# "pydvbcss - a library implementing DVB protocols for Companion Screen Synchronisation

***Release 0.5.2-latest***

**British Broadcasting Corporation**

**Feb 15, 2018**

# Contents

**DVB protocols for synchronisation between TV Devices and Companion Screen Applications.**

**Release** 0.5.2-latest

**Licence** Apache License v2.0.

**Source** https://github.com/BBC/pydvbcss/tree/master/

**How to install** https://github.com/BBC/pydvbcss/tree/master/README.md

**Changelog** https://github.com/BBC/pydvbcss/tree/master/CHANGELOG.md

CHAPTER 1

# Run the examples

The code in the *examples* directory demonstrates how to create and control servers and clients for all three protocols: CSS-CII, CSS-TS and CSS-WC.

- ***WallClockServer.py*** *and* ***WallClockClient.py***
- ***CIIServer.py*** *and* ***CIIClient.py***
- ***TSServer.py*** *and* ***TSClient.py***
- ***TVDevice.py***

There are instructions below on how to run the examples and see them interact with each other.

See the sources here: on github

## 1.1 WallClockServer.py and WallClockClient.py

### 1.1.1 Get started

The *WallClockServer* and *WallClockClient* examples use the library to implement a simple server and client for the *CSS-WC protocol*.

First start the server, specifying the host and IP to listen on:

```
$ python examples/WallClockServer.py 127.0.0.1 6677
```

Leave it running in the background and start a client, telling it where to connect to the server:

```
$ python examples/WallClockClient.py 127.0.0.1 6677
```

**Note:** The wall clock protocol is connectionless (it uses UDP) This means the client will not report an error if you enter the wrong IP address or port number.

Watch the "dispersion" values which indicate how much margin for error there is in the client's wall clock estimate. If the value is very large, this means it is not receiving responses from the server.

### 1.1.2 How they work

#### WallClockServer.py [source]

It works by instantiating a *WallClockServer* object and providing that object with a *clock* object to be used as the Wall Clock that is to be served.

At the command line you can override default options for the ip address and port the server binds to; the maximum frequency error it reports and whether it sends "follow-up" responses to requests.

Use the `--help` command line option for usage information.

#### WallClockClient.py [source]

It works by instantiating a *WallClockClient* object and plugs into that object a *LowestDispersionCandidate* algorithm object that adjusts a *TunableClock* representing the Wall Clock.

At the command line you must specify the host and port of the Wall Clock server. Default options can be overridden for the IP address and port that the client listens on.

Use the `--help` command line option for usage information.

## 1.2 CIIServer.py and CIIClient.py

### 1.2.1 Get started

The *CIIServer* and *CIIClient* examples implement the *CSS-CII* protocol, with the server sharing some pretend CII status information with the client.

First start the server:

```
$ python examples/CIIServer.py
```

The server listens on 127.0.0.1 on port 7681 and accepts WebSocket connections to *ws://<ip>:<port>/cii*.

Leave it running in the background and connect using the client and see how the CII data is pushed by the server whenever it changes:

```
$ python examples/CIIClient.py ws:/127.0.0.1:7681/cii
```

### 1.2.2 How they work

#### CIIServer.py [source]

It works by setting up a web server and the ws4py plug-in for cherrypy that provides WebSockets support. It then instantiates a *CIIServer* and mounts it into the cherrypy server at the URL resource path "/cii".

While the server is running, it pretends to be hopping between a few different broadcast channels every 7 seconds, with a 2 second "transitioning" period on each hop.

This is an artificially simple example and does not provide values for most properties of the CII message - such as a MRS URL, or any URLs for a WC or TS endpoints.

It does not do any media presentation, but just provides a CSS-CII server with some pretend data.

This server, by default, serves on port 7681 and provides a CSS-CII service at the URL resource path `/cii`. It can therefore be connected to using the WebSocket URL "ws://<host>:7681/cii" e.g. "ws://127.0.0.1:7681/cii". Command line options can be used to override these defaults and to reduce the amount of logging output.

Use the `--help` command line option for more detailed usage information.

#### CIIServer.py [source]

It works by instantiating a *CIIClient* and attaching handler functions to be notified of when connection and disconnection occurs and of changes to the CII information being pushed from the server.

At the command line you must specify:

- the WebSocket URL of the CSS-CII server, in the form *ws://<host>:<port>/<path>*

Command line options can be used to reduce the amount of logging output.

Use the `--help` command line option for usage information.

## 1.3 TSServer.py and TSClient.py

### 1.3.1 Get started

The *TServer* and *TSClient* examples implement the *CSS-TS* protocol, with the server pretending to have a few different timelines for a DVB broadcast service (where the content ID is a DVB URL).

