

---

# **quickrpc Documentation**

***Release 0.1***

**J. Loehnert**

**Sep 07, 2020**



---

## Contents

---

<b>1</b>	<b>API documentation</b>	<b>3</b>
1.1	quickrpc package . . . . .	3
1.2	quickrpc.remote_api module . . . . .	6
1.3	quickrpc.transports, .network_transports and .QtTransports modules . . . . .	9
1.4	quickrpc.codecs and .terse_codec modules . . . . .	17
1.5	quickrpc.security module . . . . .	22
1.6	quickrpc.promise and .action_queue module . . . . .	23
1.7	Other modules . . . . .	25
<b>2</b>	<b>Indices and tables</b>	<b>29</b>
	<b>Python Module Index</b>	<b>31</b>
	<b>Index</b>	<b>33</b>



Contents:



### 1.1 quickrpc package

QuickRPC is a library for quick and painless setup of communication channels and Remote-call protocols.

#### Python 3 only

To use QuickRPC, you define a *RemoteAPI* subclass. This is a special interface-like class whose methods define the possible incoming and outgoing calls.

Second, a *Codec* is needed to translate method calls into byte strings and vice-versa. This could for example be JSON-RPC or MsgPack codec.

Third, the *RemoteAPI* is bound to a *Transport*. This is basically a send-and-receive channel out of your program. Predefined transports include Stdio, TCP client and server as well as UDP. Additionally there are wrappers that can merge multiple transports together and restart a failing transport.

Codecs and Transports can be instantiated from a textual definition, so that they can easily put in a config file or on the commandline. See *transport()* (alias of *transports.Transport.fromstring()*) and *codec()* (alias of *codecs.Codec.fromstring()*).

```
class quickrpc.RemoteAPI(codec='jsonrpc', transport=None, security='null', invert=False,
                        async_processing=False)
```

Bases: object

Describes an API i.e. a set of allowed outgoing and incoming calls.

Subclass and add your calls.

codec holds the Codec for (de)serializing data. *transport* holds the underlying transport. *security* holds the security provider.

Both can also be strings, then *Transport.fromstring()* / *Codec.fromstring()* are used to acquire the respective objects. In this case, transport still needs to be started via *myapi.transport.start()*.

Methods marked as @outgoing are automatically turned into messages when called. The method body is executed before sending. (use e.g. for validation of outgoing data). They must accept a special *receivers* argument, which is passed to the Transport.

Methods marked as `@incoming` are called by the transport when messages arrive. They work like signals - you can connect your own handler(s) to them. Connected handlers must have the same signature as the incoming call. All `@incoming` methods MUST support a *senders* argument.

Connect like this:

```
>>> def handler(self, foo=None): pass
>>> remote_api.some_method.connect(handler)
>>> # later
>>> remote_api.some_method.disconnect(handler)
```

Execution order: the method of `remote_api` is executed first, then the connected handlers in the order of registering.

Incoming messages with unknown method will not be processed. If the message has `.id != 0`, it will automatically be replied with an error.

Threading:

- outgoing messages are sent on the calling thread.
- If `async_processing = False`, incoming messages are handled on the thread which handles Transport receive events. I.e. the Transport implementation defines the behaviour.
- If `async_processing = True`, an extra Thread is used to handle messages.

The latter allows the receive handler to run concurrently to message handling, allowing further requests to be sent out and to await the result. However it means one extra thread. In any case, only one incoming message is handled at a time.

Recommendation is to set `async_processing=True` if there are any outgoing calls that have a reply, `False` if not.

Inverting:

You can `invert()` the whole api, swapping incoming and outgoing methods. When inverted, the `sender` and `receiver` arguments of each method swap their roles. This is also possible upon initialization by giving `invert=True` kwarg.

**invert()**

Swaps `@incoming` and `@outgoing` decoration on all methods of this INSTANCE.

I.e. generates the opposite-side API.

Do this before connecting any handlers to incoming calls.

You can achieve the same effect by instantiating with `invert=True` kwarg.

**message\_error** (*sender, exception, in\_reply\_to=None*)

Called each time that an incoming message causes problems.

By default, it logs the error as warning. `in_reply_to` is the message that triggered the error, `None` if decoding failed. If the requested method can be identified and has a reply, an error reply is returned to the sender.

**transport**

Gets/sets the transport used to send and receive messages.

You can change the transport at runtime.

**unhandled\_calls()**

Generator, returns the names of all *incoming*, unconnected methods.



If no results are returned, all incoming messages are connected. Use this to check for missed `.connect` calls.

`quickrpc.incoming` (*unbound\_method=None, has\_reply=False, allow\_positional\_args=False*)

Marks a method as possible incoming message.

`@incoming` (*has\_reply=False, allow\_positional\_args=False*)

Incoming methods keep list of connected listeners, which are called with the signature of the incoming method (excluding `self`). The first argument will be passed positional and is a string describing the sender of the message. The remaining arguments can be chosen freely and will usually be passed as named args.

Optionally, you can receive security info (the `secinfo` dict extracted from the message). For this, call `myapi.<method>.pass_secinfo(True)`. Listener calls then receive an additional kwarg called `secinfo`, containing the received dictionary. I.e. your handler(s) must add a `secinfo=` parameter in addition to the signature specified in the `RemoteAPI`.

Listeners can be added with `myapi.<method>.connect(handler)` and disconnected with `.disconnect(handler)`. They are called in the order that they were added.

If `has_reply=True`, the handler should return a value that is sent back to the sender. If multiple handlers are connected, at most one of them must return something.

**Notice:** Processing of incoming messages does not resume until all listeners returned. This means that if you issue a followup remote call in a listener, the result can not arrive while the listener is executing. If you want to do this, use `promise.then()` to resume when the result is there.

You can also spawn a new thread in your listener, to do the processing. However, be aware that this makes you vulnerable against DOS attacks, since an attacker can make you open arbitrary many threads this way.

If `allow_positional_args=True`, messages with positional (unnamed) arguments are accepted. Otherwise such arguments throw an error message without executing the handler(s). Note that the *Codec* must support positional and/or mixed args as well. It is strongly recommended to use named args only.

Lastly, the incoming method has a `myapi.<method>.inverted()` method, which will return the `@outgoing` variant of it.

`quickrpc.outgoing` (*unbound\_method=None, has\_reply=False, allow\_positional\_args=False*)

Marks a method as possible outgoing message.

`@outgoing` (*has\_reply=False, allow\_positional\_args=False*)

Invocation of outgoing methods leads to a message being sent over the *Transport* of the *RemoteAPI*.

The first argument must be the list of receivers of the message, as a list of strings. When calling the method, usually you will use the sender name(s) received via an incoming call. Set `receivers=None` to send to all connected peers.

The remaining arguments can be choosen freely. The argument values can be anything supported by the *Codec* that you use. The builtin Codecs support all the “atomic” builtin types, as well as dicts and lists.

If `has_reply=True`, the other side is expected to return a result value. In this case, calling the outgoing method returns a *Promise* immediately.

If `allow_positional_args=True`, calls with positional (unnamed) arguments are accepted. Otherwise such arguments raise `ValueError`. **For sending, they will be converted into named arguments.** It is strongly recommended to use named args only.

Lastly, the outgoing method has a `myapi.<method>.inverted()` method, which will return the `@incoming` variant of it.

`quickrpc.transport` (*expression*)

Creates a transport from a given string expression.

The expression must be “<shorthand>:<specific parameters>”, with shorthand being the wanted transport’s .shorthand property. For the specific parameters, see the respective transport’s .fromstring method.

The base class implementation searches among all known subclasses for the Transport matching the given shorthand, and returns `Subclass.fromstring(expression)`.

`quickrpc.Codec(expression)`

Creates a codec from a given string expression.

The expression must be “<shorthand>:<specific parameters>”, with shorthand being the wanted Codec’s .shorthand property. For the specific parameters, see the respective Codec’s .fromstring method.

