IEP Documentation

Release 3.4

IEP contributors

September 29, 2015

Contents

API docs in progress.

IEP is organized in several subpackages, some of which can be used completely independent from the rest. Although IEP the IDE requires Qt and Python 3, some of its subpackages can safely be imported without these dependencies...

Contents:

CHAPTER 1

Codeeditor - The core editing component

This subpackage is independent of any other components of IEP and only has Qt (PySide or PyQt4) as a dependency. It works on Python version 2.7 and up (including 3.x).

Content and API will come here.

CHAPTER 2

Yoton - Inter process communication

Yoton is a Python package that provides a simple interface to communicate between two or more processes.

Yoton is independent of any other component of IEP and has *no dependencies* except Python itself. It runs on any Python version from 2.4.

Yoton is ...

- lightweight
- written in pure Python
- without dependencies (except Python)
- available on Python version >= 2.4, including Python 3
- cross-platform
- pretty fast

2.1 Overview

How it works:

- Multiple contexts can be connected over TCP/IP; the interconnected contexts together form a network.
- Messages are send between channel objects (channels are attached to a context).
- Channels are bound to a slot (a string name); a message send from a channel with slot X is received by all channels with slot X.
- Yoton may be used procedurally, or in an event-driven fashion.

Messaging patterns:

- Yoton supports the pub/sub pattern, in an N to M configuration.
- Yoton supports the req/rep pattern, allowing multiple requesters and repliers to exist in the same network.
- Yoton supports exchanging state information.

Some features:

- Yoton is optimized to handle large messages by reducing data copying.
- Yoton has a simple event system that makes asynchronous messaging and event-driven programming easy.

- Yoton also has functionality for basic client-server (telnet-like) communication.

2.1.1 A brief overview of the most common classes

- ***yoton.Context***
  - Represents a node in the network.
  - Has a bind() and connect() method to connect to other nodes.
- ***yoton.Connection***
  - Represents a connection to another context.
  - Wraps a single BSD-socket, using a persistent connection.
  - Has signals that the user can connect to to be notified of timeouts and closing of the connection.
- **Channel classes (i.e. *yoton.BaseChannel*)**
  - Channels are associated with a context, and send/receive at a particular slot (a string name).
  - Messages send at a particular slot can only be received by channels associated with the same slot.

2.1.2 Example

One end

```
import yoton

# Create one context and a pub channel
ct1 = yoton.Context(verbose=verbosity)
pub = yoton.PubChannel(ct1, 'chat')

# Connect
ct1.bind('publichost:test')

# Send
pub.send('hello world')
```

Other end

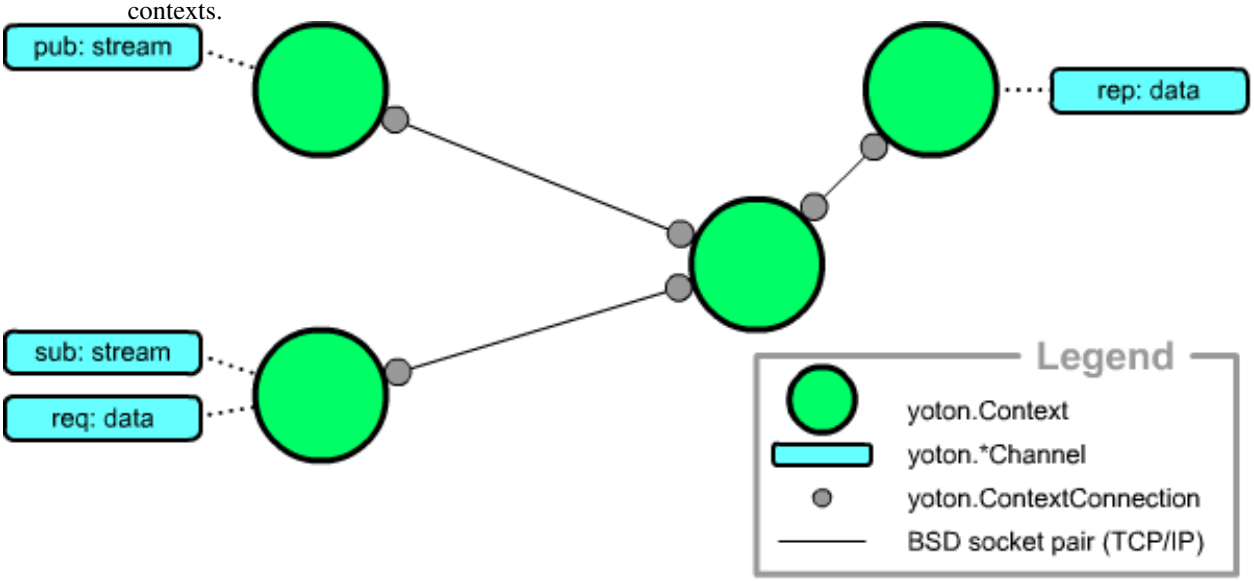
```
import yoton

# Create another context and a sub channel
ct2 = yoton.Context(verbose=verbosity)
sub = yoton.SubChannel(ct2, 'chat')

# Connect
ct2.connect('publichost:test')

# Receive
print(sub.recv())
```

