sent from parameter cache. When `False` only newly processed parameters are received.
- **timeout** (*float*) – The amount of seconds to wait for the request to complete.

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

Return type *ParameterSubscription*

get_parameter_value(*parameter*, *from_cache=True*, *timeout=10*)

Retrieve the current value of the specified parameter.

Parameters

- **parameter** (*str*) – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **from_cache** (*bool*) – 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** (*float*) – The amount of seconds to wait for a fresh value. (ignored if `from_cache=True`).

Return type *ParameterValue*

get_parameter_values(*parameters*, *from_cache=True*, *timeout=10*)

Retrieve the current value of the specified parameter.

Parameters

- **parameters** (*str[]*) – List of parameter names. These may be fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **from_cache** (*bool*) – 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** (*float*) – 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`.

Return type *ParameterValue*[]

issue_command(*command*, *args=None*, *dry_run=False*, *comment=None*, *verification=None*, *extra=None*)

Issue the given command

Parameters

- **command** (*str*) – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **args** (*dict*) – named arguments (if the command requires these)
- **dry_run** (*bool*) – 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** (*str*) – Comment attached to the command.
- **verification** (`VerificationConfig`) – Overrides to the default verification handling of this command.
- **extra** (*dict*) – 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.

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

Return type *IssuedCommand*

list_alarms(*start=None*, *stop=None*)

Lists the active alarms.

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

Parameters

- **start** (*datetime*) – Minimum trigger time of the returned alarms (inclusive)
- **stop** (*datetime*) – Maximum trigger time of the returned alarms (exclusive)

Return type Iterable[*Alarm*]

reset_alarm_ranges(*parameter*)

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

reset_algorithm(*parameter*)

Reset the algorithm text to its original definition from MDB

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

reset_calibrators(*parameter*)

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

set_alarm_range_sets(*parameter*, *sets*)

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

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

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

set_algorithm(*parameter*, *text*)
Change an algorithm text. Can only be peformed on JavaScript or Python algorithms.

Parameters

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

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

set_default_alarm_ranges(*parameter*, *watch=None*, *warning=None*, *distress=None*, *critical=None*, *severe=None*, *min_violations=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** (*str*) – Either a fully-qualified XTCE name or an alias in the format NAMESPACE/NAME.
- **watch** ((*float*,*float*)) – Range expressed as a tuple (lo, hi) where lo and hi are assumed exclusive.
- **warning** ((*float*,*float*)) – Range expressed as a tuple (lo, hi) where lo and hi are assumed exclusive.
- **distress** ((*float*,*float*)) – Range expressed as a tuple (lo, hi) where lo and hi are assumed exclusive.
- **critical** ((*float*,*float*)) – Range expressed as a tuple (lo, hi) where lo and hi are assumed exclusive.
- **severe** ((*float*,*float*)) – Range expressed as a tuple (lo, hi) where lo and hi are assumed exclusive.
- **min_violations** (*int*) – Minimum violations before an alarm is generated.

set_default_calibrator(*parameter*, *type*, *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: $y = a + bx + cx^2 + \ldots$.

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

set_parameter_value(*parameter*, *value*)
Sets the value of the specified parameter.

Parameters

- **parameter** (*str*) – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **value** – The value to set

set_parameter_values(*values*)
Sets the value of multiple parameters.

Parameters **values** (*dict*) – Values keyed by parameter name. This name can be either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.

shelve_alarm(*alarm*, *comment=None*)
Shelve an alarm.

Parameters

- **alarm** (*Alarm*) – Alarm instance
- **comment** (*str*) – Optional comment to associate with the state change.

unshelve_alarm(*alarm*, *comment=None*)
Unshelve an alarm.

Parameters

- **alarm** (*Alarm*) – Alarm instance
- **comment** (*str*) – Optional comment to associate with the state change.

class yamcs.tmtc.client.**CommandConnection**(*manager*, *client*)
Bases: *yamcs.core.futures.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*, *args=None*, *dry_run=False*, *comment=None*, *verification=None*, *extra=None*)
Issue the given command

Parameters

- **command** (*str*) – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **args** (*dict*) – Named arguments (if the command requires these)
- **dry_run** (*bool*) – 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** (*str*) – Comment attached to the command.
- **verification** (`VerificationConfig`) – Overrides to the default verification handling of this command.
- **extra** (*dict*) – 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.