**exception** `quickrpc.RemoteError(message, details)`

Bases: `Exception`

**exception** ‘`quickrpc.RemoteError`’ undocumented

## 1.2 quickrpc.remote\_api module

A *RemoteAPI* is an interface-like class whose methods correspond to remote calls.

Let us start with an example:

```
from quickrpc.remote_api import RemoteAPI, incoming, outgoing

class MyAPI(RemoteAPI):
    @incoming
    def notify(self, sender, arg1=val1, arg2=val2):
        """notification that something happened"""

    @outgoing
    def helloworld(self, receivers, arg1=val1):
        """Tell everybody that I am here."""

    @incoming(has_reply=True)
    def echo(self, sender, text="test"):
        """returns the text that was sent."""
```

`RemoteAPI` is used by subclassing it. Remote methods are defined by the `@incoming` and `@outgoing` decorators.

---

**Important:** The method body of remote methods must be empty.

---

This is because by calling `@outgoing` methods, you actually issue a call over the *Transport* that is bound to the `RemoteAPI` at runtime. Since the API is meant to be used by both sides (by means of inverting it), `@incoming` methods should be empty, too. The side effect of this is that the class definition is more or less a printable specification of your interface.

`@incoming` methods have a `.connect()` method to attach an implementation to that message. The connected handler has the same signature as the `@incoming` method, except for the `self` argument.

By default, all defined calls are resultless (i.e. notifications). To define calls with return value, decorate with `has_reply=True` kwarg.

When handling such a call on the incoming side, your handler’s return value is returned to the sender. Exceptions are caught and sent as error reply.

On the outgoing side, the call immediately returns a *Promise* object. You then use `result()` to get at the actual result. This will block until the result arrived.

(TODO: make blocking call by default, add block=False param for Promises)

```
class quickrpc.remote_api.RemoteAPI(codec='jrpc', transport=None, security='null', invert=False, async_processing=False)
```

Bases: object

Describes an API i.e. a set of allowed outgoing and incoming calls.

Subclass and add your calls.

codec holds the Codec for (de)serializing data. *transport* holds the underlying transport. *security* holds the security provider.

Both can also be strings, then *Transport.fromstring()* / *Codec.fromstring()* are used to acquire the respective objects. In this case, transport still needs to be started via *myapi.transport.start()*.

Methods marked as @outgoing are automatically turned into messages when called. The method body is executed before sending. (use e.g. for validation of outgoing data). They must accept a special *receivers* argument, which is passed to the Transport.

Methods marked as @incoming are called by the transport when messages arrive. They work like signals - you can connect your own handler(s) to them. Connected handlers must have the same signature as the incoming call. All @incoming methods MUST support a *senders* argument.

Connect like this:

```
>>> def handler(self, foo=None): pass
>>> remote_api.some_method.connect(handler)
>>> # later
>>> remote_api.some_method.disconnect(handler)
```

Execution order: the method of remote\_api is executed first, then the connected handlers in the order of registering.

Incoming messages with unknown method will not be processed. If the message has *.id != 0*, it will automatically be replied with an error.

Threading:

- outgoing messages are sent on the calling thread.
- If *async\_processing = False*, incoming messages are handled on the thread which handles Transport receive events. I.e. the Transport implementation defines the behaviour.
- If *async\_processing = True*, an extra Thread is used to handle messages.

The latter allows the receive handler to run concurrently to message handling, allowing further requests to be sent out and to await the result. However it means one extra thread. In any case, only one incoming message is handled at a time.

Recommendation is to set *async\_processing=True* if there are any outgoing calls that have a reply, False if not.

Inverting:

You can *invert()* the whole api, swapping incoming and outgoing methods. When inverted, the *sender* and *receiver* arguments of each method swap their roles. This is also possible upon initialization by giving *invert=True* kwarg.

**invert()**

Swaps @incoming and @outgoing decoration on all methods of this INSTANCE.

I.e. generates the opposite-side API.

Do this before connecting any handlers to incoming calls.

You can achieve the same effect by instantiating with `invert=True` kwarg.

**message\_error** (*sender*, *exception*, *in\_reply\_to=None*)

Called each time that an incoming message causes problems.

By default, it logs the error as warning. `in_reply_to` is the message that triggered the error, `None` if decoding failed. If the requested method can be identified and has a reply, an error reply is returned to the sender.

**transport**

Gets/sets the transport used to send and receive messages.

You can change the transport at runtime.

**unhandled\_calls** ()

Generator, returns the names of all *incoming*, unconnected methods.

If no results are returned, all incoming messages are connected. Use this to check for missed `.connect` calls.

`quickrpc.remote_api.incoming` (*unbound\_method=None*, *has\_reply=False*, *allow\_positional\_args=False*)

Marks a method as possible incoming message.

`@incoming` (*has\_reply=False*, *allow\_positional\_args=False*)

Incoming methods keep list of connected listeners, which are called with the signature of the incoming method (excluding `self`). The first argument will be passed positional and is a string describing the sender of the message. The remaining arguments can be chosen freely and will usually be passed as named args.

Optionally, you can receive security info (the `secinfo` dict extracted from the message). For this, call `myapi.<method>.pass_secinfo(True)`. Listener calls then receive an additional kwarg called `secinfo`, containing the received dictionary. I.e. your handler(s) must add a `secinfo=` parameter in addition to the signature specified in the `RemoteAPI`.

Listeners can be added with `myapi.<method>.connect(handler)` and disconnected with `.disconnect(handler)`. They are called in the order that they were added.

If `has_reply=True`, the handler should return a value that is sent back to the sender. If multiple handlers are connected, at most one of them must return something.

**Notice:** Processing of incoming messages does not resume until all listeners returned. This means that if you issue a followup remote call in a listener, the result can not arrive while the listener is executing. If you want to do this, use `promise.then()` to resume when the result is there.

You can also spawn a new thread in your listener, to do the processing. However, be aware that this makes you vulnerable against DOS attacks, since an attacker can make you open arbitrary many threads this way.

If `allow_positional_args=True`, messages with positional (unnamed) arguments are accepted. Otherwise such arguments throw an error message without executing the handler(s). Note that the *Codec* must support positional and/or mixed args as well. It is strongly recommended to use named args only.

Lastly, the incoming method has a `myapi.<method>.inverted()` method, which will return the `@outgoing` variant of it.

`quickrpc.remote_api.outgoing` (*unbound\_method=None*, *has\_reply=False*, *allow\_positional\_args=False*)

Marks a method as possible outgoing message.

`@outgoing` (*has\_reply=False*, *allow\_positional\_args=False*)

Invocation of outgoing methods leads to a message being sent over the *Transport* of the *RemoteAPI*.

The first argument must be the list of receivers of the message, as a list of strings. When calling the method, usually you will use the sender name(s) received via an incoming call. Set `receivers=None` to send to all connected peers.

The remaining arguments can be chosen freely. The argument values can be anything supported by the *Codec* that you use. The builtin Codecs support all the “atomic” builtin types, as well as dicts and lists.

If `has_reply=True`, the other side is expected to return a result value. In this case, calling the outgoing method returns a *Promise* immediately.

If `allow_positional_args=True`, calls with positional (unnamed) arguments are accepted. Otherwise such arguments raise `ValueError`. **For sending, they will be converted into named arguments.** It is strongly recommended to use named args only.

Lastly, the outgoing method has a `myapi.<method>.inverted()` method, which will return the `@incoming` variant of it.

## 1.3 quickrpc.transports, .network\_transports and .QtTransports modules

### 1.3.1 quickrpc.transports module

A Transport abstracts a two-way bytestream interface.

It can be started and stopped, and send and receive byte sequences to one or more receivers.

