CHAPTER

TEN

API REFERENCE

Communication over Reticulum networks is achieved by using a simple set of classes exposed by the RNS API. This chapter lists and explains all classes exposed by the Reticulum Network Stack API, along with their method signatures and usage. It can be used as a reference while writing applications that utilise Reticulum, or it can be read in entirity to gain an understanding of the complete functionality of RNS from a developers perspective.

10.1 Reticulum

class RNS.**Reticulum**(*configdir=None*, *loglevel=None*, *logdest=None*, *verbosity=None*, *require_shared_instance=False*)

This class is used to initialise access to Reticulum within a program. You must create exactly one instance of this class before carrying out any other RNS operations, such as creating destinations or sending traffic. Every independently executed program must create their own instance of the Reticulum class, but Reticulum will automatically handle inter-program communication on the same system, and expose all connected programs to external interfaces as well.

As soon as an instance of this class is created, Reticulum will start opening and configuring any hardware devices specified in the supplied configuration.

Currently the first running instance must be kept running while other local instances are connected, as the first created instance will act as a master instance that directly communicates with external hardware such as modems, TNCs and radios. If a master instance is asked to exit, it will not exit until all client processes have terminated (unless killed forcibly).

If you are running Reticulum on a system with several different programs that use RNS starting and terminating at different times, it will be advantageous to run a master RNS instance as a daemon for other programs to use on demand.

MTU = 500

The MTU that Reticulum adheres to, and will expect other peers to adhere to. By default, the MTU is 500 bytes. In custom RNS network implementations, it is possible to change this value, but doing so will completely break compatibility with all other RNS networks. An identical MTU is a prerequisite for peers to communicate in the same network.

Unless you really know what you are doing, the MTU should be left at the default value.

LINK_MTU_DISCOVERY = True

Whether automatic link MTU discovery is enabled by default in this release. Link MTU discovery significantly increases throughput over fast links, but requires all intermediary hops to also support it. Support for this feature was added in RNS version 0.9.0. This option will become enabled by default in the near future. Please update your RNS instances.

ANNOUNCE_CAP = 2

The maximum percentage of interface bandwidth that, at any given time, may be used to propagate announces. If an announce was scheduled for broadcasting on an interface, but doing so would exceed the allowed bandwidth allocation, the announce will be queued for transmission when there is bandwidth available.

Reticulum will always prioritise propagating announces with fewer hops, ensuring that distant, large networks with many peers on fast links don't overwhelm the capacity of smaller networks on slower mediums. If an announce remains queued for an extended amount of time, it will eventually be dropped.

This value will be applied by default to all created interfaces, but it can be configured individually on a per-interface basis. In general, the global default setting should not be changed, and any alterations should be made on a per-interface basis instead.

MINIMUM_BITRATE = 5

Minimum bitrate required across a medium for Reticulum to be able to successfully establish links. Currently 5 bits per second.

static get_instance()

Return the currently running Reticulum instance

static should_use_implicit_proof()

Returns whether proofs sent are explicit or implicit.

Returns
True if the current running configuration specifies to use implicit proofs. False if not.

static transport_enabled()

Returns whether Transport is enabled for the running instance.

When Transport is enabled, Reticulum will route traffic for other peers, respond to path requests and pass announces over the network.

Returns
True if Transport is enabled, False if not.

static link_mtu_discovery()

Returns whether link MTU discovery is enabled for the running instance.

When link MTU discovery is enabled, Reticulum will automatically upgrade link MTUs to the highest supported value, increasing transfer speed and efficiency.

Returns
True if link MTU discovery is enabled, False if not.

static remote_management_enabled()

Returns whether remote management is enabled for the running instance.

When remote management is enabled, authenticated peers can remotely query and manage this instance.

Returns
True if remote management is enabled, False if not.

10.2 Identity

class RNS.**Identity**(*create_keys=True*)

This class is used to manage identities in Reticulum. It provides methods for encryption, decryption, signatures and verification, and is the basis for all encrypted communication over Reticulum networks.

Parameters
create_keys – Specifies whether new encryption and signing keys should be generated.

CURVE = 'Curve25519'

The curve used for Elliptic Curve DH key exchanges

KEYSIZE = 512

X.25519 key size in bits. A complete key is the concatenation of a 256 bit encryption key, and a 256 bit signing key.

RATCHETSIZE = 256

X.25519 ratchet key size in bits.