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

Return type *MonitoredCommand*

class yamcs.tmtc.client.**AlarmSubscription**(*manager*)
Bases: *yamcs.core.futures.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*)
Returns the alarm state associated with a specific named alarm from local cache.

Parameters **name** (*str*) – Fully-qualified name

Return type *Alarm*

list_alarms()
Returns a snapshot of all active alarms.

Return type *Alarm*[]

class yamcs.tmtc.client.**CommandHistorySubscription**(*manager*)
Bases: *yamcs.core.futures.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*)
Gets locally cached CommandHistory for the specified command.

Parameters **issued_command** (IssuedCommand) – object representing a previously issued command.

Return type *CommandHistory*

class yamcs.tmtc.client.**ParameterSubscription**(*manager*)
Bases: *yamcs.core.futures.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*, *abort_on_invalid=True*, *send_from_cache=True*)
Add one or more parameters to this subscription.

Parameters

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

delivery_count = None
The number of parameter deliveries.

get_value(*parameter*)
Returns the last value of a specific parameter from local cache.

Return type *ParameterValue*

remove(*parameters*)
Remove one or more parameters from this subscription.

Parameters **parameters** (*Union[str, str[]]*) – Parameter(s) to be removed

value_cache = None
Value cache keyed by parameter name.

Model

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

is_terminated()

message = None
Message of this acknowledgment.

name = None
Name of this acknowledgment.

status = None
Status of this acknowlegment.

time = None
Last update time of this acknowledgment.

class yamcs.tmtc.model.**Alarm**(*proto*)

property acknowledge_message
Comment provided when acknowledging the alarm.

property acknowledge_time
Processor time when the alarm was acknowledged.

Type datetime

property acknowledged_by
Username of the acknowledger.

property count
Total number of samples while this alarm is active.

property is_acknowledged
True if this alarm has been acknowledged.

property is_latched
True if this alarm is currently latched.

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

property is_ok
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
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
True if this alarm has been shelved.

property name
Fully-qualified name of the source of this alarm.

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

property severity

property trigger_time
Processor time when the alarm was triggered.

Type datetime

property violation_count
Number of violating samples while this alarm is active.

class yamcs.tmtc.model.**AlarmUpdate**(*proto*)
Object received through callbacks when subscribing to alarm updates.

property alarm
Latest alarm state.

Type *Alarm*

property update_type
Type of update.

class yamcs.tmtc.model.**Calibrator**(*context*, *type*, *data*)
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:** $y = a + bx + cx^2 + \ldots$.

  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** (*str*) – 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** (*str*) – One of `polynomial` or `spline`.
- **data** – Calibration definition for the selected type.

POLYNOMIAL = 'polynomial'

SPLINE = 'spline'

class yamcs.tmtc.model.**CommandHistory**(*proto*)

property acknowledgments
All acknowledgments by name.

Returns Acknowledgments keyed by name.

Return type OrderedDict

property binary
Binary representation of the command.

property comment
Optional user comment attached when issuing the command.

property error
Error message in case the command failed.

generation_time = None
The generation time as set by Yamcs

Type datetime

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

is_failure()
Returns True if the command failed.

is_success()
Returns True if the command has completed successfully.

property name
Name of the command.

property origin
The origin of this command. This is often empty, but may also be a hostname.

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

property source
String representation of the command.

property username
Username of the issuer.

class yamcs.tmtc.model.**EventAlarm**(*proto*)
An alarm triggered by an event.

property current_event
Latest event for this alarm

Type *Event*

property most_severe_event
First event that invoked the highest severity level of this alarm

Type *Event*

property trigger_event
Event that originally triggered the alarm

Type *Event*

class yamcs.tmtc.model.**IssuedCommand**(*proto*, *client*)

property binary
Binary representation of this command.

property generation_time
The generation time as set by Yamcs.

Type datetime

property hex
Hexadecimal string representation of this command.

property id
A unique identifier for this command.

property name
The fully-qualified name of this command.

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

property queue
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 username
The username of the issuer.

class yamcs.tmtc.model.**MonitoredCommand**(*proto*, *client*)
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
All acknowledgments by name.

Returns Acknowledgments keyed by name.