2.2 Examples

2.2.1 Abstract example

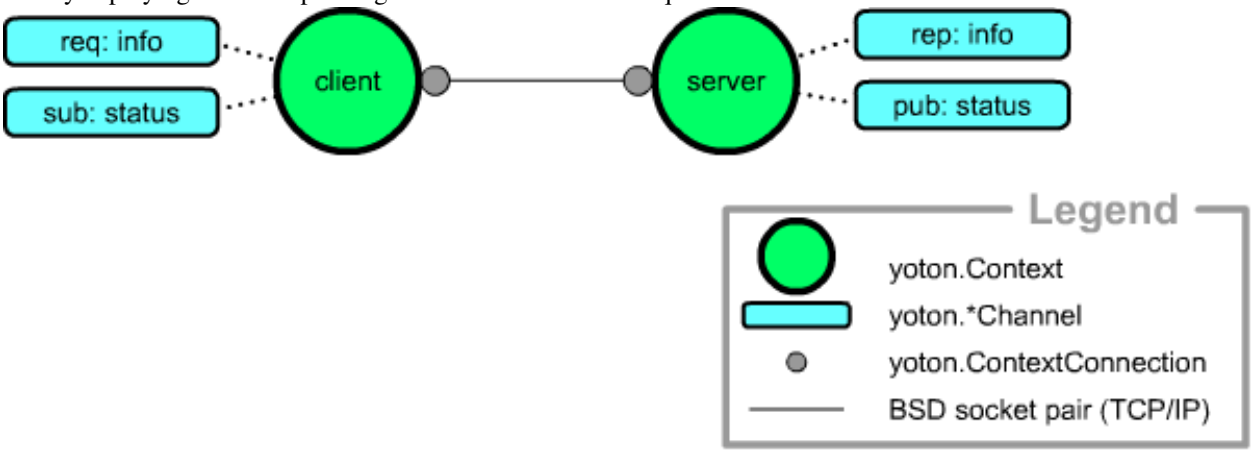
This example shows four connected contexts. In principal, all contexts are the same (a context is neither a client nor a server). Y

- The upper left context publishes a stream of messages.
- The upper right context employs a reply-channel, thereby taking on a server role.
- The bottom left context takes on a client-like role by subscribing to the "stream" channel and by having a request-channel on "data".
- The bottom right context has no channels and thus only serves as a means of connecting the different contexts.



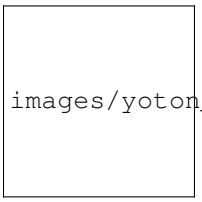
2.2.2 Simple client server

Two contexts. One takes a server role by having a publish-channel and a reply-channel. The other takes on a client role by deploying the corresponding subscribe-channel and request-channel.



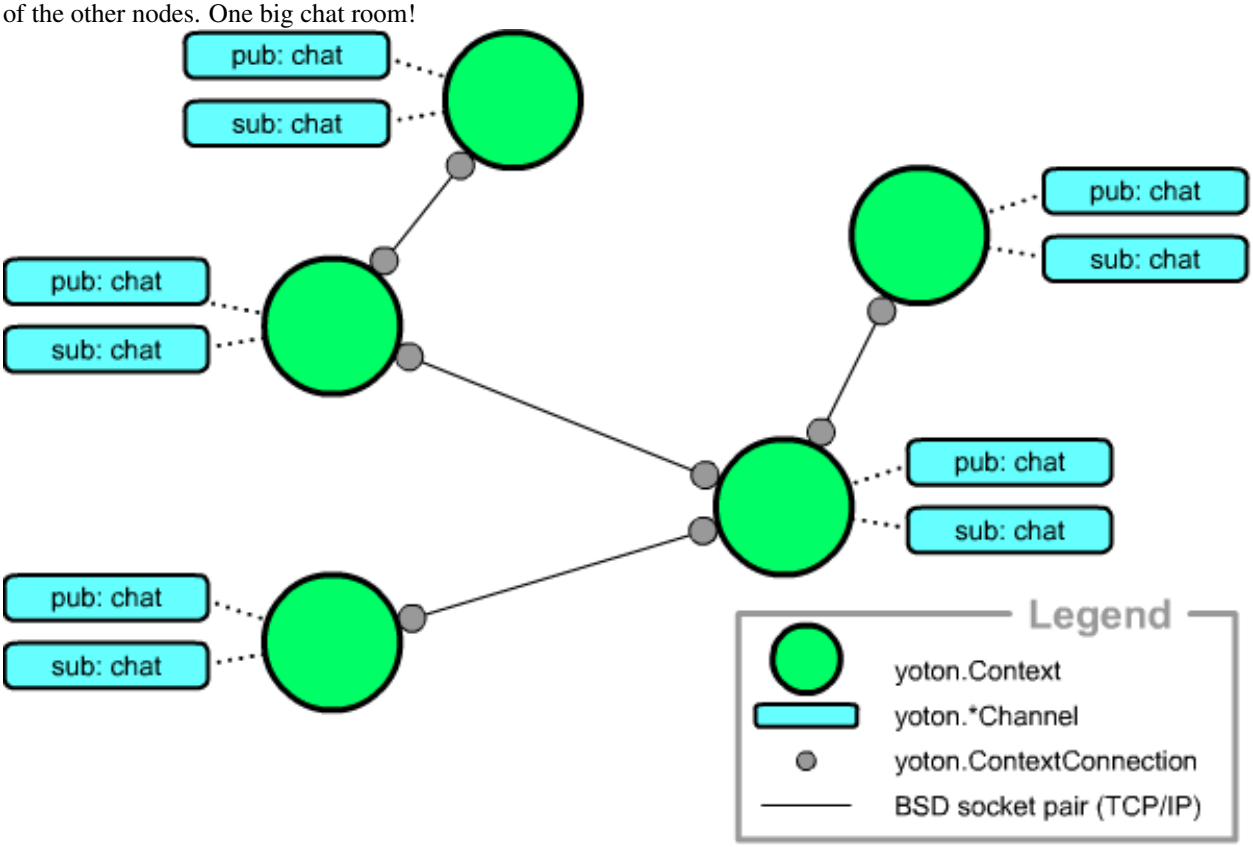
2.2.3 Multiple request/reply

This network contains only two types of context: requesters and repliers. Yoton performs a simple load balancing scheme: when the user posts a request, the req-channel first asks all repliers whether they can want to handle it. Eventually all repliers will answer that they do, but the actual request is only send to the first to respond. Requesters that are handling a previous request are unable to respond quickly so that the request will be handled by a "free" replier automatically. If all requesters are busy, the first to "come back" will handle the request.



2.2.4 Chat

This network consists of context which all take the same role; they all send chat messages and receive chat messages of the other nodes. One big chat room!