RATCHET_EXPIRY = 2592000

The expiry time for received ratchets in seconds, defaults to 30 days. Reticulum will always use the most recently announced ratchet, and remember it for up to RATCHET_EXPIRY since receiving it, after which it will be discarded. If a newer ratchet is announced in the meantime, it will be replace the already known ratchet.

TRUNCATED_HASHLENGTH = 128

Constant specifying the truncated hash length (in bits) used by Reticulum for addressable hashes and other purposes. Non-configurable.

static recall(*destination_hash*)

Recall identity for a destination hash.

Parameters
destination_hash – Destination hash as *bytes*.

Returns
An *RNS.Identity* instance that can be used to create an outgoing *RNS.Destination*, or *None* if the destination is unknown.

static recall_app_data(*destination_hash*)

Recall last heard app_data for a destination hash.

Parameters
destination_hash – Destination hash as *bytes*.

Returns
Bytes containing app_data, or *None* if the destination is unknown.

static full_hash(*data*)

Get a SHA-256 hash of passed data.

Parameters
data – Data to be hashed as *bytes*.

Returns
SHA-256 hash as *bytes*.

`static truncated_hash(data)`

Get a truncated SHA-256 hash of passed data.

Parameters
data – Data to be hashed as *bytes*.

Returns
Truncated SHA-256 hash as *bytes*.

`static get_random_hash()`

Get a random SHA-256 hash.

Parameters
data – Data to be hashed as *bytes*.

Returns
Truncated SHA-256 hash of random data as *bytes*.

`static current_ratchet_id(destination_hash)`

Get the ID of the currently used ratchet key for a given destination hash

Parameters
destination_hash – A destination hash as *bytes*.

Returns
A ratchet ID as *bytes* or *None*.

`static from_bytes(prv_bytes)`

Create a new *RNS.Identity* instance from *bytes* of private key. Can be used to load previously created and saved identities into Reticulum.

Parameters
prv_bytes – The *bytes* of private a saved private key. **HAZARD!** Never use this to generate a new key by feeding random data in prv_bytes.

Returns
A *RNS.Identity* instance, or *None* if the *bytes* data was invalid.

`static from_file(path)`

Create a new *RNS.Identity* instance from a file. Can be used to load previously created and saved identities into Reticulum.

Parameters
path – The full path to the saved *RNS.Identity* data

Returns
A *RNS.Identity* instance, or *None* if the loaded data was invalid.

`to_file(path)`

Saves the identity to a file. This will write the private key to disk, and anyone with access to this file will be able to decrypt all communication for the identity. Be very careful with this method.

Parameters
path – The full path specifying where to save the identity.

Returns
True if the file was saved, otherwise False.

`get_private_key()`

Returns
The private key as *bytes*

`get_public_key()`

Returns
The public key as *bytes*

`load_private_key(prv_bytes)`

Load a private key into the instance.

Parameters
prv_bytes – The private key as *bytes*.

Returns
True if the key was loaded, otherwise False.

`load_public_key(pub_bytes)`

Load a public key into the instance.

Parameters
pub_bytes – The public key as *bytes*.

Returns
True if the key was loaded, otherwise False.

`encrypt(plaintext, ratchet=None)`

Encrypts information for the identity.

Parameters
plaintext – The plaintext to be encrypted as *bytes*.

Returns
Ciphertext token as *bytes*.

Raises
KeyError if the instance does not hold a public key.

`decrypt(ciphertext_token, ratchets=None, enforce_ratchets=False, ratchet_id_receiver=None)`

Decrypts information for the identity.

Parameters
ciphertext – The ciphertext to be decrypted as *bytes*.

Returns
Plaintext as *bytes*, or *None* if decryption fails.

Raises
KeyError if the instance does not hold a private key.

`sign(message)`

Signs information by the identity.

Parameters
message – The message to be signed as *bytes*.

Returns
Signature as *bytes*.

Raises
KeyError if the instance does not hold a private key.

`validate(signature, message)`

Validates the signature of a signed message.

Parameters

- **signature** – The signature to be validated as *bytes*.
- **message** – The message to be validated as *bytes*.

Returns
True if the signature is valid, otherwise False.

Raises
KeyError if the instance does not hold a public key.

10.3 Destination

class RNS.**Destination**(*identity*, *direction*, *type*, *app_name*, **aspects*)

A class used to describe endpoints in a Reticulum Network. Destination instances are used both to create outgoing and incoming endpoints. The destination type will decide if encryption, and what type, is used in communication with the endpoint. A destination can also announce its presence on the network, which will distribute necessary keys for encrypted communication with it.