**Classes defined here:**

- *Transport*: abstract base
- *MuxTransport*: a transport that multiplexes several sub-transports.
- *RestartingTransport*: a transport that automatically restarts its child.
- *StdioTransport*: reads from stdin, writes to stdout.
- *TcpServerTransport*: a transport that accepts tcp connections and muxes them into one transport. Actually a forward to `quickrpc.network_transports`.
- *TcpClientTransport*: connects to a TCP server. This is a forward to `quickrpc.network_transports`.
- *RestartingTcpClientTransport*: a TCP Client that reconnects automatically.

**class** `quickrpc.transports.Transport`

Bases: `object`

A transport abstracts a two-way bytestream interface.

This is the base class, which provides multithreading logic but no actual communication channel.

In a subclass, the following methods must be implemented:

- *send()* to send outgoing messages
- *open()* to initialize the channel (if necessary)
- *run()* to receive messages until it is time to stop.

Incoming messages are passed to a callback. It must be set before the first message arrives via *set\_on\_received()*.

Provided threading functionality:

- `start()` opens and runs the channel in a new thread
- `stop()` signals `run()` to stop, by setting `running` to `False`.

The classmethod `fromstring()` can be used to create a `Transport` instance from a string (for enhanced configurability).

**classmethod `fromstring`** (*expression*)

Creates a transport from a given string expression.

The expression must be “<shorthand>:<specific parameters>”, with shorthand being the wanted transport’s `.shorthand` property. For the specific parameters, see the respective transport’s `.fromstring` method.

The base class implementation searches among all known subclasses for the Transport matching the given shorthand, and returns `Subclass.fromstring(expression)`.

**`open()`**

Open the communication channel. e.g. bind and activate a socket.

Override me.

`open` is called on the new thread opened by `start`. I.e. the same thread in which the Transport will `run`.

When `open()` returns, the communication channel should be ready for send and receive.

**`received`** (*sender, data*)

To be called by `run()` when the subclass received data.

`sender` is a unique string identifying the source. e.g. IP address and port.

If the given data has an undecodable “tail”, it is returned. In this case `run()` must prepend the tail to the next received bytes from this channel, because it is probably an incomplete message.

**`receiver_thread`**

The thread on which `on_received` will be called.

**`run()`**

Runs the transport, blocking.

Override me.

This contains the transport’s mainloop, which must:

- receive bytes from the channel (usually blocking)
- pass the bytes to `self.received`
- check periodically (e.g. each second) if `self.running` has been cleared
- if so, close the channel and return.

**`send`** (*data, receivers=None*)

Sends the given data to the specified receiver(s).

`receivers` is an iterable yielding strings. `receivers=None` sends the data to all connected peers.

TODO: specify behaviour when sending on a stopped or failed Transport.

**`set_on_received`** (*on\_received*)

Sets the function to call upon receiving data.

The callback’s signature is `on_received(sender, data)`, where `sender` is a string describing the origin; `data` is the received bytes. If decoding leaves trailing bytes, they should be returned. The Transport stores them and prepends them to the next received bytes.

**`shorthand`** = ''

**start** (*block=True, timeout=10*)

Run in a new thread.

If *block* is *True*, waits until startup is complete i.e. *open()* returns. Then returns *True*.

if *nonblocking*, returns a promise.

If something goes wrong during start, the Exception, like e.g. a *socket.error*, is passed through to the caller.

**stop** (*block=True*)

Stop running transport (possibly from another thread).

Resets *running* to signal to *run()* that it should stop.

Actual stopping can take a moment. If *block* is *True*, *stop()* waits until *run()* returns.

**class** quickrpc.transports.MuxTransport

Bases: *quickrpc.transports.Transport*

A transport that muxes several transports.

Incoming data is serialized into the thread of *MuxTransport.run()*.

Add Transports via *mux\_transport += transport*. Remove via *mux\_transport -= transport*.

Adding a transport changes its *on\_received* binding to the mux transport. If *MuxTransport* is already running, the added transport is *start()*ed by default.

Removing a transport *stop()*s it by default.

Running/Stopping the *MuxTransport* also runs/stops all muxed transports.

**add\_transport** (*transport, start=True*)

add and start the transport (if running).

**classmethod** **fromstring** (*expression*)

*mux:(<transport1>)(<transport2>)...*

where *<transport1>*, .. are again valid transport expressions.

**handle\_received** (*sender, data*)

handles INCOMING data from any of the muxed transports. *b''* is returned as leftover ALWAYS; *MuxTransport* keeps internal remainder buffers for all senders, since the leftover is only available after the message was processed.

**open** ()

Start all transports that were added so far.

The subtransports are started in parallel, then we wait until all of them are up.

If any transport fails to start, all transports are stopped again, and *TransportError* is raised. It will have a *.exceptions* attribute being a list of all failures.

**remove\_transport** (*transport, stop=True*)

remove and stop the transport.

**run** ()

Runs the transport, blocking.

Override me.

This contains the transport's mainloop, which must:

- receive bytes from the channel (usually blocking)
- pass the bytes to *self.receive*

- check periodically (e.g. each second) if `self.running` has been cleared
- if so, close the channel and return.

**send** (*data*, *receivers=None*)

Sends the given data to the specified receiver(s).

*receivers* is an iterable yielding strings. *receivers=None* sends the data to all connected peers.

TODO: specify behaviour when sending on a stopped or failed Transport.

**shorthand** = 'mux'

**stop** ()

Stop running transport (possibly from another thread).

Resets *running* to signal to *run* () that it should stop.

Actual stopping can take a moment. If *block* is True, *stop* () waits until *run* () returns.

**class** quickrpc.transports.RestartingTransport (*transport*, *check\_interval=10*, *name=""*)

Bases: *quickrpc.transports.Transport*

A transport that wraps another transport and keeps restarting it.

E.g. you can wrap a TcpClientTransport to try reconnecting it.

```
>>> tr = RestartingTransport(TcpClientTransport(*address), check_interval=10)
```

*check\_interval* gives the Restart interval in seconds. It may not be kept exactly. It cannot be lower than 1 second. Restarting is attempted as long as the transport is running.

Adding a transport changes its *on\_received* handler to the RestartingTransport.

**classmethod** fromstring (*expression*)

restart:10:<subtransport>

10 (seconds) is the restart interval.

<subtransport> is any valid transport string.

**open** ()

Open the communication channel. e.g. bind and activate a socket.

Override me.

*open* is called on the new thread opened by *start*. I.e. the same thread in which the Transport will *run*.

When *open* () returns, the communication channel should be ready for send and receive.

**receiver\_thread**

Thread on which *receive*() is called - in this case, *receiver\_thread* of the child.

**run** ()

Runs the transport, blocking.

Override me.

This contains the transport's mainloop, which must:

- receive bytes from the channel (usually blocking)
- pass the bytes to *self.received*
- check periodically (e.g. each second) if *self.running* has been cleared
- if so, close the channel and return.



**send** (*data*, *receivers=None*)

Sends the given data to the specified receiver(s).

*receivers* is an iterable yielding strings. *receivers=None* sends the data to all connected peers.

TODO: specify behaviour when sending on a stopped or failed Transport.

**shorthand** = 'restart'

**stop** ()

Stop running transport (possibly from another thread).

Resets running to signal to *run* () that it should stop.

Actual stopping can take a moment. If *block* is True, *stop* () waits until *run* () returns.

**subtransport\_running**

True if the child transport is currently running.

**class** quickrpc.transports.StdioTransport

Bases: *quickrpc.transports.Transport*

**class** 'quickrpc.transports.StdioTransport' undocumented

**classmethod** fromstring (*expression*)

No configuration options, just use "stdio".

**run** ()

run, blocking.

**send** (*data*, *receivers=None*)

Sends the given data to the specified receiver(s).

*receivers* is an iterable yielding strings. *receivers=None* sends the data to all connected peers.

TODO: specify behaviour when sending on a stopped or failed Transport.

**shorthand** = 'stdio'

**stop** ()

Stop running transport (possibly from another thread).

Resets running to signal to *run* () that it should stop.