Return type OrderedDict

property attributes

await_acknowledgment(*name*, *timeout=None*)
Waits for the result of a specific acknowledgment.

Parameters

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

Return type *Acknowledgment*

await_complete(*timeout=None*)
Wait for the command to be completed.

Parameters **timeout** (*float*) – The amount of seconds to wait.

property comment
Optional user comment attached when issuing the command.

property error
Error message in case the command failed.

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

is_success()
Returns true if this command was completed successfully.

class yamcs.tmtc.model.**ParameterAlarm**(*proto*)
An alarm triggered by a parameter that went out of limits.

property current_value
Latest parameter value for this alarm.

Type *ParameterValue*

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

Type *ParameterValue*

property trigger_value
Parameter value that originally triggered the alarm

Type *ParameterValue*

class yamcs.tmtc.model.**ParameterData**(*proto*, *mapping*)

property parameters

Type List[*ParameterValue*]

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

property eng_value
The engineering (calibrated) value.

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

Type datetime

property monitoring_result

property name
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 processing_status

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

property raw_value
The raw (uncalibrated) value.

property reception_time
The time when the parameter value was received by Yamcs.

Type datetime

property validity_duration
How long this parameter value is valid.

Type timedelta

property validity_status

class yamcs.tmtc.model.**RangeSet**(*context*, *watch=None*, *warning=None*, *distress=None*, *critical=None*, *severe=None*, *min_violations=1*)
A set of alarm range that apply in a specific context.

Parameters

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

class yamcs.tmtc.model.**VerificationConfig**

Contains overrides to the default verification handling of Yamcs.

disable(*verifier=None*)

Disable verification.

Parameters **verifier** (*str*) – Name of a specific verifier to disable. If unspecified all verifiers are disabled.

modify_check_window(*verifier*, *start=None*, *stop=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** (*str*) – Name of the verifier
- **start** (*float*) – Window start time (relative, in seconds)
- **stop** (*float*) – Window stop time (relative, in seconds)

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')
```

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,
})
```

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)
```

(continues on next page)

(continued from previous page)

```
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'))
```

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)
```