Parameters

- **identity** – An instance of *RNS.Identity*. Can hold only public keys for an outgoing destination, or holding private keys for an ingoing.
- **direction** – RNS.Destination.IN or RNS.Destination.OUT.
- **type** – RNS.Destination.SINGLE, RNS.Destination.GROUP or RNS.Destination.PLAIN.
- **app_name** – A string specifying the app name.
- ***aspects** – Any non-zero number of string arguments.

RATCHET_COUNT = 512

The default number of generated ratchet keys a destination will retain, if it has ratchets enabled.

RATCHET_INTERVAL = 1800

The minimum interval between rotating ratchet keys, in seconds.

static expand_name(*identity*, *app_name*, **aspects*)

Returns
A string containing the full human-readable name of the destination, for an app_name and a number of aspects.

static app_and_aspects_from_name(*full_name*)

Returns
A tuple containing the app name and a list of aspects, for a full-name string.

static hash_from_name_and_identity(*full_name*, *identity*)

Returns
A destination name in adressable hash form, for a full name string and Identity instance.

static hash(*identity*, *app_name*, **aspects*)

Returns
A destination name in adressable hash form, for an app_name and a number of aspects.

announce(*app_data=None*, *path_response=False*, *attached_interface=None*, *tag=None*, *send=True*)

Creates an announce packet for this destination and broadcasts it on all relevant interfaces. Application specific data can be added to the announce.

Parameters

- **app_data** – *bytes* containing the app_data.
- **path_response** – Internal flag used by *RNS.Transport*. Ignore.

accepts_links(*accepts=None*)

Set or query whether the destination accepts incoming link requests.

Parameters

accepts – If `True` or `False`, this method sets whether the destination accepts incoming link requests. If not provided or `None`, the method returns whether the destination currently accepts link requests.

Returns

`True` or `False` depending on whether the destination accepts incoming link requests, if the *accepts* parameter is not provided or `None`.

set_link_established_callback(*callback*)

Registers a function to be called when a link has been established to this destination.

Parameters

callback – A function or method with the signature *callback(link)* to be called when a new link is established with this destination.

set_packet_callback(*callback*)

Registers a function to be called when a packet has been received by this destination.

Parameters

callback – A function or method with the signature *callback(data, packet)* to be called when this destination receives a packet.

set_proof_requested_callback(*callback*)

Registers a function to be called when a proof has been requested for a packet sent to this destination. Allows control over when and if proofs should be returned for received packets.

Parameters

callback – A function or method to with the signature *callback(packet)* be called when a packet that requests a proof is received. The callback must return one of True or False. If the callback returns True, a proof will be sent. If it returns False, a proof will not be sent.

set_proof_strategy(*proof_strategy*)

Sets the destinations proof strategy.

Parameters

proof_strategy – One of `RNS.Destination.PROVE_NONE`, `RNS.Destination.PROVE_ALL` or `RNS.Destination.PROVE_APP`. If `RNS.Destination.PROVE_APP` is set, the *proof_requested_callback* will be called to determine whether a proof should be sent or not.

register_request_handler(*path*, *response_generator=None*, *allow=ALLOW_NONE*, *allowed_list=None*)

Registers a request handler.

Parameters

- **path** – The path for the request handler to be registered.

- **response_generator** – A function or method with the signature *response_generator(path, data, request_id, link_id, remote_identity, requested_at)* to be called. Whatever this funcion returns will be sent as a response to the requester. If the function returns `None`, no response will be sent.
- **allow** – One of `RNS.Destination.ALLOW_NONE`, `RNS.Destination.ALLOW_ALL` or `RNS.Destination.ALLOW_LIST`. If `RNS.Destination.ALLOW_LIST` is set, the request handler will only respond to requests for identified peers in the supplied list.
- **allowed_list** – A list of *bytes-like RNS.Identity* hashes.

Raises
`ValueError` if any of the supplied arguments are invalid.

deregister_request_handler(*path*)

Deregisters a request handler.

Parameters
path – The path for the request handler to be deregistered.

Returns
True if the handler was deregistered, otherwise False.

enable_ratchets(*ratchets_path*)

Enables ratchets on the destination. When ratchets are enabled, Reticulum will automatically rotate the keys used to encrypt packets to this destination, and include the latest ratchet key in announces.