Actual stopping can take a moment. If *block* is True, *stop* () waits until *run* () returns.

quickrpc.transports.TcpServerTransport (*port*, *interface=""*, *announcer=None*)

**function** 'quickrpc.transports.TcpServerTransport' undocumented

quickrpc.transports.TcpClientTransport (*host*, *port*)

**function** 'quickrpc.transports.TcpClientTransport' undocumented

quickrpc.transports.RestartingTcpClientTransport (*host*, *port*, *check\_interval=10*)

Convenience wrapper for the most common use case. Returns TcpClientTransport wrapped in a Restarting-Transport.

### 1.3.2 quickrpc.network\_transports module

**module** 'quickrpc.network\_transports' undocumented

**class** quickrpc.network\_transports.UdpTransport (*port*)

Bases: *quickrpc.transports.Transport*

transport that communicates over UDP datagrams.

Connectionless - sender/receiver are IP addresses. Sending and receiving is done on the same port. Sending with receiver=None makes a broadcast.

Use messages > 500 Bytes at your own peril.

**classmethod fromstring** (*expression*)

udp:1234 - the number being the port for send/receive.

**open** ()

Open the communication channel. e.g. bind and activate a socket.

Override me.

open is called on the new thread opened by *start*. I.e. the same thread in which the Transport will run.

When open () returns, the communication channel should be ready for send and receive.

**run** ()

Runs the transport, blocking.

Override me.

This contains the transport's mainloop, which must:

- receive bytes from the channel (usually blocking)
- pass the bytes to `self.received`
- check periodically (e.g. each second) if `self.running` has been cleared
- if so, close the channel and return.

**send** (*data*, *receivers=None*)

Sends the given data to the specified receiver(s).

*receivers* is an iterable yielding strings. *receivers=None* sends the data to all connected peers.

TODO: specify behaviour when sending on a stopped or failed Transport.

**shorthand** = 'udp'

```
class quickrpc.network_transports.TcpServerTransport (port, interface="",  
                                                    announcer=None,  
                                                    keepalive_msg=b",  
                                                    keepalive_interval=10, buffer-  
                                                    size=1024)
```

Bases: `quickrpc.transports.MuxTransport`

transport that accepts TCP connections as transports.

Basically a mux transport coupled with a TcpServer. Each time somebody connects, the connection is wrapped into a transport and added to the muxer.

There is (for now) no explicit notification about connects/disconnects; use the API for that.

Use `.close()` for server-side disconnect.

You can optionally pass an announcer (as returned by `announcer_api.make_udp_announcer`). It will be started/stopped together with the TcpServerTransport.

Optionally, a keepalive message can be configured. On each connection, `keepalive_msg` is sent verbatim every `keepalive_interval` seconds while the connection is idle. Any sending or receiving resets the timer. You can change the attributes directly while transport is stopped.

**Threads:**

- `TcpServerTransport.run()` blocks (use `.start()` for automatic extra Thread)

- `.run()` starts a new thread for listening to connections
- each incoming connection will start another Thread.

**close** (*name*)

close the connection with the given sender/receiver name.

**classmethod fromstring** (*expression*)

tcp: <interface>: <port>

Leave <interface> empty to listen on all interfaces.

**open** ()

Start all transports that were added so far.

The subtransports are started in parallel, then we wait until all of them are up.

If any transport fails to start, all transports are stopped again, and `TransportError` is raised. It will have a `.exceptions` attribute being a list of all failures.

**run** ()

Runs the transport, blocking.

Override me.

This contains the transport's mainloop, which must:

- receive bytes from the channel (usually blocking)
- pass the bytes to `self.received`
- check periodically (e.g. each second) if `self.running` has been cleared
- if so, close the channel and return.

**shorthand** = 'tcp:serv'

```
class quickrpc.network_transports.TcpClientTransport (host, port, connect_timeout=10,
                                                         keepalive_msg=b'',
                                                         keepalive_interval=10, buffer_size=1024)
```

Bases: `quickrpc.transports.Transport`

Transport that connects to a TCP server.

Optionally, a keepalive message can be configured. `keepalive_msg` is sent verbatim every `keepalive_interval` seconds while the connection is idle. Any sending or receiving resets the timer. You can change the attributes anytime.

**classmethod fromstring** (*expression*)

tcp: <host>: <port>

**open** ()

Open the communication channel. e.g. bind and activate a socket.

Override me.

`open` is called on the new thread opened by `start`. I.e. the same thread in which the Transport will `run`.

When `open ()` returns, the communication channel should be ready for send and receive.

**run** ()

run, blocking.

**send** (*data*, *receivers=None*)

Sends the given data to the specified receiver(s).

*receivers* is an iterable yielding strings. *receivers=None* sends the data to all connected peers.

TODO: specify behaviour when sending on a stopped or failed Transport.

**shorthand** = 'tcp'

### 1.3.3 quickrpc.QtTransports module

transports based on Qt communication classes, running in the Qt event loop.

**class** quickrpc.QtTransports.QProcessTransport (*cmdline*, *sendername='qprocess'*)

Bases: *quickrpc.transports.Transport*

A Transport communicating with a child process.

Start the process using *.start()*.

Sent data is written to the process' stdin.

Data is received from the process's stdout and processed on the Qt mainloop thread.

**classmethod** **fromstring** (*expression*)

qprocess:<commandline>

**on\_finished** ()

method 'quickrpc.QtTransports.QProcessTransport.on\_finished' undocumented

**on\_ready\_read** ()

method 'quickrpc.QtTransports.QProcessTransport.on\_ready\_read' undocumented

**send** (*data*, *receivers=None*)

Sends the given data to the specified receiver(s).

*receivers* is an iterable yielding strings. *receivers=None* sends the data to all connected peers.

TODO: specify behaviour when sending on a stopped or failed Transport.

**shorthand** = 'qprocess'

**start** ()

Run in a new thread.

If *block* is True, waits until startup is complete i.e. *open()* returns. Then returns True.

if *nonblocking*, returns a promise.

If something goes wrong during start, the Exception, like e.g. a *socket.error*, is passed through to the caller.

**stop** (*kill=False*)

Stop running transport (possibly from another thread).

Resets *running* to signal to *run()* that it should stop.

Actual stopping can take a moment. If *block* is True, *stop()* waits until *run()* returns.

**class** quickrpc.QtTransports.QTcpTransport (*host*, *port*, *sendername='qtcp'*)

Bases: *quickrpc.transports.Transport*

A Transport connecting to a TCP server.

Connect using *.start()*.

Received data is processed on the Qt mainloop thread.

**classmethod** `fromstring (expression)`

qtcp:<host>:<port>

**on\_connect ()**

method ‘quickrpc.QtTransports.QTcpTransport.on\_connect’ undocumented

**on\_error (error)**

method ‘quickrpc.QtTransports.QTcpTransport.on\_error’ undocumented

**on\_ready\_read ()**

method ‘quickrpc.QtTransports.QTcpTransport.on\_ready\_read’ undocumented

**send (data, receivers=None)**

Sends the given data to the specified receiver(s).

`receivers` is an iterable yielding strings. `receivers=None` sends the data to all connected peers.

TODO: specify behaviour when sending on a stopped or failed Transport.

**shorthand = 'qtcp'**

**start ()**

Run in a new thread.

If `block` is `True`, waits until startup is complete i.e. `open ()` returns. Then returns `True`.

if `nonblocking`, returns a promise.

If something goes wrong during start, the Exception, like e.g. a `socket.error`, is passed through to the caller.

**stop ()**

Stop running transport (possibly from another thread).

Resets `running` to signal to `run ()` that it should stop.

Actual stopping can take a moment. If `block` is `True`, `stop ()` waits until `run ()` returns.

## 1.4 quickrpc.codecs and .terse\_codec modules

### 1.4.1 quickrpc.codecs module

Codecs convert message structures into bytes and vice versa.

**Classes defined here:**

- Codec: base class
- Message, DecodeError

**class** `quickrpc.codecs.Codec`

Bases: `object`

Responsible for serializing and deserializing method calls.