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, comment='false alarm')
```

2.2.4 Archive

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

Reference

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.archive.client.**ArchiveClient**(*client*, *instance*)

create_stream_subscription(*stream*, *on_data*, *timeout=60*)
Create a new stream subscription.

Parameters

- **stream** (*str*) – The name of the stream.
- **on_data** – Function that gets called with *StreamData* updates.
- **timeout** (*float*) – 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 *WebSocketSubscriptionFuture*

dump_table(*table*, *chunk_size=1024*)

execute_sql(*statement*)
Executes a single SQL statement.

Parameters **statement** – SQL string

Returns A result set for consuming rows

Return type *ResultSet*

export_packets(*name=None*, *start=None*, *stop=None*, *chunk_size=1024*)
Export raw packets.

Packets are sorted by generation time and sequence number.

Parameters

- **name** (*str*) – Archived name of the packet
- **start** (*datetime*) – Minimum generation time of the returned packets (inclusive)
- **stop** (*datetime*) – Maximum generation time of the returned packets (exclusive)

Return type An iterator over received chunks

get_packet(*generation_time*, *sequence_number*)
Gets a single packet by its identifying key (gentime, seqNum).

Parameters

- **generation_time** (*datetime*) – When the packet was generated (packet time)
- **sequence_number** (*int*) – Sequence number of the packet

Return type *Packet*

get_stream(*stream*)
Gets a single stream.

Parameters **stream** (*str*) – The name of the stream.

Return type *Stream*

get_table(*table*)
Gets a single table.

Parameters **table** (*str*) – The name of the table.

Return type *Table*

list_command_histogram(*name=None*, *start=None*, *stop=None*, *merge_time=2*)
Reads command-related index records between the specified start and stop time.

Each iteration returns a chunk of chronologically-sorted records.

Parameters **merge_time** (*float*) – Maximum gap in seconds before two consecutive index records are merged together.

Return type Iterable[*IndexGroup*]

list_command_history(*command=None*, *start=None*, *stop=None*, *page_size=500*, *descending=False*)
Reads command history entries between the specified start and stop time.

Parameters

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

Return type Iterable[*CommandHistory*]

list_completeness_index(*start=None*, *stop=None*)
Reads completeness index records between the specified start and stop time.

Each iteration returns a chunk of chronologically-sorted records.

Return type Iterable[*IndexGroup*]

list_event_histogram(*source=None*, *start=None*, *stop=None*, *merge_time=2*)
Reads event-related index records between the specified start and stop time.

Each iteration returns a chunk of chronologically-sorted records.

Parameters **merge_time** (*float*) – Maximum gap in seconds before two consecutive index records are merged together.

Return type Iterable[*IndexGroup*]

list_event_sources()
Returns the existing event sources.

Return type Iterable[str]

list_events(*source=None*, *severity=None*, *text_filter=None*, *start=None*, *stop=None*, *page_size=500*, *descending=False*)
Reads events between the specified start and stop time.

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

Parameters

- **source** (*str*) – The source of the returned events.
- **severity** (*str*) – The minimum severity level of the returned events. One of `INFO`, `WATCH`, `WARNING`, `DISTRESS`, `CRITICAL` or `SEVERE`.
- **text_filter** (*str*) – Filter the text message of the returned events
- **start** (*datetime*) – Minimum start date of the returned events (inclusive)
- **stop** (*datetime*) – Maximum start date of the returned events (exclusive)
- **page_size** (*int*) – Page size of underlying requests. Higher values imply less overhead, but risk hitting the maximum message size limit.
- **descending** (*bool*) – If set to `True` events are fetched in reverse order (most recent first).

Return type Iterable[*Event*]

list_packet_histogram(*name=None*, *start=None*, *stop=None*, *merge_time=2*)
Reads packet-related index records between the specified start and stop time.

Each iteration returns a chunk of chronologically-sorted records.

Parameters **merge_time** (*float*) – Maximum gap in seconds before two consecutive index records are merged together.

Return type Iterable[*IndexGroup*]

list_packet_names()
Returns the existing packet names.

Return type Iterable[str]

list_packets(*name=None*, *start=None*, *stop=None*, *page_size=500*, *descending=False*)
Reads packet information between the specified start and stop time.

Packets are sorted by generation time and sequence number.

Parameters

- **name** (*str*) – Archived name of the packet
- **start** (*datetime*) – Minimum generation time of the returned packets (inclusive)
- **stop** (*datetime*) – Maximum generation time of the returned packets (exclusive)
- **page_size** (*int*) – Page size of underlying requests. Higher values imply less overhead, but risk hitting the maximum message size limit.
- **descending** (*bool*) – If set to `True` packets are fetched in reverse order (most recent first).

Return type Iterable[*Packet*]

list_parameter_ranges(*parameter*, *start=None*, *stop=None*, *min_gap=None*, *max_gap=None*, *parameter_cache='realtime'*)