Enabling ratchets on a destination will provide forward secrecy for packets sent to that destination, even when sent outside a `Link`. The normal Reticulum `Link` establishment procedure already performs its own ephemeral key exchange for each link establishment, which means that ratchets are not necessary to provide forward secrecy for links.

Enabling ratchets will have a small impact on announce size, adding 32 bytes to every sent announce.

Parameters
ratchets_path – The path to a file to store ratchet data in.

Returns
True if the operation succeeded, otherwise False.

enforce_ratchets()

When ratchet enforcement is enabled, this destination will never accept packets that use its base Identity key for encryption, but only accept packets encrypted with one of the retained ratchet keys.

set_retained_ratchets(*retained_ratchets*)

Sets the number of previously generated ratchet keys this destination will retain, and try to use when decrypting incoming packets. Defaults to `Destination.RATCHET_COUNT`.

Parameters
retained_ratchets – The number of generated ratchets to retain.

Returns
True if the operation succeeded, False if not.

set_ratchet_interval(*interval*)

Sets the minimum interval in seconds between ratchet key rotation. Defaults to `Destination.RATCHET_INTERVAL`.

Parameters
interval – The minimum interval in seconds.

Returns
True if the operation succeeded, False if not.

`create_keys()`

For a `RNS.Destination.GROUP` type destination, creates a new symmetric key.

Raises
`TypeError` if called on an incompatible type of destination.

`get_private_key()`

For a `RNS.Destination.GROUP` type destination, returns the symmetric private key.

Raises
`TypeError` if called on an incompatible type of destination.

`load_private_key(key)`

For a `RNS.Destination.GROUP` type destination, loads a symmetric private key.

Parameters
key – A *bytes-like* containing the symmetric key.

Raises
`TypeError` if called on an incompatible type of destination.

`encrypt(plaintext)`

Encrypts information for `RNS.Destination.SINGLE` or `RNS.Destination.GROUP` type destination.

Parameters
plaintext – A *bytes-like* containing the plaintext to be encrypted.

Raises
`ValueError` if destination does not hold a necessary key for encryption.

`decrypt(ciphertext)`

Decrypts information for `RNS.Destination.SINGLE` or `RNS.Destination.GROUP` type destination.

Parameters
ciphertext – *Bytes* containing the ciphertext to be decrypted.

Raises
`ValueError` if destination does not hold a necessary key for decryption.

`sign(message)`

Signs information for `RNS.Destination.SINGLE` type destination.

Parameters
message – *Bytes* containing the message to be signed.

Returns
A *bytes-like* containing the message signature, or *None* if the destination could not sign the message.

`set_default_app_data(app_data=None)`

Sets the default app_data for the destination. If set, the default app_data will be included in every announce sent by the destination, unless other app_data is specified in the *announce* method.

Parameters
app_data – A *bytes-like* containing the default app_data, or a *callable* returning a *bytes-like* containing the app_data.

`clear_default_app_data()`

Clears default app_data previously set for the destination.

10.4 Packet

class RNS.**Packet**(*destination*, *data*, *create_receipt=True*)

The Packet class is used to create packet instances that can be sent over a Reticulum network. Packets will automatically be encrypted if they are addressed to a `RNS.Destination.SINGLE` destination, `RNS.Destination.GROUP` destination or a *RNS.Link*.

For `RNS.Destination.GROUP` destinations, Reticulum will use the pre-shared key configured for the destination. All packets to group destinations are encrypted with the same AES-128 key.

For `RNS.Destination.SINGLE` destinations, Reticulum will use a newly derived ephemeral AES-128 key for every packet.

For *RNS.Link* destinations, Reticulum will use per-link ephemeral keys, and offers **Forward Secrecy**.

Parameters

- **destination** – A *RNS.Destination* instance to which the packet will be sent.
- **data** – The data payload to be included in the packet as *bytes*.
- **create_receipt** – Specifies whether a *RNS.PacketReceipt* should be created when instantiating the packet.

ENCRYPTED_MDU = 383

The maximum size of the payload data in a single encrypted packet

PLAIN_MDU = 464

The maximum size of the payload data in a single unencrypted packet

send()

Sends the packet.

Returns

A *RNS.PacketReceipt* instance if *create_receipt* was set to *True* when the packet was instantiated, if not returns *None*. If the packet could not be sent *False* is returned.

resend()

Re-sends the packet.