First start the server:

```
$ python examples/TSServer.py
```

The server listens on 127.0.0.1 on port 7681 and accepts WebSocket connections to *ws://<ip>:<port>/ts*. It also includes a wall clock server, also on 127.0.0.1 on port 6677.

Leave it running in the background and connect using the client and see how the client is able to synchronise and periodically print an estimate of the timeline position (converted to units of seconds):

```
$ python examples/TSClient.py ws://127.0.0.1:7681/ts udp://127.0.0.1:6677 "dvb://"
↪"urn:dvb:css:timeline:pts" 90000
```

Here we have told it to request a timeline for whatever content the server thinks it is showing provided that the content ID begins with "dvb://". Assuming that matches, then the timeline is to be a PTS timeline, which ticks at 90kHz (the standard rate of PTS in an MPEG transport stream).

### 1.3.2 How they work

#### TSServer.py [source]

It works by setting up a web server and the ws4py plug-in for cherrypy that provides WebSockets support. It then instantiates a `TSServer` and mounts it into the cherrypy server at the URL resource path "/ts". It also includes a wall clock server.

It does not play any media, but instead serves an imaginary set of timelines.

It creates `clock` objects to represent timelines and the wall clock. `SimpleClockTimelineSource` objects are used to interface the clocks as sources of timelines to the TS server object.

It has a hardcoded DVB URL as the content ID (displayed when you start it running) and provides the following timelines:

- "*urn:dvb:css:timeline:pts*" ... a PTS timeline
- "*urn:dvb:css:timelime:temi:1:1*" ... a TEMI timeline ticking at 1kHz
- "*urn:dvb:css:timelime:temi:1:5*" ... a TEMI timeline ticking at 1kHz, that toggles availability every 10 seconds
- "*urn:dvb:css:timelime:temi:1:2*" ... the same, but it takes 10 seconds for the server to begin providing this timeline after a client first requests it
- "*urn:pydvbcss:sporadic*" ... a meaningless timeline whose availability toggles every 10 seconds.

The PTS and TEMI timelines both pause periodically and have their timing tweaked by a fraction of a second. The "sporadic" timeline shows how the protocol supports having timelines appear (become available) and disappear (become unavailable) while a client is connected.

By default, this server serves at 127.0.0.1 on port 7681 and provides a CSS-TS service at the URL *ws://127.0.0.1:7681/ts*. It also provides a wall clock server bound to 0.0.0.0 on UDP port 6677. Command line options can be used to override these defaults and to reduce the amount of logging output.

Use the `--help` command line option for more detailed usage information.

#### TSClient.py [source]

It works by implementing *both* a wall clock client and a CSS-TS client. A `TSClientClockController` object is instantiated and provided with a `CorrelatedClock` object to represent the synchronisation timeline. The controller adjusts the clock object to match the timeline information coming from the server.

At the command line you must specify:

- the WebSocket URL of the CSS-TS server, in the form *ws://<host>:<port>/<path>*
- a *udp://<host>:<port>* format URL for the Wall Clock server
- The content ID stem and timeline selector to be used when requesting the timeline
- The tick rate of the timeline.

Default options can be overridden for the IP address and port that the Wall Clock client binds to and to reduce the amount of logging output.

Use the `--help` command line option for usage information.

## 1.4 TVDevice.py

### 1.4.1 Get started

This is a very simple example of a server running all three protocols (CSS-WC, CSS-TS and CSS-CII). It pretends to be showing a DVB broadcast service and able to provide a PTS or TEMI timeline for it.

First start the server:

```
$ python examples/TVDevice.py
```

While we leave it running in the background, we can try to interact with it using the various example clients described above.

By default it provides a wall clock server on all interfaces on port 6677

```
$ python examples/WallClockClient.py 127.0.0.1 6677
```

... and a CSS-CII server that can be reached at *ws://{{host}}:7681/cii* where *{{host}}* is any interface, including 127.0.0.1.

```
$ python examples/CIIClient.py ws:/127.0.0.1:7681/cii
```

... and a CSS-TS server that can be reached at *ws://{{host}}:7681/ts*

```
$ python examples/TSClient.py ws://127.0.0.1:7681/ts udp://127.0.0.1:6677 "dvb://"
↪"urn:dvb:css:timeline:temi:1:1" 1000
```

### 1.4.2 How it works

#### TVDevice.py [source]

This example works by setting up a web server and the ws4py plug-in for cherrypy that provides WebSockets support. It then instantiates a *TSServer* and *CIIServer* and mounts it into the cherrypy server. It also includes a wall clock server.