Subclass and override `encode`, `decode`, optionally `encode_reply`, `encode_error`.

*Protocol overview*

Byte-data payload is generated from python data by using:

- `encode` for “regular” messages / requests

- `encode_reply` for return data
- `encode_error` for error return data.

Python data is retrieved from bytes by `decode`. This returns a list of objects, which can be instances of `Message`, `Reply` and `ErrorReply`.

### Security

Let *payload* denote the “inner” message data and *frame* the message going on the wire, both being byte sequences. `encode*()` can be given a `sec_out()` callback, taking the payload data and returning `(secinfo, new_payload)`.

`secinfo` is a dict containing e.g. user info, signature, etc. (specific of Security provider).

`new_payload` is an optional transformed payload (bytes), e.g. encrypted data. If omitted, use original payload. `encode*()` then builds a frame using new payload and `secinfo` data, e.g. add crypt headers.

Depending on protocol, `encode` could be downwards-compatible if “guest” security applies i.e. `secinfo` is empty and payload stays untransformed.

Decoding: `decode` again takes a `sec_in()` callback, accepting security info and payload data, returning the “unpacked” payload. E.g. `secinfo` could check the signature and raise an error if the message was forged. The `secinfo` dictionary is returned within the `Message`, `Reply` or `ErrorReply` object.

**decode** (*data*, *sec\_in=None*)

decode data to method call with kwargs.

Return: [messages], remainder where [messages] is the list of decoded messages and remainder is leftover data (which may contain the beginning of another message).

If a message cannot be decoded properly, an exception is added in the message list. Decode should never *raise* an error, because in this case the remaining data cannot be retrieved.

**messages can be instances of:**

- `Message`
- `Reply` (to the previous message with the same id)
- `ErrorReply` (to the previous message with the same id)

**Message attributes** `.method` attribute (string), `.kwargs` attribute (dict), `.id`, `.secinfo` (dict)

**Reply attributes** `.result`, `.id`, `.secinfo` (dict)

**ErrorReply attributes** `.exception`, `.id`, `.errorcode`, `.secinfo` (dict)

**encode** (*method*, *kwargs=None*, *id=0*, *sec\_out=None*)

encode a method call with given kwargs.

`sec_out` callback parameters:

- `payload` (bytes): Payload data for the frame.

`sec_out` returns `(secinfo, new_payload)`:

- `sec_info` (dict): security information, dictionary str->str, keys defined by Security provider.
- `new_payload` (bytes): transformed payload; `None` indicates that original payload can be used.

Returns: frame data (bytes)

**encode\_error** (*in\_reply\_to*, *exception*, *errorcode=0*, *sec\_out=None*)

encode error caused by the given `Message`.

**encode\_reply** (*in\_reply\_to*, *result*, *sec\_out=None*)  
encode reply to the Message

**classmethod fromstring** (*expression*)  
Creates a codec from a given string expression.

The expression must be “<shorthand>:<specific parameters>”, with shorthand being the wanted Codec’s .shorthand property. For the specific parameters, see the respective Codec’s .fromstring method.

**shorthand** = ''

**exception** quickrpc.codecs.DecodeError  
Bases: Exception

**exception** 'quickrpc.codecs.DecodeError' undocumented

**exception** quickrpc.codecs.EncodeError  
Bases: Exception

**exception** 'quickrpc.codecs.EncodeError' undocumented

**class** quickrpc.codecs.Message (*method*, *kwargs*, *id=0*, *secinfo=None*)  
Bases: object

**class** 'quickrpc.codecs.Message' undocumented

**class** quickrpc.codecs.Reply (*result*, *id*, *secinfo=None*)  
Bases: object

**class** 'quickrpc.codecs.Reply' undocumented

**class** quickrpc.codecs.ErrorReply (*exception*, *id*, *errorcode=0*, *secinfo=None*)  
Bases: object

**class** 'quickrpc.codecs.ErrorReply' undocumented

**exception** quickrpc.codecs.RemoteError (*message*, *details*)  
Bases: Exception

**exception** 'quickrpc.codecs.RemoteError' undocumented

**class** quickrpc.codecs.JsonRpcCodec (*delimiter=b'x00'*)  
Bases: *quickrpc.codecs.Codec*

Json codec: convert to json

bytes values are converted into a an object containing the single key `__bytes` with value being base64-encoded data.

If security is used, the following “Authenticated-JSON-RPC” protocol applies:

*Encoding*

Prepend a special, valid json-rpc message before the payload:

```
{"jsonrpc": "2.0", "method": "rpc.secinfo", "params":  
<secinfo>}<DELIM><payload><DELIM>
```

If secinfo is empty, NOTHING is prepended (i.e. behaves like unextended JSON-RPC)

---

**Note:** Payload must not contain the delimiter even if it is encrypted. Raw data could be b64-encoded. If payload is encrypted, basic-JSON-RPC compatibility is of course lost.

---

*Decoding with security*

Decode delimited messages one-by-one as usual (“one” being the bytes between delimiters).

If a `rpc.secinfo` call is detected, take the unaltered payload from the next message, giving `secinfo` and payload. If next message is incomplete (no trailing delim), throw the `rpc.secinfo` message back into the remainder.

For regular call (method `!= rpc.secinfo`), return the message itself as payload with empty `secinfo`.

*Discussion:*

- allows framing without touching payload :-)
- allows decoding the header without decoding payload :-)
- allows using byte-payload as is, particularly allows encrypted+literal payload to coexist (however encrypted payload breaks JSON-RPC compat!) :-)
- Msg to “unaware” peer: will throw the `rpc.secinfo` calls away silently or loudly, but is able to operate. Missing ID indicates a notification, i.e. peer will not send response back per JSON-RPC spec. :-)
- Msg from “unaware” peer: will implicitly be treated as no-security message.

**decode** (*data*, *sec\_in=None*)

decode data to method call with kwargs.

Return: [messages], remainder where [messages] is the list of decoded messages and remainder is leftover data (which may contain the beginning of another message).

If a message cannot be decoded properly, an exception is added in the message list. Decode should never *raise* an error, because in this case the remaining data cannot be retrieved.

**messages can be instances of:**

- Message
- Reply (to the previous message with the same id)
- ErrorReply (to the previous message with the same id)

**Message attributes** `.method` attribute (string), `.kwargs` attribute (dict), `.id`, `.secinfo` (dict)

**Reply attributes** `.result`, `.id`, `.secinfo` (dict)

**ErrorReply attributes** `.exception`, `.id`, `.errorcode`, `.secinfo` (dict)

**encode** (*method*, *kwargs*, *id=0*, *sec\_out=None*)

encode a method call with given kwargs.

`sec_out` callback parameters:

- payload (bytes): Payload data for the frame.

`sec_out` returns (`secinfo`, `new_payload`):

- `sec_info` (dict): security information, dictionary str->str, keys defined by Security provider.
- `new_payload` (bytes): transformed payload; `None` indicates that original payload can be used.

Returns: frame data (bytes)

**encode\_error** (*in\_reply\_to*, *exception*, *errorcode=0*, *sec\_out=None*)

encode error caused by the given Message.

**encode\_reply** (*in\_reply\_to*, *result*, *sec\_out=None*)

encode reply to the Message



```
classmethod fromstring (expression)
    jrpc:delimiter
```

delimiter is the character splitting the telegrams and must not occur within any telegram. Default = <null>.

```
shorthand = 'jrpc'
```

## 1.4.2 quickrpc.terse\_codec module

**module 'quickrpc.terse\_codec' undocumented**

```
quickrpc.terse_codec.L()
```

```
function 'quickrpc.terse_codec.L' undocumented
```

```
class quickrpc.terse_codec.TerseCodec
```

Bases: *quickrpc.codecs.Codec*

Terse codec: encodes with minimum punctuation.

encodes to: method[id] param1:1, param2:"foo"<NL> values:

- int/float: 1.0
- bytes: '(base64-string'
- str: "python-escaped str"
- list: [val1 val2 val3 ...]
- dict: {key1:val1 key2:val2 ... }

Reply is encoded to: [id]:value Error is encoded to: [id]! message:"string" details:"string"

- Commands must be terminated by newline.
- Newlines, double quote and backslash in strings are escaped as usual
- Allowed dtypes: int, float, str, bytes (content base64-encoded), list, dict

```
decode (data, sec_in=None)
```

decode data to method call with kwargs.