Returns

A *RNS.PacketReceipt* instance if *create_receipt* was set to *True* when the packet was instantiated, if not returns *None*. If the packet could not be sent *False* is returned.

get_rssi()

Returns

The physical layer *Received Signal Strength Indication* if available, otherwise `None`.

get_snr()

Returns

The physical layer *Signal-to-Noise Ratio* if available, otherwise `None`.

get_q()

Returns

The physical layer *Link Quality* if available, otherwise `None`.

10.5 Packet Receipt

class RNS.**PacketReceipt**

The PacketReceipt class is used to receive notifications about *RNS.Packet* instances sent over the network. Instances of this class are never created manually, but always returned from the *send()* method of a *RNS.Packet* instance.

get_status()

Returns
The status of the associated *RNS.Packet* instance. Can be one of `RNS.PacketReceipt.SENT`, `RNS.PacketReceipt.DELIVERED`, `RNS.PacketReceipt.FAILED` or `RNS.PacketReceipt.CULLED`.

get_rtt()

Returns
The round-trip-time in seconds

set_timeout(*timeout*)

Sets a timeout in seconds

Parameters
timeout – The timeout in seconds.

set_delivery_callback(*callback*)

Sets a function that gets called if a successfull delivery has been proven.

Parameters
callback – A *callable* with the signature *callback(packet_receipt)*

set_timeout_callback(*callback*)

Sets a function that gets called if the delivery times out.

Parameters
callback – A *callable* with the signature *callback(packet_receipt)*

10.6 Link

class RNS.**Link**(*destination*, *established_callback=None*, *closed_callback=None*)

This class is used to establish and manage links to other peers. When a link instance is created, Reticulum will attempt to establish verified and encrypted connectivity with the specified destination.

Parameters

- **destination** – A *RNS.Destination* instance which to establish a link to.
- **established_callback** – An optional function or method with the signature *callback(link)* to be called when the link has been established.
- **closed_callback** – An optional function or method with the signature *callback(link)* to be called when the link is closed.

CURVE = 'Curve25519'

The curve used for Elliptic Curve DH key exchanges

ESTABLISHMENT_TIMEOUT_PER_HOP = 6

Timeout for link establishment in seconds per hop to destination.

KEEPALIVE_TIMEOUT_FACTOR = 4

RTT timeout factor used in link timeout calculation.

STALE_GRACE = 2

Grace period in seconds used in link timeout calculation.

KEEPALIVE = 360

Interval for sending keep-alive packets on established links in seconds.

STALE_TIME = 720

If no traffic or keep-alive packets are received within this period, the link will be marked as stale, and a final keep-alive packet will be sent. If after this no traffic or keep-alive packets are received within `RTT * KEEPALIVE_TIMEOUT_FACTOR + STALE_GRACE`, the link is considered timed out, and will be torn down.

identify(*identity*)

Identifies the initiator of the link to the remote peer. This can only happen once the link has been established, and is carried out over the encrypted link. The identity is only revealed to the remote peer, and initiator anonymity is thus preserved. This method can be used for authentication.

Parameters

identity – An RNS.Identity instance to identify as.

request(*path*, *data=None*, *response_callback=None*, *failed_callback=None*, *progress_callback=None*, *timeout=None*)

Sends a request to the remote peer.

Parameters

- **path** – The request path.
- **response_callback** – An optional function or method with the signature *response_callback(request_receipt)* to be called when a response is received. See the *Request Example* for more info.
- **failed_callback** – An optional function or method with the signature *failed_callback(request_receipt)* to be called when a request fails. See the *Request Example* for more info.
- **progress_callback** – An optional function or method with the signature *progress_callback(request_receipt)* to be called when progress is made receiving the response. Progress can be accessed as a float between 0.0 and 1.0 by the *request_receipt.progress* property.
- **timeout** – An optional timeout in seconds for the request. If *None* is supplied it will be calculated based on link RTT.

Returns

A *RNS.RequestReceipt* instance if the request was sent, or *False* if it was not.

track_phy_stats(*track*)

You can enable physical layer statistics on a per-link basis. If this is enabled, and the link is running over an interface that supports reporting physical layer statistics, you will be able to retrieve stats such as *RSSI*, *SNR* and physical *Link Quality* for the link.