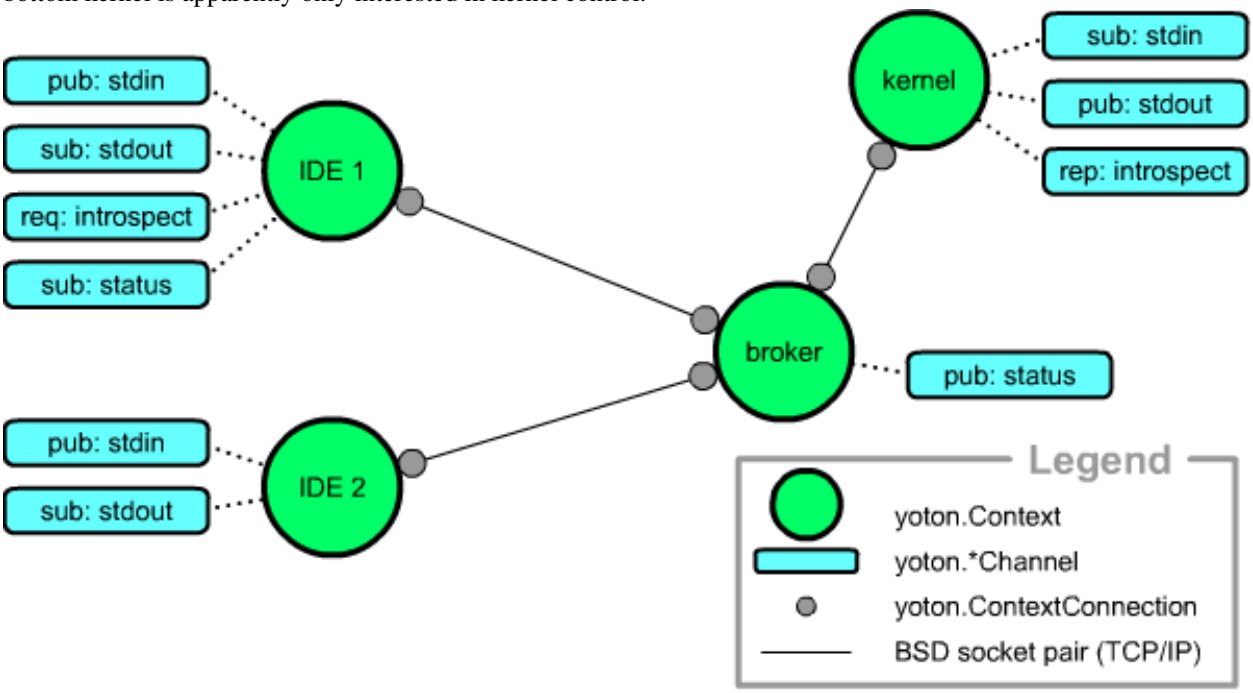


2.2.5 IDE / kernel

This network illustrates a simplified version of what yoton was initially designed for: client-kernel communication. IEP uses yoton for its kernel-client communications, see here which channels IEP uses for that.

The network consists of one kernel and two clients which are connected via a broker. Both clients can control the kernel via an stdin stream and receive output on stdout. The kernel also has a reply-channel so that the IDE's can

obtain introspection information (think auto-completion). The broker also publishes some status messages. The bottom kernel is apparently only interested in kernel control.



2.3 Context

class `yoton.`**`Context`**(*verbose=0*, *queue_params=None*)
Inherits from object

A context represents a node in the network. It can connect to multiple other contexts (using a *yoton.Connection*. These other contexts can be in another process on the same machine, or on another machine connected via a network or the internet.

This class represents a context that can be used by channel instances to communicate to other channels in the network. (Thus the name.)

The context is the entity that queue routes the packages produced by the channels to the other context in the network, where the packages are distributed to the right channels. A context queues packages while it is not connected to any other context.

If messages are send on a channel registered at this context while the context is not connected, the messages are stored by the context and will be send to the first connecting context.

Example 1

```
# Create context and bind to a port on localhost
context = yoton.Context()
context.bind('localhost:11111')
# Create a channel and send a message
pub = yoton.PubChannel(context, 'test')
pub.send('Hello world!')
```

Example 2

```
# Create context and connect to the port on localhost
context = yoton.Context()
context.connect('localhost:11111')
# Create a channel and receive a message
sub = yoton.SubChannel(context, 'test')
print(sub.recv() # Will print 'Hello world!'
```

Queue params

The queue_params parameter allows one to specify the package queues used in the system. It is recommended to use the same parameters for every context in the network. The value of queue_params should be a 2-element tuple specifying queue size and discard mode. The latter can be 'old' (default) or 'new', meaning that if the queue is full, either the oldest or newest messages are discarted.

PROPERTIES

connection_count
Get the number of connected contexts. Can be used as a boolean to check if the context is connected to any other context.

connections
Get a list of the Connection instances currently active for this context. In addition to normal list indexing, the connections objects can be queried from this list using their name.

connections_all
Get a list of all Connection instances currently associated with this context, including pending connections (connections waiting for another end to connect). In addition to normal list indexing, the connections objects can be queried from this list using their name.

id
The 8-byte UID of this context.

METHODS

bind(*address*, *max_tries=1*, *name=''*)
Setup a connection with another Context, by being the host. This method starts a thread that waits for incoming connections. Error messages are printed when an attemped connect fails. the thread keeps trying until a successful connection is made, or until the connection is closed.

Returns a Connection instance that represents the connection to the other context. These connection objects can also be obtained via the Context.connections property.

Parameters

address [str] Should be of the shape hostname:port. The port should be an integer number between 1024 and 2**16. If port does not represent a number, a valid port number is created using a hash function.

max_tries [int] The number of ports to try; starting from the given port, subsequent ports are tried until a free port is available. The final port can be obtained using the 'port' property of the returned Connection instance.

name [string] The name for the created Connection instance. It can be used as a key in the connections property.

Notes on hostname

The hostname can be:

- The IP address, or the string hostname of this computer.
- 'localhost': the connections is only visible from this computer. Also some low level networking layers are bypassed, which results in a faster connection. The other context should also connect to 'localhost'.

- 'publichost': the connection is visible by other computers on the same network. Optionally an integer index can be appended if the machine has multiple IP addresses (see socket.gethostbyname_ex).

close()
Close the context in a nice way, by closing all connections and all channels.

Closing a connection means disconnecting two contexts. Closing a channel means disasociating a channel from its context. Unlike connections and channels, a Context instance can be reused after closing (although this might not always the best strategy).

close_channels()
Close all channels associated with this context. This does not close the connections. See also close().

connect(*self*, *address*, *timeout=1.0*, *name=''*)
Setup a connection with another context, by connection to a hosting context. An error is raised when the connection could not be made.