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** (*str*) – Either a fully-qualified XTCE name or an alias in the format NAMESPACE/NAME.
- **start** (*datetime*) – Minimum generation time of the considered values (inclusive)
- **stop** (*datetime*) – Maximum generation time of the considered values (exclusive)
- **min_gap** (*float*) – 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** (*float*) – 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.
- **parameter_cache** (*str*) – 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.

Return type *ParameterRange*[]

list_parameter_values(*parameter*, *start=None*, *stop=None*, *page_size=500*, *descending=False*, *parameter_cache='realtime'*, *source='ParameterArchive'*)

Reads parameter values between the specified start and stop time.

Parameters

- **parameter** (*str*) – Either a fully-qualified XTCE name or an alias in the format NAMESPACE/NAME.
- **start** (*datetime*) – Minimum generation time of the returned values (inclusive)
- **stop** (*datetime*) – Maximum generation time of the returned values (exclusive)
- **page_size** (*int*) – Page size of underlying requests. Higher values imply less overhead, but risk hitting the maximum message size limit.
- **descending** (*bool*) – If set to True values are fetched in reverse order (most recent first).
- **parameter_cache** (*str*) – 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** (*str*) – 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.

Return type Iterable[*ParameterValue*]

list_processed_parameter_group_histogram(*group=None*, *start=None*, *stop=None*, *merge_time=20*)

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** (*float*) – Maximum gap in seconds before two consecutive index records are merged together.

Return type Iterable[*IndexGroup*]

list_processed_parameter_groups()

Returns the existing parameter groups.

Return type Iterable[str]

list_streams()

Returns the existing streams.

Streams are returned in lexicographical order.

Return type Iterable[*Stream*]

list_tables()

Returns the existing tables.

Tables are returned in lexicographical order.

Return type Iterable[*Table*]

load_table(*table*, *data*)

sample_parameter_values(*parameter*, *start=None*, *stop=None*, *sample_count=500*, *parameter_cache='realtime'*, *source='ParameterArchive'*)

Returns parameter samples.

The query range is split in sample intervals of equal length. For each interval a *Sample* 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** (*str*) – Either a fully-qualified XTCE name or an alias in the format `NAMESPACE/NAME`.
- **start** (*datetime*) – Minimum generation time of the sampled parameter values (inclusive). If not set this defaults to one hour ago.
- **stop** (*datetime*) – Maximum generation time of the sampled parameter values (exclusive). If not set this defaults to the current time.
- **sample_count** (*int*) – The number of returned samples.
- **parameter_cache** (*str*) – 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** (*str*) – 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.

Return type *Sample*[]

Model

class yamcs.archive.model.**ColumnData**(*proto*)

property name
Column name.

property value
Value for this column.

class yamcs.archive.model.**IndexGroup**(*proto*)
Group of index records that represent the same type of underlying objects.

property name
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
Index records within this group

Type List[*IndexRecord*]

class yamcs.archive.model.**IndexRecord**(*proto*)
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
Number of underlying objects this index record represents

property start
Start time of the record

Type `datetime`

property stop
Stop time of the record

Type `datetime`

class yamcs.archive.model.**Packet**(*proto*)

property binary
Raw binary of this packet

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

Type `datetime`

property name
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
The time when the packet was received by Yamcs.

Type datetime

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

class yamcs.archive.model.**ParameterRange**(*proto*)
Indicates an interval during which a parameter's value was uninterrupted and unchanged.

property eng_value
The engineering (calibrated) value.

property parameter_count
The number of received parameter values during this range.

property start
Start time of this range (inclusive).

Type datetime

property stop
Stop time of this range (exclusive).

Type datetime

class yamcs.archive.model.**ResultSet**(*response*)
Provides capability to consume the rows returned by a SQL query, or access related information.

property column_types
Column types.

Type str[]

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

Type str[]

class yamcs.archive.model.**Sample**(*proto*)
Provides aggregation properties over a range of a parameter's values.

property avg
Average value.

property max
Maximum value.

property min
Minimum value.

property parameter_count
The number of parameter values this sample represents.

property time
Sample time.

Type datetime

class yamcs.archive.model.**Stream**(*proto*)

property name
Stream name.

class yamcs.archive.model.**StreamData**(*proto*)

property columns
Tuple columns.

property stream
Stream name.

class yamcs.archive.model.**Table**(*proto*)

property name
Table name.

Snippets

Create an *ArchiveClient* for a specific instance:

```
from yamcs.client import YamcsClient

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

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.utcnow()
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.utcnow()
start = now - timedelta(hours=1)

with open('/tmp/dump.raw', 'wb') as f:
```

(continues on next page)

(continued from previous page)

```
    for chunk in archive.export_packets(start=start, stop=now):
        f.write(chunk)
```

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.utcnow()
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')
```

Event Retrieval

Iterate a specific range of events:

```
now = datetime.utcnow()
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')
```

Command Retrieval

Retrieve the last 10 issued commands:

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

Histogram Retrieval

Print the number of packets grouped by packet name:

```
for name in archive.list_packet_names():
    if not name == '/YSS/SIMULATOR/DHS':
        continue
    packet_count = 0
    for group in archive.list_packet_histogram(name):
        for rec in group.records:
            packet_count += rec.count
    print('  {: <40} {: >20}'.format(name, packet_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('  {: <40} {: >20}'.format(source, event_count))
```

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('  {: <40} {: >20}'.format(group, frame_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('  {: <40} {: >20}'.format(command.qualified_name, total))
```

2.2.5 Link Management

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

Reference

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.link.client.**LinkClient**(*client*, *instance*, *link*)

Client object that groups operations for a specific link.

create_cop1_subscription(*on_data*, *timeout=60*)

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** (*Optional[Callable[Cop1Status]]*) – Function that gets called on each *Cop1Status*.
- **timeout** (*Optional[float]*) – 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 *Cop1Subscription*

disable_cop1(*bypass_all=True*)

Disable COP1 for a data link.

Parameters **bypass_all** (*bool*) – 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()

Gets the COP1 configuration for a data link.

Return type *Cop1Config*

get_cop1_status()

Retrieve the COP1 status.

Return type *Cop1Status*

get_info()
Get info on this link.

Return type *Link*

initialize_cop1(*type*, *clcw_wait_timeout=None*, *v_r=None*)
Initialize COP1.

Parameters

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

resume_cop1()
Resume COP1.

update_cop1_config(*window_width=None*, *timeout_type=None*, *tx_limit=None*, *t1=None*)
Sets the COP1 configuration for a data link.

Return type *Cop1Config*

class yamcs.link.client.**Cop1Subscription**(*manager*)
Bases: *yamcs.core.futures.WebSocketSubscriptionFuture*

Local object providing access to COP1 status updates.

Initializes the future. Should not be called by clients.

property bypass_all

property cop1_active

get_status()
Returns the latest known COP1 status.

Return type *Cop1Status*

property nn_r

property state

property v_s

Model

class yamcs.link.model.**Cop1Config**(*proto*)
COP1 configuration

property t1

property timeout_type

property tx_limit

property vc_id
Virtual Channel ID.

property window_width

class yamcs.link.model.**Cop1Status**(*proto*)
COP1 status

property bypass_all

property cop1_active

property nn_r

property state

property v_s

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()
for link in client.list_data_links(instance='simulator'):
    client.enable_data_link(instance=link.instance, link=link.name)
```

2.2.6 Object Storage

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

Reference

Client

class yamcs.storage.**StorageClient**(*client*, *instance='_global'*)
Client for working with buckets and objects managed by Yamcs.

create_bucket(*bucket_name*)
Create a new bucket.

Parameters **bucket_name** (*str*) – The name of the bucket.

download_object(*bucket_name*, *object_name*)
Download an object.

Parameters

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

get_bucket(*name*)
Get a specific bucket.

Parameters **name** (*str*) – The bucket name.

Return type *Bucket*

list_buckets()
List the buckets.

Return type Iterable[*Bucket*]

list_objects(*bucket_name*, *prefix=None*, *delimiter=None*)
List the objects for a bucket.

Parameters

- **bucket_name** (*str*) – The name of the bucket.
- **prefix** (*str*) – If specified, only objects that start with this prefix are listed.
- **delimiter** (*str*) – 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*)
Remove a bucket.

Parameters **bucket_name** (*str*) – The name of the bucket.

remove_object(*bucket_name*, *object_name*)
Remove an object from a bucket.

Parameters

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

upload_object(*bucket_name*, *object_name*, *file_obj*, *content_type=None*)
Upload an object to a bucket.

Parameters

- **bucket_name** (*str*) – The name of the bucket.
- **object_name** (*str*) – The target name of the object.
- **file_obj** (*file*) – The file (or file-like object) to upload.
- **content_type** (*str*) – 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`.

Model

class yamcs.storage.model.**Bucket**(*proto*, *client*)

delete()
Remove this bucket in its entirety.

delete_object(*object_name*)
Remove an object from this bucket.

Parameters **object_name** (*str*) – The object to remove.

download_object(*object_name*)
Download an object.

Parameters **object_name** (*str*) – The object to fetch.

list_objects(*prefix=None*, *delimiter=None*)
List the objects for this bucket.

Parameters

- **prefix** (*str*) – If specified, only objects that start with this prefix are listed.
- **delimiter** (*str*) – 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 name
Name of this bucket.

property object_count
Number of objects in this bucket.

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

upload_object(*object_name*, *file_obj*, *content_type=None*)
Upload an object to this bucket.

Parameters

- **object_name** (*str*) – The target name of the object.
- **file_obj** (*file*) – The file (or file-like object) to upload.
- **content_type** (*str*) – 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`.

class yamcs.storage.model.**ObjectInfo**(*proto*, *bucket*, *client*)

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

Type datetime

delete()
Remove this object.

download()
Download this object.

property name
The name of this object.

property size
Size in bytes of this object (excluding metadata).

upload(*file_obj*)
Replace the content of this object.