It does not play any media, but instead serves an imaginary set of timelines and pretends to be presenting a broadcast service.

It creates *clock* objects to represent timelines and the wall clock. *SimpleClockTimelineSource* objects are used to interface the clocks as sources of timelines to the TS server object.

It has a hardcoded DVB URL as the content ID (displayed when you start it running) and provides the following timelines:

- *urn:dvb:css:timeline:pts* ... a PTS timeline
- *urn:dvb:css:timelime:temi:1:1* ... a TEMI timeline ticking at 1kHz that toggles availability every 30 seconds

The PTS and TEMI timelines both start ticking up from zero the moment the server starts.

By default, this server binds to 0.0.0.0 on port 7681 and provides a CSS-CII service at the URL *ws://{host}:7681/cii* and a CSS-TS service at the URL *ws://{host}:7681/ts*. It also provides a wall clock server bound to 0.0.0.0 on UDP port 6677.

The CII service will provide URLs for the TS and WC endpoints that match the host name on which the

Command line options can be used to override these defaults and to reduce the amount of logging output.

Use the `--help` command line option for more detailed usage information.

CHAPTER 2

# DVB CSS Protocol modules

Module: *dvbcss.protocol*

The `dvbcss.protocol` module contains classes to implement the CSS-CII, CSS-TS and CSS-WC protocols. For each protocol there are objects to represent the messages that flow across the protocols and classes that implement clients and servers for the protocols.

## 2.1 CSS-CII protocol

### 2.1.1 CSS-CII Protocol introduction

**Here is a quick introduction to the CSS-CII protocol. For full details, refer to the** DVB specification ETSI 103 286 part 2.

The CSS-CII protocol is for sharing the server's (e.g. TV's) current "Content Identifier and other Information" (yes really!) with the client (e.g. companion). It also includes the URL of the *CSS-TS* and *CSS-WC* servers so the client knows where to find them.

CII comprises a set of defined properties. The server pushes state update messages containing some or all properties (at minimum those that have changed). How often these messages are pushed and which properties are included are up to the server.

It is a WebSockets based protocol and messages are in JSON format.

### Sequence of interaction

The client is assumed to already know the WebSocket URL for the CSS-CII server (for example: because the TV chooses to advertise it via a network service discovery mechanism).

1. The client connects to the CSS-TS server. Either this is refused via an HTTP status code response, or it is accepted.
2. The server immediately responds with a first CII state update message. This contains (at minimum) all properties whose values are not *null*.
3. The server can re-send the CII state update message as often as it wishes. At minimum it will do so when one or more of the properties have changed value. The server will, at minimum, include the properties that have changed, but could also include others in the message.

This protocol is a state update mechanism. The client is locally mirroring the state of the TV by remembering the most recent values received for each of the properties.

At the start, the client assumes all properties have the value *null*. Then, when a message is received the client updates its mirror of the TV state:

- If a property is included in the message (even if its value is *null*), then this is the new value for that property.
- If a property is not included in the message, then its value has not changed.

Any messages sent by the client are ignored by the server.

### CII message properties

Every message sent by the server is a CII message and consists of a single JSON object with zero, one, more or all of the following properties:

- `protocolVersion` - currently "1.1" and must be included in the first message sent by the server after the client connects.
- `contentId` - a URI representing the ID of the content being presented by the server. This will be a variant on a DVB URL ("dvb://") for DVB broadcast services, or the URL of the MPD for MPEG DASH streams.
- `contentIdStatus` - whether the content Id is in its "final" form or whether it is a "partial" version until full information is available. For example: a DVB broadcast content ID might not include some elements until the TV detects certain metadata in the broadcast stream which can take a few seconds.
- `presentationStatus` - Primarily, whether presentation of the content is "okay", "transitioning" from one piece of content to the next, or in a "fault" condition. This can be extended by suffixing space separated additional terms after the primary term.
- `mrsUrl` - The URL of an MRS server.
- `tsUrl` - The WebSockets URL of the CSS-TS server that a client should use if it wants to do Timeline Synchronisation.
- `wcUrl` - The UDP URL ("udp://<host>:<port") of the CSS-WC server.
- `teUrl` - The WebSockets URL of the CSS-TE server that a client should use if it wants to receive Trigger Events.
- `timelines` - a list of zero, one or more timelines that the TV believes to be available for synchronising to.
- `private` - Extension mechanism to carry additional private data.