Returns a Connection instance that represents the connection to the other context. These connection objects can also be obtained via the Context.connections property.

Parameters

address [str] Should be of the shape hostname:port. The port should be an integer number between 1024 and 2**16. If port does not represent a number, a valid port number is created using a hash function.

max_tries [int] The number of ports to try; starting from the given port, subsequent ports are tried until a free port is available. The final port can be obtained using the 'port' property of the returned Connection instance.

name [string] The name for the created Connection instance. It can be used as a key in the connections property.

Notes on hostname

The hostname can be:

- The IP address, or the string hostname of this computer.
- 'localhost': the connection is only visible from this computer. Also some low level networking layers are bypassed, which results in a faster connection. The other context should also host as 'localhost'.
- 'publichost': the connection is visible by other computers on the same network. Optionally an integer index can be appended if the machine has multiple IP addresses (see socket.gethostbyname_ex).

flush(*timeout=5.0*)
Wait until all pending messages are send. This will flush all messages posted from the calling thread. However, it is not guaranteed that no new messages are posted from another thread.

Raises an error when the flushing times out.

2.4 Connection

The connection classes represent the connection between two context. There is one base class (yoton.Connection) and currently there is one implementation: the *yoton.TcpConnection*. In the future other connections might be added that use other methods than TCP/IP.

class yoton.**Connection**(*context*, *name=''*)
Inherits from object

Abstract base class for a connection between two Context objects. This base class defines the full interface; subclasses only need to implement a few private methods.

The connection classes are intended as a simple interface for the user, for example to query port number, and be notified of timeouts and closing of the connection.

All connection instances are intended for one-time use. To make a new connection, instantiate a new Connection object. After instantiation, either _bind() or _connect() should be called.

PROPERTIES

closed
Signal emitted when the connection closes. The first argument is the ContextConnection instance, the second argument is the reason for the disconnection (as a string).

hostname1
Get the hostname corresponding to this end of the connection.

hostname2
Get the hostname for the other end of this connection. Is empty string if not connected.

id1
The id of the context on this side of the connection.

id2
The id of the context on the other side of the connection.

is_alive
Get whether this connection instance is alive (i.e. either waiting or connected, and not in the process of closing).

is_connected
Get whether this connection instance is connected.

is_waiting
Get whether this connection instance is waiting for a connection. This is the state after using bind() and before another context connects to it.

name
Set/get the name that this connection is known by. This name can be used to obtain the instance using the Context.connections property. The name can be used in networks in which each context has a particular role, to easier distinguish between the different connections. Other than that, the name has no function.

pid1
The pid of the context on this side of the connection. (hint: os.getpid())

pid2
The pid of the context on the other side of the connection.

port1
Get the port number corresponding to this end of the connection. When binding, use this port to connect the other context.

port2
Get the port number for the other end of the connection. Is zero when not connected.

timedout
This signal is emitted when no data has been received for over 'timeout' seconds. This can mean that the connection is unstable, or that the other end is running extension code.

Handlers are called with two arguments: the ContextConnection instance, and a boolean. The latter is True when the connection times out, and False when data is received again.

timeout
Set/get the amount of seconds that no data is received from the other side after which the timedout signal is emitted.

METHODS

close(*reason=None*)
Close the connection, disconnecting the two contexts and stopping all trafic. If the connection was waiting for a connection, it stops waiting.

Optionally, a reason for closing can be specified. A closed connection cannot be reused.

close_on_problem(*reason=None*)
Disconnect the connection, stopping all trafic. If it was waiting for a connection, we stop waiting.

Optionally, a reason for stopping can be specified. This is highly recommended in case the connection is closed due to a problem.

In contrast to the normal close() method, this method does not try to notify the other end of the closing.

flush(*timeout=3.0*)
Wait until all pending packages are send. An error is raised when the timeout passes while doing so.

class yoton.**TcpConnection**(*context*, *name=''*)
Inherits from Connection

The TcpConnection class implements a connection between two contexts that are in differenr processes or on different machines connected via the internet.

This class handles the low-level communication for the context. A ContextConnection instance wraps a single BSD socket for its communication, and uses TCP/IP as the underlying communication protocol. A persisten connection is used (the BSD sockets stay connected). This allows to better distinguish between connection problems and timeouts caused by the other side being busy.

2.5 Channels

The channel classes represent the mechanism for the user to send messages into the network and receive messages from it. A channel needs a context to function; the context represents a node in the network.

2.5.1 Slots

To be able to route messages to the right channel, channels are associated with a slot (a string name). This slot consists of a user-defined base name and an extension to tell the message type and messaging pattern. Messages send from a channel with slot X, are only received by channels with the same slot X. Slots are case insensitive.

2.5.2 Messaging patterns

Yoton supports three base messaging patterns. For each messaging pattern there are specific channel classes. All channels derive from *yoton.BaseChannel*.

publish/subscribe The *yoton.PubChannel* class is used for sending messages into the network, and the *yoton.SubChannel* class is used to receiving these messages. Multiple PubChannels and SubChannels can exist in the same network at the same slot; the SubChannels simply collect the messages send by all PubChannels.

request/reply The *yoton.ReqChannel* class is used to do requests, and the *yoton.RepChannel* class is used to reply to requests. If multiple ReqChannels are present at the same slot, simple load balancing is performed.

state The *yoton.StateChannel* class is used to communicate state to other state channels. Each *yoton.StateChannel* can set and get the state.

2.5.3 Message types

Messages are of a specific type (text, binary, ...), the default being Unicode text. The third (optional) argument to a Channel's initializer is a MessageType object that specifies how messages should be converted to bytes and the other way around.

This way, the channels classes themself can be agnostic about the message type, while the user can implement its own MessageType class to send whatever messages he/she likes.

class `yoton.`**`BaseChannel`**(*context*, *slot_base*, *message_type=yoton.TEXT*)
Inherits from object

Abstract class for all channels.

Parameters

context [*yoton.Context* instance] The context that this channel uses to send messages in a network.

slot_base [string] The base slot name. The channel appends an extension to indicate message type and messaging pattern to create the final slot name. The final slot is used to connect channels at different contexts in a network

message_type [yoton.MessageType instance] (default is yoton.TEXT) Object to convert messages to bytes and bytes to messages. Users can create their own message_type class to enable communicating any type of message they want.