Parameters **file_obj** (*file*) – The file (or file-like object) to upload.

class yamcs.storage.model.**ObjectListing**(*proto*, *bucket*, *client*)

property objects
The objects in this listing.

Type List[*ObjectInfo*]

property prefixes
The prefixes in this listing.

Type str[]

Snippets

Create a *StorageClient* for a specific instance:

```
from yamcs.client import YamcsClient

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

2.2.7 CFDP

The CFDP API provides methods that you can use to programmatically work with CFDP transfers.

Reference

Client

class yamcs.cfdp.**CFDPClient**(*client*, *instance*)
Client for working with CFDP transfers managed by Yamcs.

cancel_transfer(*id*)
Cancel a transfer

create_transfer_subscription(*on_data=None*, *timeout=60*)
Create a new transfer subscription.

Parameters

- **on_data** – (Optional) Function that gets called with `TransferInfo` updates.
- **timeout** (*float*) – 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 TransferSubscription

get_transfer(*id*)
Get a specific transfer.

Return type *Transfer*

list_transfers()
List the transfers.

Return type Iterable[*Transfer*]

pause_transfer(*id*)
Pauses a transfer

resume_transfer(*id*)
Resume a transfer

upload(*bucket_name*, *object_name*, *remote_path*, *overwrite=True*, *parents=True*, *reliable=False*)
Uploads a file located in a bucket to a remote destination path.

Parameters

- **bucket_name** (*str*) – Name of the bucket containing the source object.
- **object_name** (*str*) – Name of the source object.
- **remote_path** (*str*) – Remote destination.
- **overwrite** (*bool*) – Replace a destination if it already exists.
- **parents** (*bool*) – Create the remote path if it does not yet exist.
- **reliable** (*bool*) – Whether to use a Class 2 CFDP transfer.

Return type *Transfer*

Model

class yamcs.cfdp.model.**Transfer**(*proto*, *client*)

Represents a CFDP transfer.

await_complete(*timeout=None*)

Wait for the transfer to be completed.

Parameters **timeout** (*float*) – The amount of seconds to wait.

property bucket

cancel()

Cancel this transfer

property error

Error message in case the transfer failed.

property id

Yamcs-local transfer identifier.

is_complete()

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

is_success()

Returns true if this transfer was completed successfully.

property object_name

pause()

Pause this transfer

property reliable

True if this is a Class 2 CFDP transfer.

property remote_path

resume()

Resume this transfer

property size

Total bytes to transfer.

property state

Current transfer state.

property time

Time when the transfer was started.

property transferred_size
Total bytes already transferred.

Snippets

Create a *CFDPClient* for a specific instance:

```
from yamcs.client import YamcsClient

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

2.2.8 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`.

alarms.py

```
# fmt: off
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, 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
```

(continues on next page)

(continued from previous page)

```
    # the subscription.

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

archive_breakdown.py

```
# fmt: off
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():
        if not name == '/YSS/SIMULATOR/DHS':
            continue
        packet_count = 0
        for group in archive.list_packet_histogram(name):
            for rec in group.records:
                packet_count += rec.count
        print('  {: <40} {: >20}'.format(name, packet_count))


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('  {: <40} {: >20}'.format(group, frame_count))


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('  {: <40} {: >20}'.format(source, event_count))


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('  {: <40} {: >20}'.format(command.qualified_name, total))
```

(continues on next page)

(continued from previous page)

```
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()
```

archive_retrieval.py

```
# fmt: off
from datetime import datetime, timedelta
from itertools import islice

from yamcs.client import YamcsClient


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.utcnow()
    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')
```

(continues on next page)

(continued from previous page)

```
def export_raw_packets():
    """Export raw packet binary."""
    now = datetime.utcnow()
    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.utcnow()
    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.utcnow()
    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')


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()
```

(continues on next page)

(continued from previous page)

```
    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('\nLast 10 commands:')
    print_last_commands()
```

authenticate.py

```
# fmt: off
from time import sleep

from yamcs.client import YamcsClient
from yamcs.core.auth import Credentials

# 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')
    client = YamcsClient('localhost:8090', credentials=credentials)

    for link in client.list_data_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_data_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'))
```

(continues on next page)

(continued from previous page)

```
        sleep(1)


if __name__ == '__main__':
    print('Authenticate with username/password')
    authenticate_with_username_password()

    print('\nImpersonate with client credentials')
    impersonate_with_client_credentials()
```

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.