Return: [messages], remainder where [messages] is the list of decoded messages and remainder is leftover data (which may contain the beginning of another message).

If a message cannot be decoded properly, an exception is added in the message list. Decode should never *raise* an error, because in this case the remaining data cannot be retrieved.

**messages can be instances of:**

- Message
- Reply (to the previous message with the same id)
- ErrorReply (to the previous message with the same id)

**Message attributes** .method attribute (string), .kwargs attribute (dict), .id, .secinfo (dict)

**Reply attributes** .result, .id, .secinfo (dict)

**ErrorReply attributes** .exception, .id, .errorcode, .secinfo (dict)

```
encode (method, kwargs, id=0, sec_out=None)
```

encodes the call, including trailing newline

```
encode_error (in_reply_to, exception, errorcode=0, sec_out=None)
```

encode error caused by the given Message.

```
encode_reply (in_reply_to, result, sec_out=None)  
    encode reply to the Message  
  
classmethod fromstring (expression)  
    terse:  
  
shorthand = 'terse'
```

## 1.5 quickrpc.security module

Security Providers.

```
exception quickrpc.security.SecurityError  
    Bases: Exception  
  
    Security-related error
```

```
exception quickrpc.security.InvalidSignatureError  
    Bases: quickrpc.security.SecurityError  
  
    Received message had an invalid signature
```

```
exception quickrpc.security.UnknownUserError  
    Bases: quickrpc.security.SecurityError  
  
    User account not found
```

```
class quickrpc.security.Security  
    Bases: object  
  
    Base class for Security providers.
```

A security provider has `sec_in` and `sec_out` methods, which are used to process inbound and outbound messages, respectively.

Apart from that, it is up to the security provider what it does to the messages and how it manages authentication.

```
classmethod fromstring (expression)  
    Creates a security instance from a given string expression.
```

The expression must be “<shorthand>:<specific parameters>”, with shorthand being the wanted Security’s `.shorthand` property. For the specific parameters, see the respective Security’s `.fromstring` method.

```
sec_in (payload, secinfo)  
    Secure an inbound message.
```

### Parameters

- **payload** (*bytes*) – Payload as received.
- **secinfo** (*dict of str -> str*) – Security headers as received.

**Returns** `new_payload` (*bytes*); `None` indicates that received payload can be used.

The provider can e.g. decrypt the payload and or check the signature (raising an exception on failure).

```
sec_out (payload)  
    Secure an outbound message.
```

**Parameters** **payload** (*bytes*) – Payload data for the frame.

Returns (`secinfo`, `new_payload`):

- `secinfo` (dict): security information, dictionary str->str
- `new_payload` (bytes): transformed payload; `None` indicates that original payload can be used.

`secinfo` can contain arbitrary keys specified by the subclass.

The provider can e.g. calculate a signature and/or encrypt the payload.

```
shorthand = ''
```

```
class quickrpc.security.NullSecurity
```

Bases: `quickrpc.security.Security`

no security added, no user management at all (anonymous communication).

Default if nothing is specified.

```
classmethod fromstring (expression)
```

Creates a security instance from a given string expression.

The expression must be “<shorthand>:<specific parameters>”, with shorthand being the wanted Security’s `.shorthand` property. For the specific parameters, see the respective Security’s `.fromstring` method.

```
sec_in = None
```

```
sec_out = None
```

```
shorthand = 'null'
```

```
class quickrpc.security.NoSecurity (user="")
```

Bases: `quickrpc.security.Security`

Provides transmission of a username, without any checking.

There is no validation or message integrity checking.

Only use this if you absolutely trust each communication endpoint. ... Actually, please don’t.

To specify username for outbound messages, set the `user` attribute.

```
classmethod fromstring (expression)
```

Creates a security instance from a given string expression.

The expression must be “<shorthand>:<specific parameters>”, with shorthand being the wanted Security’s `.shorthand` property. For the specific parameters, see the respective Security’s `.fromstring` method.

```
sec_in (payload, secinfo)
    does nothing
```

```
sec_out (payload)
    return self.user as username.
```

```
shorthand = 'blindly_believe_everything'
```

## 1.6 quickrpc.promise and .action\_queue module

These are tools for async programming.

A *Promise* (also known as a Deferred or a Future) is like an order slip for something that is still being produced.

An *ActionQueue* is a background worker that manages its own worker thread automatically.

### 1.6.1 quickrpc.promise module

Defines a basic *Promise* class.

A Promise (also known as a Deferred or a Future) is like an order slip for something that is still being produced.

This is just a barebone implementation, with method names aligned with `concurrent.Future` from the standard lib.

**class** `quickrpc.promise.Promise` (*setter\_thread=None*)

Bases: `object`

Encapsulates a result that will arrive later.

A Promise (also known as a Deferred or a Future) is like an order slip for something that is still being produced.

Promises are dispensed by asynchronous functions. Calling `.result()` waits until the operation is complete, then returns the result.

You can also use `.then(callback)` to have the promise call you with the result.

The constructor takes an argument `setter_thread`, which should be the thread that will set the result later. If not given, the current thread is assumed (which will usually be the case). The `setter_thread` is used to provide basic deadlock protection.

**result** (*timeout=1.0*)

Return the result, waiting for it if necessary.

If the promise failed, this will raise the exception that the issuer gave.

If the promise is still unfulfilled after the *timeout* (in seconds) elapsed, `PromiseTimeoutError` is raised.

If the promise is unfulfilled and the calling thread is the designated promise-setter thread, `PromiseDeadlockError` is raised immediately.

**set\_exception** (*exception*)

called by the promise issuer to indicate failure.

**set\_result** (*val*)

called by the promise issuer to set the result.

**then** (*callback, errback=None*)

set handler to run as soon as the result is set.

callback takes the result as single argument.

You can also set an errback that is called in case of an exception. If not set, the exception will be passed to callback as result.

If the result already arrived, callback or errback is called immediately.

**exception** `quickrpc.promise.PromiseError`

Bases: `Exception`

promise-related error

**exception** `quickrpc.promise.PromiseTimeoutError`

Bases: `quickrpc.promise.PromiseError`, `TimeoutError`

waiting for the promise took too long.

**exception** `quickrpc.promise.PromiseDoneError`

Bases: `quickrpc.promise.PromiseError`, `RuntimeError`

raised to the promise issuer if a result or exception was already set.

**exception** `quickrpc.promise.PromiseDeadlockError`  
Bases: `quickrpc.promise.PromiseError`, `RuntimeError`  
raised if the result-setter thread tries to wait for the result (i.e. itself).

## 1.6.2 quickrpc.action\_queue module

ActionQueue: a background worker that manages its own worker thread automatically.

**class** `quickrpc.action_queue.ActionQueue`  
Bases: `object`  
A background worker that manages its own worker thread automatically.  
Enqueue work items using `.put()`. Work items are functions that do not take any parameters and return `None`.  
`.put()` returns immediately. The work items are processed in a background thread, in the order in which they arrived. Only one work item is processed at a time.  
The background thread is started when there is work to do, and teared down when the queue is empty.  
**put** (*action*)  
Put an action into the queue.  
**Parameters** *action* (*func*) – a callable without params. The return value is not used.