An example *CII* message:

```
{
        "protocolVersion"    : "1.1",
        "mrsUrl"             : "http://css.bbc.co.uk/dvb/233A/mrs",
        "contentId"          : "dvb://233a.1004.1044;363a~20130218T0915Z--PT00H45M",
        "contentIdStatus"    : "partial",
        "presentationStatus" : "okay",
        "wcUrl"              : "udp://192.168.1.5:5800",
        "tsUrl"              : "ws://192.168.1.8:5815",
        "timelines" : [
                {
                        "timelineSelector"   : "urn:dvb:css:timeline:temi:1:1",
                        "timelineProperties" : {
                                "unitsPerTick"   : 5,
                                "unitsPerSecond" : 10
                        }
                }
        ]
}
```

Another example where the contentId has changed, due to a channel change. The server has chosen to omit properties that have not changed since the previous message:

```
{
        "contentId"          : "dvb://233a.1004.1044;364f~20130218T1000Z--PT01H15M",
        "contentIdStatus"    : "partial",
}
```

### 2.1.2 CSS-CII Message objects

Module: *dvbcss.protocol.cii*



A `CII` object represents a CII message sent from server to client via the CSS-CII protocol.

A `TimelineOption` object describes a timeline selector and the tick rate of the timeline if that selector is used to request a timeline from the CSS-TS server. It is carried in a list in the `timelines` property of a `CII` message.

#### Examples

`CII` messages:

```
>>> from dvbcss.protocol.cii import CII
>>> from dvbcss.protocol import OMIT

>>> jsonCiiMessage = \"""\
...     { "protocolVersion":"1.1",
...       "contentId":"dvb://1234.5678.01ab",
```

```
...       "contentIdStatus":"partial"
...     }
... \"""

>>> cii = CII.unpack(jsonCiiMessage)
>>> cii.contentId
'dvb://1234.5678.01ab'

>>> print cii.mrsUrl
OMIT

>>> cii.protocolVersion = OMIT
>>> cii.pack()
'{contentId":"dvb://1234.5678.01ab","contentIdStatus":"partial"}'
```

*TimelineOption* within CII messages:

```
>>> from dvbcss.protocol.cii import CII, TimelineOption

>>> t1 = TimelineOption(timelineSelector="urn:dvb:css:timeline:pts", unitsPerTick=1,␣
↪unitsPerSecond=90000)
>>> t2 = TimelineOption(timelineSelector="urn:dvb:css:timeline:temi:1:1",␣
↪unitsPerTick=1, unitsPerSecond=1000)

>>> print t1.timelineSelector, t1.unitsPerTick, t1.unitsPerSecond, t1.accuracy
urn:dvb:css:timeline:pts 1 90000 OMIT
```

```
>>> cii = CII(presentationStatus="final", timelines=[t1, t2])
>>> cii.pack()
'{ "presentationStatus": "final",
   "timelines": [ { "timelineProperties": {"unitsPerSecond": 90000, "unitsPerTick": 1}
↪,
                    "timelineSelector": "urn:dvb:css:timeline:pts"
                  },
                  { "timelineProperties": {"unitsPerSecond": 1000, "unitsPerTick": 1},
                    "timelineSelector": "urn:dvb:css:timeline:temi:1:1"
                  }
                ]
 }'
```

## Classes

### CII Message

**class** dvbcss.protocol.cii.**CII**(***kwargs*)

Object representing a CII message used in the CSS-CII protocol.

**Initialisation takes the following parameters, all of which are optional keyword arguments that default to** *OMIT* :

**Parameters**

- **protocolVersion** (*OMIT* or "1.1") – The protocol version being used by the server.
- **mrsUrl** (*OMIT* or `str`) – The URL of an MRS server known to the server.

- **contentId** (*OMIT* or str) – Content identifier URI.
- **contentIdStatus** (*OMIT* or "partial" or "final") – Content identifier status.
- **presentationStatus** (*OMIT* or list of str) – Presentation status as a list of one or more strings, e.g. `[ "okay" ]`
- **wcUrl** (*OMIT* or str) – CSS-WC server endpoint URL in the form "udp://<host>:<port>"
- **tsUrl** (*OMIT* or str) – CSS-TS server endpoint WebSocket URL
- **teUrl** (*OMIT* or str) – CSS-TE server endpoint WebSocket URL
- **timelines** (*OMIT* or list of *TimelineOption*) – List of timeline options.
- **private** (*OMIT* or *Signalling that a property is to be omitted from a message*) – Private data.