Parameters

track – Whether or not to keep track of physical layer statistics. Value must be `True` or `False`.

get_rssi()

Returns
The physical layer *Received Signal Strength Indication* if available, otherwise None. Physical layer statistics must be enabled on the link for this method to return a value.

get_snr()

Returns
The physical layer *Signal-to-Noise Ratio* if available, otherwise None. Physical layer statistics must be enabled on the link for this method to return a value.

get_q()

Returns
The physical layer *Link Quality* if available, otherwise None. Physical layer statistics must be enabled on the link for this method to return a value.

get_establishment_rate()

Returns
The data transfer rate at which the link establishment procedure ocurred, in bits per second.

get_mtu()

Returns
The MTU of an established link.

get_mdu()

Returns
The packet MDU of an established link.

get_expected_rate()

Returns
The packet expected in-flight data rate of an established link.

get_age()

Returns
The time in seconds since this link was established.

no_inbound_for()

Returns
The time in seconds since last inbound packet on the link. This includes keepalive packets.

no_outbound_for()

Returns
The time in seconds since last outbound packet on the link. This includes keepalive packets.

no_data_for()

Returns
The time in seconds since payload data traversed the link. This excludes keepalive packets.

inactive_for()

Returns
The time in seconds since activity on the link. This includes keepalive packets.

`get_remote_identity()`

Returns
The identity of the remote peer, if it is known. Calling this method will not query the remote initiator to reveal its identity. Returns `None` if the link initiator has not already independently called the `identify(identity)` method.

`teardown()`

Closes the link and purges encryption keys. New keys will be used if a new link to the same destination is established.

`get_channel()`

Get the `Channel` for this link.

Returns
`Channel` object

`set_link_closed_callback(callback)`

Registers a function to be called when a link has been torn down.

Parameters
callback – A function or method with the signature *callback(link)* to be called.

`set_packet_callback(callback)`

Registers a function to be called when a packet has been received over this link.

Parameters
callback – A function or method with the signature *callback(message, packet)* to be called.

`set_resource_callback(callback)`

Registers a function to be called when a resource has been advertised over this link. If the function returns *True* the resource will be accepted. If it returns *False* it will be ignored.

Parameters
callback – A function or method with the signature *callback(resource)* to be called. Please note that only the basic information of the resource is available at this time, such as *get_transfer_size()*, *get_data_size()*, *get_parts()* and *is_compressed()*.

`set_resource_started_callback(callback)`

Registers a function to be called when a resource has begun transferring over this link.

Parameters
callback – A function or method with the signature *callback(resource)* to be called.

`set_resource_concluded_callback(callback)`

Registers a function to be called when a resource has concluded transferring over this link.

Parameters
callback – A function or method with the signature *callback(resource)* to be called.

`set_remote_identified_callback(callback)`

Registers a function to be called when an initiating peer has identified over this link.

Parameters
callback – A function or method with the signature *callback(link, identity)* to be called.

`set_resource_strategy(resource_strategy)`

Sets the resource strategy for the link.

Parameters
resource_strategy – One of RNS.Link.ACCEPT_NONE, RNS.Link.ACCEPT_ALL or RNS.Link.ACCEPT_APP. If RNS.Link.ACCEPT_APP is set, the *resource_callback* will be called to determine whether the resource should be accepted or not.

Raises
TypeError if the resource strategy is unsupported.

10.7 Request Receipt

class RNS.**RequestReceipt**

An instance of this class is returned by the request method of RNS.Link instances. It should never be instantiated manually. It provides methods to check status, response time and response data when the request concludes.

get_request_id()

Returns
The request ID as *bytes*.

get_status()

Returns
The current status of the request, one of RNS.RequestReceipt.FAILED, RNS.RequestReceipt.SENT, RNS.RequestReceipt.DELIVERED, RNS.RequestReceipt.READY.

get_progress()

Returns
The progress of a response being received as a *float* between 0.0 and 1.0.

get_response()

Returns
The response as *bytes* if it is ready, otherwise *None*.

get_response_time()

Returns
The response time of the request in seconds.

concluded()

Returns
True if the associated request has concluded (successfully or with a failure), otherwise False.

10.8 Resource

class RNS.**Resource**(*data*, *link*, *advertise=True*, *auto_compress=True*, *callback=None*, *progress_callback=None*, *timeout=None*)

The Resource class allows transferring arbitrary amounts of data over a link. It will automatically handle sequencing, compression, coordination and checksumming.