## 1.7 Other modules

### 1.7.1 quickrpc.announcer\_api module

**module** ‘`quickrpc.announcer_api`’ undocumented

**class** `quickrpc.announcer_api.AnnouncerAPI` (*codec='jrpc', transport=None, security='null', invert=False, async\_processing=False*)  
Bases: `quickrpc.remote_api.RemoteAPI`  
AnnouncerAPI provides a means of discovering services in some kind of distributed system.  
“Clients” broadcast a seek call. All “servers” who feel spoken to respond with an advertise call to the seeker.  
**advertise** (*receivers=None, description=""*)  
Advertise that I am present.  
Usually the advertisement should be sent only to the seeker, in return of seek().  
description is a to-be-defined expression or structure giving details about service type, version, etc.  
**seek** (*sender, filter=""*)  
Request advertising of present services.  
filter is a to-be-defined expression or data structure specifying the services that are wanted.  
`quickrpc.announcer_api.make_announcer` (*transport, description="", filter\_func=None, codec=<quickrpc.terse\_codec.TerseCodec object>*)  
Returns a ready-to-use announcer server running over the given transport.  
Sets the transport’s API.  
description is the service description to hand out.

`filter_func` is a predicate accepting the *filter* parameter of `AnnouncerAPI.seek` and returning `True` if the filter matches this service. If left out, it is assumed to be always `True`.

All you need to do afterwards is to call `transport.start()`. Keep a reference to the transport.

```
quickrpc.announcer_api.make_udp_announcer(port, description="", filter_func=None,
                                           codec=<quickrpc.terse_codec.TerseCodec
                                           object>)
```

makes an announcer using `UdpTransport(port)` and returns it.

Start/stop with `announcer.transport.start() / .stop()`.

## 1.7.2 quickrpc.echo\_api module

EchoAPI: simple chat server.

Demonstrates use of `RemoteAPI` as well as `StdioTransport` and `TcpServerTransport`.

The server responds to a “say” call with “echo” of the text to all clients. The message “quit”:

- if coming from stdio, shuts the server down
- if coming over a tcp connection, makes the server close the connection.

Run with `python3 -m quickrpc.echo_api`. Enter json messages on the commandline to test stdio transport. Use `telnet localhost 8888` to test tcp functionality. Use `tail -F echo_api.log` in another terminal to watch logged events.

```
class quickrpc.echo_api.EchoAPI(codec='jrpc', transport=None, security='null', invert=False,
                                async_processing=False)
```

Bases: `quickrpc.remote_api.RemoteAPI`

Demo of how to use `RemoteAPI`.

Echo API answers incoming *say* calls with an *echo* call.

```
echo (receivers=None, text="")
    method 'quickrpc.echo_api.EchoAPI.echo' undocumented

quit (sender="")
    method 'quickrpc.echo_api.EchoAPI.quit' undocumented

say (sender="", text="")
    method 'quickrpc.echo_api.EchoAPI.say' undocumented
```

```
quickrpc.echo_api.L()
    function 'quickrpc.echo_api.L' undocumented
```

```
quickrpc.echo_api.test()
    function 'quickrpc.echo_api.test' undocumented
```

## 1.7.3 quickrpc.util module

**module** 'quickrpc.util' undocumented

```
quickrpc.util.subclasses(cls)
    function 'quickrpc.util.subclasses' undocumented
```

```
quickrpc.util.paren_partition(text)
    pop first parenthesized expression from a string.
```

The text must start with one of (`[<`. The function finds the matching closing paren, then returns a tuple of (`paren_content`, `paren`, `rest`):

- paren\_content is the text in parens, without the actual parens
- paren is the opening paren
- rest is what comes after the closing paren.

```
>>> paren_partition('(a (contrived) example) (foo)bar')  
('a (contrived) example', '(', '(foo)bar')
```





## CHAPTER 2

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### q

- `quickrpc`, [3](#)
- `quickrpc.action_queue`, [25](#)
- `quickrpc.announcer_api`, [25](#)
- `quickrpc.codecs`, [17](#)
- `quickrpc.echo_api`, [26](#)
- `quickrpc.network_transports`, [13](#)
- `quickrpc.promise`, [24](#)
- `quickrpc.QtTransports`, [16](#)
- `quickrpc.remote_api`, [6](#)
- `quickrpc.security`, [22](#)
- `quickrpc terse_codec`, [21](#)
- `quickrpc.transports`, [9](#)
- `quickrpc.util`, [26](#)



## A

ActionQueue (class in quickrpc.action\_queue), 25  
 add\_transport() (quickrpc.transports.MuxTransport method), 11  
 advertise() (quickrpc.announcer\_api.AnnouncerAPI method), 25  
 AnnouncerAPI (class in quickrpc.announcer\_api), 25

## C

close() (quickrpc.network\_transports.TcpServerTransport method), 15  
 Codec (class in quickrpc.codecs), 17  
 codec() (in module quickrpc), 6

## D

decode() (quickrpc.codecs.Codec method), 18  
 decode() (quickrpc.codecs.JsonRpcCodec method), 20  
 decode() (quickrpc terse\_codec.TerseCodec method), 21  
 DecodeError, 19

## E

echo() (quickrpc.echo\_api.EchoAPI method), 26  
 EchoAPI (class in quickrpc.echo\_api), 26  
 encode() (quickrpc.codecs.Codec method), 18  
 encode() (quickrpc.codecs.JsonRpcCodec method), 20  
 encode() (quickrpc terse\_codec.TerseCodec method), 21  
 encode\_error() (quickrpc.codecs.Codec method), 18  
 encode\_error() (quickrpc.codecs.JsonRpcCodec method), 20  
 encode\_error() (quickrpc terse\_codec.TerseCodec method), 21  
 encode\_reply() (quickrpc.codecs.Codec method), 18  
 encode\_reply() (quickrpc.codecs.JsonRpcCodec method), 20

encode\_reply() (quickrpc terse\_codec.TerseCodec method), 22  
 EncodeError, 19  
 ErrorReply (class in quickrpc.codecs), 19

## F

fromstring() (quickrpc.codecs.Codec class method), 19  
 fromstring() (quickrpc.codecs.JsonRpcCodec class method), 20  
 fromstring() (quickrpc.network\_transports.TcpClientTransport class method), 15  
 fromstring() (quickrpc.network\_transports.TcpServerTransport class method), 15  
 fromstring() (quickrpc.network\_transports.UdpTransport class method), 14  
 fromstring() (quickrpc.QtTransports.QProcessTransport class method), 16  
 fromstring() (quickrpc.QtTransports.QTcpTransport class method), 17  
 fromstring() (quickrpc.security.NoSecurity class method), 23  
 fromstring() (quickrpc.security.NullSecurity class method), 23  
 fromstring() (quickrpc.security.Security class method), 22  
 fromstring() (quickrpc terse\_codec.TerseCodec class method), 22  
 fromstring() (quickrpc.transports.MuxTransport class method), 11  
 fromstring() (quickrpc.transports.RestartingTransport class method), 12  
 fromstring() (quickrpc.transports.StdioTransport class method), 13