Details

Messages send via a channel are delivered asynchronically to the corresponding channels.

All channels are associated with a context and can be used to send messages to other channels in the network. Each channel is also associated with a slot, which is a string that represents a kind of address. A message send by a channel at slot X can only be received by a channel with slot X.

Note that the channel appends an extension to the user-supplied slot name, that represents the message type and messaging pattern of the channel. In this way, it is prevented that for example a PubChannel can communicate with a RepChannel.

PROPERTIES

`closed`
Get whether the channel is closed.

`pending`
Get the number of pending incoming messages.

`received`
Signal that is emitted when new data is received. Multiple arrived messages may result in a single call to this method. There is no guarantee that recv() has not been called in the mean time. The signal is emitted with the channel instance as argument.

`slot_incoming`
Get the incoming slot name.

`slot_outgoing`
Get the outgoing slot name.

METHODS

close()
Close the channel, i.e. unregisters this channel at the context. A closed channel cannot be reused.

Future attempt to send() messages will result in an IOError being raised. Messages currently in the channel's queue can still be recv()'ed, but no new messages will be delivered at this channel.

class yoton.**PubChannel**(*context*, *slot_base*, *message_type=yoton.TEXT*)
Inherits from BaseChannel

The publish part of the publish/subscribe messaging pattern. Sent messages are received by all *yoton.SubChannel* instances with the same slot.

There are no limitations for this channel if events are not processed.

Parameters

context [*yoton.Context* instance] The context that this channel uses to send messages in a network.

slot_base [string] The base slot name. The channel appends an extension to indicate message type and messaging pattern to create the final slot name. The final slot is used to connect channels at different contexts in a network

message_type [yoton.MessageType instance] (default is yoton.TEXT) Object to convert messages to bytes and bytes to messages. Users can create their own message_type class to let channels any type of message they want.

METHODS

send(*message*)
Send a message over the channel. What is send as one message will also be received as one message.

The message is queued and delivered to all corresponding SubChannels (i.e. with the same slot) in the network.

class yoton.**SubChannel**(*context*, *slot_base*, *message_type=yoton.TEXT*)
Inherits from BaseChannel

The subscribe part of the publish/subscribe messaging pattern. Received messages were sent by a *yoton.PubChannel* instance at the same slot.

This channel can be used as an iterator, which yields all pending messages. The function *yoton.select_sub_channel* can be used to synchronize multiple SubChannel instances.

If no events being processed this channel works as normal, except that the received signal will not be emitted, and sync mode will not work.

Parameters

context [*yoton.Context* instance] The context that this channel uses to send messages in a network.

slot_base [string] The base slot name. The channel appends an extension to indicate message type and messaging pattern to create the final slot name. The final slot is used to connect channels at different contexts in a network

message_type [yoton.MessageType instance] (default is yoton.TEXT) Object to convert messages to bytes and bytes to messages. Users can create their own message_type class to let channels any type of message they want.

METHODS

next()
Return the next message, or raises StopIteration if non available.

recv(*block=True*)
Receive a message from the channel. What was send as one message is also received as one message.

If block is False, returns empty message if no data is available. If block is True, waits forever until data is available. If block is an int or float, waits that many seconds. If the channel is closed, returns empty message.

recv_all()
Receive a list of all pending messages. The list can be empty.

recv_selected()
Receive a list of messages. Use only after calling *yoton.select_sub_channel* with this channel as one of the arguments.

The returned messages are all received before the first pending message in the other SUB-channels given to select_sub_channel.

The combination of this method and the function select_sub_channel enables users to combine multiple SUB-channels in a way that preserves the original order of the messages.

set_sync_mode(*value*)
Set or unset the SubChannel in sync mode. When in sync mode, all channels that send messages to this channel are blocked if the queue for this SubChannel reaches a certain size.

This feature can be used to limit the rate of senders if the consumer (i.e. the one that calls recv()) cannot keep up with processing the data.

This feature requires the yoton event loop to run at the side of the SubChannel (not necessary for the *yoton.PubChannel* side).

yoton.**select_sub_channel**(*channel1*, *channel2*, ...)
Returns the channel that has the oldest pending message of all given yoton.SubCannel instances. Returns None if there are no pending messages.

This function can be used to read from SubCannels instances in the order that the messages were send.

After calling this function, use channel.recv_selected() to obtain all messages that are older than any pending messages in the other given channels.

class yoton.**ReqChannel**(*context*, *slot_base*)
Inherits from BaseChannel

The request part of the request/reply messaging pattern. A ReqChannel instance sends request and receive the corresponding replies. The requests are replied by a *yoton.RepChannel* instance.

This class adopts req/rep in a remote procedure call (RPC) scheme. The handling of the result is done using a *yoton.Future* object, which follows the approach specified in PEP 3148. Note that for the use of callbacks, the yoton event loop must run.

Basic load balancing is performed by first asking all potential repliers whether they can handle a request. The actual request is then send to the first replier to respond.

Parameters

context [*yoton.Context* instance] The context that this channel uses to send messages in a network.

slot_base [string] The base slot name. The channel appends an extension to indicate message type and messaging pattern to create the final slot name. The final slot is used to connect channels at different contexts in a network

Usage

One performs a call on a virtual method of this object. The actual method is executed by the *yoton.RepChannel* instance. The method can be called with normal and keyword arguments, which can be (a combination of): None, bool, int, float, string, list, tuple, dict.

Example

```
# Fast, but process is idling when waiting for the response.
reply = req.add(3,4).result(2.0) # Wait two seconds

# Asynchronous processing, but no waiting.
def reply_handler(future):
    ... # Handle reply
future = req.add(3,4)
future.add_done_callback(reply_handler)
```

class yoton.**RepChannel**(*context*, *slot_base*)
Inherits from BaseChannel

The reply part of the request/reply messaging pattern. A RepChannel instance receives request and sends the corresponding replies. The requests are send from a *yoton.ReqChannel* instance.

This class adopts req/rep in a remote procedure call (RPC) scheme.

To use a RepChannel, subclass this class and implement the methods that need to be available. The reply should be (a combination of) None, bool, int, float, string, list, tuple, dict.