```
# fmt: off
from datetime import datetime, timedelta

from yamcs.client import YamcsClient


def print_latest():
    """
    Prints completeness records for the last two days
    """
    now = datetime.utcnow()
    start = now - timedelta(days=2)

    # 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('  --> Total packets for {}: {}'.format(apid, total))


if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

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

cfdp.py

```
import io

from yamcs.client import YamcsClient

if __name__ == "__main__":
    client = YamcsClient("localhost:8090")
    storage = client.get_storage_client()
    cfdp = client.get_cfdp_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)

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

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

commanding.py

```
# fmt: off
from time import sleep

from yamcs.client import YamcsClient
from yamcs.tmtc.model import VerificationConfig


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)
```

(continues on next page)

(continued from previous page)

```
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)

    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."""
```

(continues on next page)

(continued from previous page)

```
    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()

    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)
```

cop1.py

```
from time import sleep

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)
```

(continues on next page)

(continued from previous page)

```
    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)
```

links.py

```
# fmt: off
from time import sleep

from yamcs.client import YamcsClient


def enable_link(link):
    """Enable a link."""
    link.enable_link()
    for link in client.list_data_links(instance='simulator'):
        client.enable_data_link(instance=link.instance, link=link.name)


def callback(message):
    print('Link Event: {}'.format(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', callback)
```

(continues on next page)

(continued from previous page)

```
    sleep(10)

    print('-----')
    # You don't have to use the on_data callback. You could also
    # directly retrieve the latest data link state from a local cache:
    print('Last values from cache:')
    for link in subscription.list_data_links():
        print(link)
```

events.py

```
# fmt: off
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)
```

mission_time.py

```
# fmt: off
from time import sleep

from yamcs.client import YamcsClient


def callback(dt):
    print('Mission time:', dt)
```

(continues on next page)

(continued from previous page)

```
if __name__ == '__main__':
    client = YamcsClient('localhost:8090')
    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)
```

parameter_subscription.py

```
# fmt: off
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'))


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'
    ])
```

(continues on next page)

(continued from previous page)

```
    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()
```

plot_with_matplotlib.py

Note: To run this example, install matplotlib:

```
pip install matplotlib
```

```
# fmt: off
from datetime import datetime, timedelta

import matplotlib.pyplot as plt
from yamcs.client import YamcsClient

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

(continues on next page)

(continued from previous page)

```
    stop = datetime.utcnow()
    start = stop - timedelta(hours=1)

    samples = archive.sample_parameter_values(
        '/YSS/SIMULATOR/Altitude', start=start, stop=stop)
    x = [s.time for s in samples]
    y = [s.avg for s in samples]
    plt.subplot(2, 1, 1)
    plt.title('Sampled at ' + str(stop))
    plt.plot(x, y)
    plt.ylabel('Altitude')
    plt.grid()

    samples = archive.sample_parameter_values(
        '/YSS/SIMULATOR/SinkRate', start=start, stop=stop)
    x = [s.time for s in samples]
    y = [s.avg for s in samples]
    plt.subplot(2, 1, 2)
    plt.plot(x, y)
    plt.xlabel('UTC')
    plt.ylabel('Sink Rate')
    plt.grid()

    plt.gcf().canvas.set_window_title('Launch & Landing Simulator')
    plt.show()
```

query_mdb.py

```
# fmt: off
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)


def find_parameter():
    """Find one parameter."""
    p1 = mdb.get_parameter('/YSS/SIMULATOR/BatteryVoltage2')
    print('Via qualified name:', p1)
```

(continues on next page)

(continued from previous page)

```
    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()
```

read_write_parameters.py

```
# fmt: off
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():
```

(continues on next page)

(continued from previous page)

```
    """Writes multiple software parameters."""
    processor.set_parameter_values({
        '/YSS/SIMULATOR/AllowCriticalTC1': False,
        '/YSS/SIMULATOR/AllowCriticalTC2': False,
    })


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()
```

PYTHON MODULE INDEX

y

INDEX

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y