`fromstring()` (*quickrpc.transports.Transport class method*), 10

## H

`handle_received()` (*quickrpc.transports.MuxTransport method*), 11

## I

`incoming()` (*in module quickrpc*), 5

`incoming()` (*in module quickrpc.remote\_api*), 8

`InvalidSignatureError`, 22

`invert()` (*quickrpc.remote\_api.RemoteAPI method*), 7

`invert()` (*quickrpc.RemoteAPI method*), 4

## J

`JsonRpcCodec` (*class in quickrpc.codecs*), 19

## L

`L()` (*in module quickrpc.echo\_api*), 26

`L()` (*in module quickrpc terse\_codec*), 21

## M

`make_announcer()` (*in module quickrpc.announcer\_api*), 25

`make_udp_announcer()` (*in module quickrpc.announcer\_api*), 26

`Message` (*class in quickrpc.codecs*), 19

`message_error()` (*quickrpc.remote\_api.RemoteAPI method*), 8

`message_error()` (*quickrpc.RemoteAPI method*), 4

`MuxTransport` (*class in quickrpc.transports*), 11

## N

`NoSecurity` (*class in quickrpc.security*), 23

`NullSecurity` (*class in quickrpc.security*), 23

## O

`on_connect()` (*quickrpc.QtTransports.QTcpTransport method*), 17

`on_error()` (*quickrpc.QtTransports.QTcpTransport method*), 17

`on_finished()` (*quickrpc.QtTransports.QProcessTransport method*), 16

`on_ready_read()` (*quickrpc.QtTransports.QProcessTransport method*), 16

`on_ready_read()` (*quickrpc.QtTransports.QTcpTransport method*), 17

`open()` (*quickrpc.network\_transports.TcpClientTransport method*), 15

`open()` (*quickrpc.network\_transports.TcpServerTransport method*), 15

`open()` (*quickrpc.network\_transports.UdpTransport method*), 14

`open()` (*quickrpc.transports.MuxTransport method*), 11

`open()` (*quickrpc.transports.RestartingTransport method*), 12

`open()` (*quickrpc.transports.Transport method*), 10

`outgoing()` (*in module quickrpc*), 5

`outgoing()` (*in module quickrpc.remote\_api*), 8

## P

`paren_partition()` (*in module quickrpc.util*), 26

`Promise` (*class in quickrpc.promise*), 24

`PromiseDeadlockError`, 24

`PromiseDoneError`, 24

`PromiseError`, 24

`PromiseTimeoutError`, 24

`put()` (*quickrpc.action\_queue.ActionQueue method*), 25

## Q

`QProcessTransport` (*class in quickrpc.QtTransports*), 16

`QTcpTransport` (*class in quickrpc.QtTransports*), 16

`quickrpc` (*module*), 3

`quickrpc.action_queue` (*module*), 25

`quickrpc.announcer_api` (*module*), 25

`quickrpc.codecs` (*module*), 17

`quickrpc.echo_api` (*module*), 26

`quickrpc.network_transports` (*module*), 13

`quickrpc.promise` (*module*), 24

`quickrpc.QtTransports` (*module*), 16

`quickrpc.remote_api` (*module*), 6

`quickrpc.security` (*module*), 22

`quickrpc.terse_codec` (*module*), 21

`quickrpc.transports` (*module*), 9

`quickrpc.util` (*module*), 26

`quit()` (*quickrpc.echo\_api.EchoAPI method*), 26

## R

`received()` (*quickrpc.transports.Transport method*), 10

`receiver_thread` (*quickrpc.transports.RestartingTransport attribute*), 12

`receiver_thread` (*quickrpc.transports.Transport attribute*), 10

`RemoteAPI` (*class in quickrpc*), 3

`RemoteAPI` (*class in quickrpc.remote\_api*), 7

`RemoteError`, 6, 19

`remove_transport()` (*quickrpc.transports.MuxTransport method*), 11

`Reply` (*class in quickrpc.codecs*), 19

RestartingTcpClientTransport() (in module *quickrpc.transports*), 13

RestartingTransport (class in *quickrpc.transports*), 12

result() (*quickrpc.promise.Promise* method), 24

run() (*quickrpc.network\_transports.TcpClientTransport* method), 15

run() (*quickrpc.network\_transports.TcpServerTransport* method), 15

run() (*quickrpc.network\_transports.UdpTransport* method), 14

run() (*quickrpc.transports.MuxTransport* method), 11

run() (*quickrpc.transports.RestartingTransport* method), 12

run() (*quickrpc.transports.StdioTransport* method), 13

run() (*quickrpc.transports.Transport* method), 10

## S

say() (*quickrpc.echo\_api.EchoAPI* method), 26

sec\_in (*quickrpc.security.NullSecurity* attribute), 23

sec\_in() (*quickrpc.security.NoSecurity* method), 23

sec\_in() (*quickrpc.security.Security* method), 22

sec\_out (*quickrpc.security.NullSecurity* attribute), 23

sec\_out() (*quickrpc.security.NoSecurity* method), 23

sec\_out() (*quickrpc.security.Security* method), 22

Security (class in *quickrpc.security*), 22

SecurityError, 22

seek() (*quickrpc.announcer\_api.AnnouncerAPI* method), 25

send() (*quickrpc.network\_transports.TcpClientTransport* method), 15

send() (*quickrpc.network\_transports.UdpTransport* method), 14

send() (*quickrpc.QtTransports.QProcessTransport* method), 16

send() (*quickrpc.QtTransports.QTcpTransport* method), 17

send() (*quickrpc.transports.MuxTransport* method), 12

send() (*quickrpc.transports.RestartingTransport* method), 12

send() (*quickrpc.transports.StdioTransport* method), 13

send() (*quickrpc.transports.Transport* method), 10

set\_exception() (*quickrpc.promise.Promise* method), 24

set\_on\_received() (*quickrpc.transports.Transport* method), 10

set\_result() (*quickrpc.promise.Promise* method), 24

shorthand (*quickrpc.codecs.Codec* attribute), 19

shorthand (*quickrpc.codecs.JsonRpcCodec* attribute), 21

shorthand (*quickrpc.network\_transports.TcpClientTransport* attribute), 16

shorthand (*quickrpc.network\_transports.TcpServerTransport* attribute), 15

shorthand (*quickrpc.network\_transports.UdpTransport* attribute), 14

shorthand (*quickrpc.QtTransports.QProcessTransport* attribute), 16

shorthand (*quickrpc.QtTransports.QTcpTransport* attribute), 17

shorthand (*quickrpc.security.NoSecurity* attribute), 23

shorthand (*quickrpc.security.NullSecurity* attribute), 23

shorthand (*quickrpc.security.Security* attribute), 23

shorthand (*quickrpc.terse\_codec.TerseCodec* attribute), 22

shorthand (*quickrpc.transports.MuxTransport* attribute), 12

shorthand (*quickrpc.transports.RestartingTransport* attribute), 13

shorthand (*quickrpc.transports.StdioTransport* attribute), 13

shorthand (*quickrpc.transports.Transport* attribute), 10

start() (*quickrpc.QtTransports.QProcessTransport* method), 16

start() (*quickrpc.QtTransports.QTcpTransport* method), 17

start() (*quickrpc.transports.Transport* method), 10

StdioTransport (class in *quickrpc.transports*), 13

stop() (*quickrpc.QtTransports.QProcessTransport* method), 16

stop() (*quickrpc.QtTransports.QTcpTransport* method), 17

stop() (*quickrpc.transports.MuxTransport* method), 12

stop() (*quickrpc.transports.RestartingTransport* method), 13

stop() (*quickrpc.transports.StdioTransport* method), 13

stop() (*quickrpc.transports.Transport* method), 11

subclasses() (in module *quickrpc.util*), 26

subtransport\_running (*quickrpc.transports.RestartingTransport* attribute), 13

## T

TcpClientTransport (class in *quickrpc.network\_transports*), 15

TcpClientTransport() (in module *quickrpc.transports*), 13

TcpServerTransport (class in *quickrpc.network\_transports*), 14

TcpServerTransport() (in module *quickrpc.transports*), 13

TerseCodec (class in *quickrpc.terse\_codec*), 21

test() (in module *quickrpc.echo\_api*), 26

`then()` (*quickrpc.promise.Promise method*), [24](#)  
`Transport` (*class in quickrpc.transports*), [9](#)  
`transport` (*quickrpc.remote\_api.RemoteAPI attribute*), [8](#)  
`transport` (*quickrpc.RemoteAPI attribute*), [4](#)  
`transport()` (*in module quickrpc*), [5](#)

## U

`UdpTransport` (*class in quickrpc.network\_transports*), [13](#)  
`unhandled_calls()` (*quickrpc.remote\_api.RemoteAPI method*), [8](#)  
`unhandled_calls()` (*quickrpc.RemoteAPI method*), [4](#)  
`UnknownUserError`, [22](#)