This channel needs to be set to event or thread mode to function (in the first case yoton events need to be processed too). To stop handling events again, use set_mode('off').

Parameters

context [*yoton.Context* instance] The context that this channel uses to send messages in a network.

slot_base [string] The base slot name. The channel appends an extension to indicate message type and messaging pattern to create the final slot name. The final slot is used to connect channels at different contexts in a network

METHODS

echo(*arg1*, *sleep=0.0*)
Default procedure that can be used for testing. It returns a tuple (first_arg, context_id)

set_mode(*mode*)
Set the replier to its operating mode, or turn it off.

Modes:

- 0 or 'off': do not process requests
- 1 or 'event': use the yoton event loop to process requests
- 2 or 'thread': process requests in a separate thread

class yoton.**Future**(*req_channel*, *req*, *request_id*)
Inherits from object

The Future object represents the future result of a request done at a *yoton.ReqChannel*.

It enables:

- checking whether the request is done.
- getting the result or the exception raised during handling the request.
- canceling the request (if it is not yet running)

- registering callbacks to handle the result when it is available

METHODS

add_done_callback(*fn*)
Attaches the callable fn to the future. fn will be called, with the future as its only argument, when the future is cancelled or finishes running.

Added callables are called in the order that they were added. If the callable raises a Exception subclass, it will be logged and ignored. If the callable raises a BaseException subclass, the behavior is undefined.

If the future has already completed or been cancelled, fn will be called immediately.

cancel()
Attempt to cancel the call. If the call is currently being executed and cannot be cancelled then the method will return False, otherwise the call will be cancelled and the method will return True.

cancelled()
Return True if the call was successfully cancelled.

done()
Return True if the call was successfully cancelled or finished running.

exception(*timeout*)
Return the exception raised by the call. If the call hasn't yet completed then this method will wait up to timeout seconds. If the call hasn't completed in timeout seconds, then a TimeoutError will be raised. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.

If the future is cancelled before completing then CancelledError will be raised.

If the call completed without raising, None is returned.

result(*timeout=None*)
Return the value returned by the call. If the call hasn't yet completed then this method will wait up to timeout seconds. If the call hasn't completed in timeout seconds, then a TimeoutError will be raised. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.

If the future is cancelled before completing then CancelledError will be raised.

If the call raised, this method will raise the same exception.

result_or_cancel(*timeout=1.0*)
Return the value returned by the call. If the call hasn't yet completed then this method will wait up to timeout seconds. If the call hasn't completed in timeout seconds, then the call is cancelled and the method will return None.

running()
Return True if the call is currently being executed and cannot be cancelled.

set_auto_cancel_timeout(*timeout*)
Set the timeout after which the call is automatically cancelled if it is not done yet. By default, this value is 10 seconds.

If timeout is None, there is no limit to the wait time.

set_exception(*exception*)
Sets the result of the work associated with the Future to the Exception exception. This method should only be used by Executor implementations and unit tests.

set_result(*result*)
Sets the result of the work associated with the Future to result. This method should only be used by Executor implementations and unit tests.

set_running_or_notify_cancel()
This method should only be called by Executor implementations before executing the work associated with the Future and by unit tests.

If the method returns False then the Future was cancelled, i.e. Future.cancel() was called and returned True.

If the method returns True then the Future was not cancelled and has been put in the running state, i.e. calls to Future.running() will return True.

This method can only be called once and cannot be called after Future.set_result() or Future.set_exception() have been called.

class yoton.**StateChannel**(*context*, *slot_base*, *message_type=yoton.TEXT*)
Inherits from BaseChannel

Channel class for the state messaging pattern. A state is synchronized over all state channels of the same slot. Each channel can send (i.e. set) the state and recv (i.e. get) the current state. Note however, that if two StateChannel instances set the state around the same time, due to the network delay, it is undefined which one sets the state the last.

The context will automatically call this channel's send_last() method when a new context enters the network.

The recv() call is always non-blocking and always returns the last received message: i.e. the current state.

There are no limitations for this channel if events are not processed, except that the received signal is not emitted.

Parameters

context [*yoton.Context* instance] The context that this channel uses to send messages in a network.

slot_base [string] The base slot name. The channel appends an extension to indicate message type and messaging pattern to create the final slot name. The final slot is used to connect channels at different contexts in a network

message_type [yoton.MessageType instance] (default is yoton.TEXT) Object to convert messages to bytes and bytes to messages. Users can create their own message_type class to let channels any type of message they want.

METHODS

recv(*block=False*)
Get the state of the channel. Always non-blocking. Returns the most up to date state.

send(*message*)
Set the state of this channel.

The state-message is queued and send over the socket by the IO-thread. Zero-length messages are ignored.

send_last()
Resend the last message.

2.6 Event system

Module *yoton.events*

Yoton comes with a simple event system to enable event-driven applications.

All channels are capable of running without the event system, but some channels have limitations. See the documentation of the channels for more information. Note that signals only work if events are processed.

class yoton.**Signal**
Inherits from object

The purpose of a signal is to provide an interface to bind/unbind to events and to fire them.

One can bind() or unbind() a callable to the signal. When emitted, an event is created for each bound handler. Therefore, the event loop must run for signals to work.

Some signals call the handlers using additional arguments to specify specific information.

PROPERTIES

type
The type (__class__) of this event.

METHODS

bind(*func*)
Add an eventhandler to this event.

The callback/handler (func) must be a callable. It is called with one argument: the event instance, which can contain additional information about the event.

emit(**args*, ***kwargs*)
Emit the signal, calling all bound callbacks with *args and **kwargs. An event is queues for each callback registered to this signal. Therefore it is safe to call this method from another thread.

emit_now(**args*, ***kwargs*)
Emit the signal *now*. All handlers are called from the calling thread. Beware, this should only be done from the same thread that runs the event loop.

unbind(*func=None*)
Unsubscribe a handler, If func is None, remove all handlers.

class yoton.**Timer**(*interval=1.0*, *oneshot=True*)
Inherits from Signal

Timer class. You can bind callbacks to the timer. The timer is fired when it runs out of time.

Parameters

interval [number] The interval of the timer in seconds.

oneshot [bool] Whether the timer should do a single shot, or run continuously.