Parameters

- **data** – The data to be transferred. Can be *bytes* or an open *file handle*. See the *Filetransfer Example* for details.
- **link** – The *RNS.Link* instance on which to transfer the data.
- **advertise** – Optional. Whether to automatically advertise the resource. Can be *True* or *False*.
- **auto_compress** – Optional. Whether to auto-compress the resource. Can be *True* or *False*.
- **callback** – An optional *callable* with the signature *callback(resource)*. Will be called when the resource transfer concludes.
- **progress_callback** – An optional *callable* with the signature *callback(resource)*. Will be called whenever the resource transfer progress is updated.

advertise()

Advertise the resource. If the other end of the link accepts the resource advertisement it will begin transferring.

cancel()

Cancels transferring the resource.

get_progress()

Returns
The current progress of the resource transfer as a *float* between 0.0 and 1.0.

get_transfer_size()

Returns
The number of bytes needed to transfer the resource.

get_data_size()

Returns
The total data size of the resource.

get_parts()

Returns
The number of parts the resource will be transferred in.

get_segments()

Returns
The number of segments the resource is divided into.

get_hash()

Returns
The hash of the resource.

is_compressed()

Returns
Whether the resource is compressed.

10.9 Channel

class RNS.Channel.**Channel**

Provides reliable delivery of messages over a link.

Channel differs from Request and Resource in some important ways:

Continuous
Messages can be sent or received as long as the Link is open.

Bi-directional
Messages can be sent in either direction on the Link; neither end is the client or server.

Size-constrained
Messages must be encoded into a single packet.

Channel is similar to Packet, except that it provides reliable delivery (automatic retries) as well as a structure for exchanging several types of messages over the Link.

Channel is not instantiated directly, but rather obtained from a Link with get_channel().

register_message_type(*message_class: Type[*MessageBase*]*)

Register a message class for reception over a Channel.

Message classes must extend MessageBase.

Parameters
message_class – Class to register

add_message_handler(*callback: MessageCallbackType*)

Add a handler for incoming messages. A handler has the following signature:

```
(message: MessageBase) -> bool
```

Handlers are processed in the order they are added. If any handler returns True, processing of the message stops; handlers after the returning handler will not be called.

Parameters
callback – Function to call

remove_message_handler(*callback: MessageCallbackType*)

Remove a handler added with add_message_handler.

Parameters
callback – handler to remove

is_ready_to_send() → bool

Check if Channel is ready to send.

Returns
True if ready

send(*message:* MessageBase) → Envelope

Send a message. If a message send is attempted and `Channel` is not ready, an exception is thrown.

Parameters
message – an instance of a `MessageBase` subclass

property mdu

Maximum Data Unit: the number of bytes available for a message to consume in a single send. This value is adjusted from the `Link` MDU to accommodate message header information.

Returns
number of bytes available

10.10 MessageBase

class RNS.**MessageBase**

Base type for any messages sent or received on a Channel. Subclasses must define the two abstract methods as well as the `MSGTYPE` class variable.

MSGTYPE = None

Defines a unique identifier for a message class.

- Must be unique within all classes registered with a `Channel`
- Must be less than `0xf000`. Values greater than or equal to `0xf000` are reserved.

abstract pack() → bytes

Create and return the binary representation of the message

Returns
binary representation of message

abstract unpack(*raw: bytes*)

Populate message from binary representation

Parameters
raw – binary representation

10.11 Buffer

class RNS.**Buffer**

Static functions for creating buffered streams that send and receive over a `Channel`.

These functions use `BufferedReader`, `BufferedWriter`, and `BufferedRWPair` to add buffering to `RawChannelReader` and `RawChannelWriter`.

static create_reader(*stream_id: int*, *channel:* Channel, *ready_callback: Optional[Callable[[int], None]] = None*) → BufferedReader

Create a buffered reader that reads binary data sent over a `Channel`, with an optional callback when new data is available.

Callback signature: `(ready_bytes: int) -> None`

For more information on the reader-specific functions of this object, see the Python documentation for `BufferedReader`

Parameters

- **stream_id** – the local stream id to receive from
- **channel** – the channel to receive on
- **ready_callback** – function to call when new data is available

Returns
a BufferedReader object

static create_writer(*stream_id: int*, *channel:* Channel) → BufferedWriter

Create a buffered writer that writes binary data over a `Channel`.