PROPERTIES

interval
Set/get the timer's interval in seconds.

oneshot
Set/get whether this is a oneshot timer. If not is runs continuously.

running
Get whether the timer is running.

METHODS

start(*interval=None*, *oneshot=None*)
Start the timer. If interval or oneshot are not given, their current values are used.

stop()
Stop the timer from running.

yoton.**call_later**(*func*, *timeout=0.0*, **args*, ***kwargs*)
Call the given function after the specified timeout.

Parameters

func [callable] The function to call.

timeout [number] The time to wait in seconds. If zero, the event is put on the event queue. If negative, the event will be put at the front of the event queue, so that it's processed asap.

args [arguments] The arguments to call func with.

kwargs: keyword arguments. The keyword arguments to call func with.

yoton.**process_events**(*block=False*)
Process all yoton events currently in the queue. This function should be called periodically in order to keep the yoton event system running.

block can be False (no blocking), True (block), or a float blocking for maximally 'block' seconds.

yoton.**start_event_loop**()
Enter an event loop that keeps calling yoton.process_events(). The event loop can be stopped using stop_event_loop().

yoton.**stop_event_loop**()
Stops the event loop if it is running.

yoton.**embed_event_loop**(*callback*)
Embed the yoton event loop in another event loop. The given callback is called whenever a new yoton event is created. The callback should create an event in the other event-loop, which should lead to a call to the process_events() method. The given callback should be thread safe.

Use None as an argument to disable the embedding.

2.7 clientserver – Request-reply pattern using a client-server model

yoton.clientserver.py

Yoton comes with a small framework to setup a request-reply pattern using a client-server model (over a non-persistent connection), similar to telnet. This allows one process to easily ask small pieces of information from another process.

To create a server, create a class that inherits from *yoton.RequestServer* and implement its handle_request() method.

A client process can simply use the *yoton.do_request* function. Example: `yoton.do_request('www.google.com:80', 'GET http/1.1\r\n')`

The client server model is implemented using one function and one class: *yoton.do_request* and *yoton.RequestServer*.

Details

The server implements a request/reply pattern by listening at a socket. Similar to telnet, each request is handled using a connection and the socket is closed after the response is send.

The request server can setup to run in the main thread, or can be started using its own thread. In the latter case, one can easily create multiple servers in a single process, that listen on different ports.

2.7.1 Implementation

The client server model is implemented using one function and one class: *yoton.do_request* and *yoton.RequestServer*.

yoton.**do_request**(*address*, *request*, *timeout=-1*)

Do a request at the server at the specified address. The server can be a *yoton.RequestServer*, or any other server listening on a socket and following a REQ/REP pattern, such as html or telnet. For example: `html = do_request('www.google.com:80', 'GET http/1.1\r\n')`

Parameters

address [str] Should be of the shape hostname:port.

request [string] The request to make.

timeout [float] If larger than 0, will wait that many seconds for the respons, and return None if timed out.

Notes on hostname

The hostname can be:

- The IP address, or the string hostname of this computer.
- 'localhost': the connections is only visible from this computer. Also some low level networking layers are bypassed, which results in a faster connection. The other context should also connect to 'localhost'.
- 'publichost': the connection is visible by other computers on the same network.

class yoton.**RequestServer**(*address*, *async=False*, *verbose=0*)

Inherits from Thread

Setup a simple server that handles requests similar to a telnet server, or asyncore. Starting the server using run() will run the server in the calling thread. Starting the server using start() will run the server in a separate thread.

To create a server, subclass this class and re-implement the handle_request method. It accepts a request and should return a reply. This server assumes utf-8 encoded messages.

Parameters

address [str] Should be of the shape hostname:port.

async [bool] If True, handles each incoming connection in a separate thread. This might be advantageous if a the handle_request() method takes a long time to execute.

verbose [bool] If True, print a message each time a connection is accepted.

Notes on hostname

The hostname can be:

- The IP address, or the string hostname of this computer.
- 'localhost': the connections is only visible from this computer. Also some low level networking layers are bypassed, which results in a faster connection. The other context should also connect to 'localhost'.
- 'publichost': the connection is visible by other computers on the same network.

2.8 Internals (if you want to know more)

In yoton, the *yoton.Context* is the object that represents the a node in the network. The context only handles packages. It gets packages from all its associated channels and from the other nodes in the network. It routes packages to the other nodes, and deposits packages in channel instances if the package slot matches the channel slot.

The *yoton.Connection* represents a one-to-one connection between two contexts. It handles the low level messaging. It breaks packages into pieces and tries to send them as efficiently as possible. It also receives bytes from the other end, and reconstructs it into packages, which are then given to the context to handle (i.e. route). For the *yoton.TcpConnection* this is all done by dedicated io threads.

2.8.1 Packages

Packages are simply a bunch of bytes (the encoded message), wrapped in a header. Packages are directed at a certain slot. They also have a source id, source sequence number, and optionally a destination id and destination sequence number (so that packages can be replies to other packages). When a package is received, it also gets assigned a receiving sequence number (in order to synchronize channels).

2.8.2 Levels of communication

Two *yoton.Connection* instances also communicate directly with each-other. They do this during the handshaking procedure, obviously, but also during operation they send each-other heart beat messages to detect time-outs. When the connection is closed in a nice way, thet also send a close message to the other end. A package addressed directly at the Connection has no body (consists only of a header).

Two contexts can also communicate. They do this to notify each-other of new formed connections, closing of contexts, etc. A package directed at a context uses a special slot.

Channel instances can also communicate. Well, that's what yoton is all about... A sending channels packs a message in a package and gives it to the contect. All other contexts will receive the package and deposit it in the channel's queue if the slots match. On recv'ing, the message is extracted/decoded from the package.

2.8.3 Persistent connection

Two *yoton.TcpConnection* instances are connected using a single BSD-socket (TCP/IP). The socket operates in persistent mode; once the connection is established, the socket remains open until the connection is closed indefinetely.

Would we adopt a req/rep approach (setting up the connection for each request), failure could mean either that the kernel is running extension code, or that the connection is broken. It's not possible to differentiate between the two.

On initialization of the connection, TcpConnection's perform a small handshake procedue to establish that both are a *yoton.TcpConnection* objects, and to exchange the context id's.

There is one thread dedicated to receive data from the socket, and subsequently have the context route the packages. Another dedicated thread gets data from a queue (of the Connection) and sends the packages over the sockets. The sockets and queues are blocking, but on a timeout (the receiving thread uses a select() call for this). This makes it easy to periodically send heartbeat packages if necessary, and their absence can be detected.

In a previous design, there was a single io thread per context that did all the work. It would run through a generator function owned by the connections to send/receive data. This required all queueing and io to be non-blocking. After changing the design the code got *much* smaller, cleaner and easier to read, and is probably more robust. We could also get rid of several classes to buffer data, because with blocking threads the data can sinply be buffered at the queues and sockets.

2.8.4 Message framing

To differentiate between messages, there are two common approaches. One can add a small header to each message that indicates how long the message is. Or one can delimit the messages with a specific character.

In earlier designs, yoton used the second approach and was limited to sending text which was encoded using utf-8. This meant the bytes 0xff and 0xfe could be used for delimiting.

The first approach is more complex and requires more per-message processing. However, because the message size is know, messages can be received with much less copying of data. This significantly improved the performance for larger messages (with the delimiting approach we would get memory errors when Yoton tried to encode/decode the message to/from utf-8).

The current design is such that as little data has to be copied (particularly for larger messages).

2.8.5 Heart beat signals

If there is no data to send for a while, small heart beat messages are produced, so that connection problems can be easily detected. For TCP one needs to send data in order to detect connection problem (because no ACK's will be received). However, the TCP timeout is in the order of minutes and is different between OS's. Therefore we check when the last time was that data was received, enabling us to detect connection problems in the order of a few seconds.

Note that when two Context's are connected using 'localhost', there is no way for the connection to be lost, as several network layers are bypassed. In such a situation, we can therefore be sure that the reason for the timeout lies not in the connection, but is caused for example by the process running extension code.

2.8.6 When the process runs extension code

With respect to client-kernel comminication: the kernel will not be able to send any data (neither heart beat signals) if its running extension code. In such a case, the client can still send messages; this data is transported by TCP and ends up in the network buffer until the kernel returns from extension code and starts receiving messages again.

For this reason, in a client-kernel configuration, the kernel should always be connected to another process via 'localhost', and should use a proxi/broker to connect with clients on another box.

In that case, the client can detect that the kernel is running extension code because the kernel stopped sending data (incl heartbeat messages).

2.8.7 Congestion prevention

In any communication system, there is a risk of congestion: one end sends data faster than the other end can process it. This data can be buffered, but as the buffer fills, it consumes more memory.

Yoton uses two approaches to solve this problem. The first (and most common) solution is that all queues have a maximum size. When this size is reached and a new messages is added, messages will be discarted. The user can choose whether the oldest or the newest message should be discarted.

The second approach is only possible for the PUB/SUB channels. If the *yoton.SubChannel* is put in sync-mode (using the set_sync_mode method), the *yoton.SubChannel* will send a message to the corresponding PubChannels if its queue reaches a certain size. This size is relatively small (e.g. 10-100). When a *yoton.PubChannel* receives the message, its send method will block (for at most 1 second). The SubChannel sends a second message when the queue is below a certain level again. Note that it takes a while for these control messages to be received by the PubChannel. Therefore the actual queue size can easily grow larger than the threshold. In this situation, the first approach (discarting messages is still used as a failsave, but messages are very unlikely to be discarted since the threshold is much much smaller than the maximum queue size.

An important aspect for the second approach is that the queue that buffers packages before they are send over the socket remains small. If this is not the case, the PubChannel is able to spam the queue with gigantic amounts of messages before the SubChannel even receives the first message. To keep this queue small, much like the queue of the SubChannel, it has a certain threshold. If this threshold is reached, subsequent pushes on the queue will block for maximally 1 second. The threshold is in the same order of magnitude as the queue for the SubChannel.

2.8.8 References

- http://www.unixguide.net/network/socketfaq/2.9.shtml
- http://nitoprograms.blogspot.com/2009/04/message-framing.html
- http://nitoprograms.blogspot.com/2009/05/detection-of-halfopen-dropped.html

2.9 Experiments with sockets

I performed a series on tests on bot Windows and Linux. Testing sockets on localhost and publichost (what you get with gethostname()), for killing one of the processes, and removing the connection (unplugging the cable).

I wanted to answer the following questions:

- When and how can we detect that the other process dropped (killed or terminated)?
- When and how can we detect connection problems?
- Can we still send data if the other end stops receiving data? And if not, can we still detect a connection drop?

2.9.1 On Windows, same box

- If hosting on network level, and killing the network device, the disconnection is immediately detected (socket.error is raised when calling send() or recv()).
- Killing either process will immediately result in an error being raised.
- This is true for hosting local or public.
- -> Therefore, when hosting on localhost (the connection cannot be lost) we do not need a heartbeat.

2.9.2 On Linux, same box

- I cannot kill the connection, only the process. When I do, the other side will be able to receive, but receives EOF (empty bytes), which looks like a nice exit. Note that this behavior is different than on Windows.
- When the other side does not receive, I can still send huge amounts of data.
- This applies when hosting as localhost or as gethostname.

2.9.3 On Linux & Windows, different boxes

- When I kill the connection (remove network cable), it takes a while for either end to see that the connection is dead. I can even put the cable back in and go on communicating. This is a feature of TCP to be robust against network problems. See http://www.unixguide.net/network/socketfaq/2.8.shtml

- When I keep the connection, but kill the process on either end, this is detected immediately (in the same way as described above, depending on the OS); there's still low-level communication between the two boxes, and the port is detected to be unavailable.
- On Linux I can keep sending huge amounts of data even if the other end does not receive. On Windows I can't. In both cases they can detect the other process dropping.

2.9.4 Conclusions

- On local machine (broker-kernel), we use localhost so that we will not have network problems. We can detect if a process drops, which is essential.
- Between boxes, we use a heartbeat signal to be able to determine whether the connection is still there. If we set that timeout low enough (<30 sec or so) we can even distinguish a network problem from a process crash. This should not be necessary, however, because we can assume (I guess) that the broker and client close nicely.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

y

T

U

Y