For more information on the writer-specific functions of this object, see the Python documentation for `BufferedWriter`

Parameters

- **stream_id** – the remote stream id to send to
- **channel** – the channel to send on

Returns
a BufferedWriter object

static create_bidirectional_buffer(*receive_stream_id: int*, *send_stream_id: int*, *channel:* Channel, *ready_callback: Optional[Callable[[int], None]] = None*) → BufferedRWPair

Create a buffered reader/writer pair that reads and writes binary data over a `Channel`, with an optional callback when new data is available.

Callback signature: `(ready_bytes: int) -> None`

For more information on the reader-specific functions of this object, see the Python documentation for `BufferedRWPair`

Parameters

- **receive_stream_id** – the local stream id to receive at
- **send_stream_id** – the remote stream id to send to
- **channel** – the channel to send and receive on
- **ready_callback** – function to call when new data is available

Returns
a BufferedRWPair object

10.12 RawChannelReader

class RNS.**RawChannelReader**(*stream_id: int*, *channel:* Channel)

An implementation of RawIOBase that receives binary stream data sent over a `Channel`.

This class generally need not be instantiated directly. Use *RNS.Buffer.create_reader()*, *RNS.Buffer.create_writer()*, and *RNS.Buffer.create_bidirectional_buffer()* functions to create buffered streams with optional callbacks.

For additional information on the API of this object, see the Python documentation for `RawIOBase`.

__init__(*stream_id: int*, *channel:* Channel)

Create a raw channel reader.

Parameters

- **stream_id** – local stream id to receive at
- **channel** – Channel object to receive from

add_ready_callback(*cb: Callable[[int], None]*)

Add a function to be called when new data is available. The function should have the signature `(ready_bytes: int) -> None`

Parameters

cb – function to call

remove_ready_callback(*cb: Callable[[int], None]*)

Remove a function added with *RNS.RawChannelReader.add_ready_callback()*

Parameters

cb – function to remove

10.13 RawChannelWriter

class RNS.**RawChannelWriter**(*stream_id: int*, *channel:* Channel)

An implementation of RawIOBase that receives binary stream data sent over a channel.

This class generally need not be instantiated directly. Use *RNS.Buffer.create_reader()*, *RNS.Buffer.create_writer()*, and *RNS.Buffer.create_bidirectional_buffer()* functions to create buffered streams with optional callbacks.

For additional information on the API of this object, see the Python documentation for `RawIOBase`.

__init__(*stream_id: int*, *channel:* Channel)

Create a raw channel writer.

Parameters

- **stream_id** – remote stream id to sent do
- **channel** – Channel object to send on

10.14 Transport

class RNS.**Transport**

Through static methods of this class you can interact with the Transport system of Reticulum.

PATHFINDER_M = 128

Maximum amount of hops that Reticulum will transport a packet.

static register_announce_handler(*handler*)

Registers an announce handler.

Parameters

handler – Must be an object with an *aspect_filter* attribute and a *received_announce(destination_hash, announced_identity, app_data)* or *received_announce(destination_hash, announced_identity, app_data, announce_packet_hash)*

callable. Can optionally have a *receive_path_responses* attribute set to `True`, to also receive all path responses, in addition to live announces. See the *Announce Example* for more info.

static deregister_announce_handler(*handler*)

Deregisters an announce handler.

Parameters
handler – The announce handler to be deregistered.

static has_path(*destination_hash*)

Parameters
destination_hash – A destination hash as *bytes*.

Returns
True if a path to the destination is known, otherwise *False*.

static hops_to(*destination_hash*)

Parameters
destination_hash – A destination hash as *bytes*.

Returns
The number of hops to the specified destination, or `RNS.Transport.PATHFINDER_M` if the number of hops is unknown.

static next_hop(*destination_hash*)

Parameters
destination_hash – A destination hash as *bytes*.

Returns
The destination hash as *bytes* for the next hop to the specified destination, or *None* if the next hop is unknown.

static next_hop_interface(*destination_hash*)

Parameters
destination_hash – A destination hash as *bytes*.

Returns
The interface for the next hop to the specified destination, or *None* if the interface is unknown.

static request_path(*destination_hash*, *on_interface=None*, *tag=None*, *recursive=False*)

Requests a path to the destination from the network. If another reachable peer on the network knows a path, it will announce it.

Parameters

- **destination_hash** – A destination hash as *bytes*.
- **on_interface** – If specified, the path request will only be sent on this interface. In normal use, Reticulum handles this automatically, and this parameter should not be used.