The attributes of the object have the same name as the CII message properties:

- *protocolVersion*
- *mrsUrl*
- *contentId*
- *contentIdStatus*
- *presentationStatus*
- *wcUrl*
- *tsUrl*
- *teUrl*
- *timelines*
- *private*

Properties are accessed as attributes of this object using the same name as their JSON property name.

Converting to and from JSON representation is performed using the *pack()* method and *unpack()* class method. Properties set to equal *OMIT* will be omitted when the message is packed to a JSON representation.

**protocolVersion = OMIT**
(read/write *OMIT* or "1.1") The protocol version being used by the server.

**mrsUrl = OMIT**
(read/write *OMIT* or str) The URL of an MRS server known to the server

**contentId = OMIT**
(read/write *OMIT* or str) Content identifier (URL)

**contentIdStatus = OMIT**
(read/write *OMIT* or "partial" or "final") Content identifier status

**presentationStatus = OMIT**
(read/write *OMIT* or list of str) Presentation status, e.g. `[ "okay" ]`

**wcUrl = OMIT**
(read/write *OMIT* or str) CSS-WC server endpoint URL in form "udp://<host>:<port>"

**tsUrl = OMIT**
(read/write *OMIT* or str) CSS-TS server endpoint WebSocket URL

**teUrl = OMIT**
(read/write *OMIT* or str) CSS-TE server endpoint WebSocket URL

**timelines = OMIT**
(read/write *OMIT* or list`(:class:`TimelineOption)) Timeline options

**private = OMIT**
(*OMIT* or list of dict ) Private data as a list of dict objects that can be converted to JSON by json.dumps(). Each dict must contain at least a key called "type" with a URI string as its value.

**classmethod allProperties**()
Returns a list of all property names, whether OMITted or not

**combine**(*diff*)
Copies this CII object, and updates that copy with any properties (that are not omitted) in the CII object supplied as the *diff* argument. The updated copy is then returned.

**Parameters** **diff** – (*CII*) A CII object whose properties (that are not omitted) will be used to update the copy before it is returned.

new = old.combine(diff) is equivalent to the following operations:

```
new = old.copy()
new.update(diff)
```

**copy**()
**Returns** a copy of this CII object. The copy is a deep copy.

**definedProperties**()
Returns a list of the names of properties whose value is not OMIT

**classmethod diff**(*old*, *new*)
**Parameters**
- **old** – (*CII*) A CII object
- **new** – (*CII*) A CII object

**Returns** CII object representing changes from old to new CII objects.

If in the new CII object a property is OMITted, it property won't appear in the returned CII object that represents the changes.

If in the old CII object a property is OMITted, but it has a non-omitted value in the new object, then it is assumed to be a change.

**pack**()
**Returns** string containing JSON representation of this message.
**Throws ValueError** if there are values for properties that are not permitted.

**classmethod unpack**(*msg*)
Convert JSON string representation of this message encoded as a *CII* object.
**Throws ValueError** if not possible.

**update**(*diff*)
Updates this CII object with the values of any properties (that are not omitted) in the CII object provided as the *diff* argument.

Note that this changes this object.

**Parameters** **diff** – (*CII*) A CII object whose properties (that are not omitted) will be used to update this CII object.

### Timeline Option

**class** dvbcss.protocol.cii.**TimelineOption**(*timelineSelector*, *unitsPerTick*, *unitsPerSecond*, *accuracy=OMIT*, *private=OMIT*)

Object representing a CSS-CII Timeline Option used in the "timelines" property of a CII message.

**Initialisation takes the following parameters:**

**Parameters**

- **timelineSelector** (*str*) – The timeline selector
- **unitsPerTick** (*int*) – Denominator of tick rate (in ticks per second) for the corresponding timeline
- **unitsPerSecond** (*int*) – Numerator of tick rate (in ticks per second) for the corresponding timeline
- **accuracy** (*OMIT* or float) – Optional indication of timeline accuracy
- **private** (*OMIT* or *Signalling that a property is to be omitted from a message*) – Optional private data.

It represents a timeline selector and the tick rate of the timeline if that selector is used to request a timeline from the CSS-TS server. It is carried in a list in the *timelines* property of a *CII* message.

The tick rate of the timeline is expressed by the *unitsPerTick* and *unitsPerSecond* values. The tick rate in ticks per second is equal to *unitsPerTick* / *unitsPerSecond*.

Accuracy and private data are optional, but the other fields are mandatory.

The attributes of the object have the same name as the relevant CII message properties:

- *timelineSelector*
- *unitsPerTick*
- *unitsPerSecond*
- *accuracy*
- *private*

Converting to and from JSON representation is performed using the *pack()* method and *unpack()* class method. Properties set to equal *OMIT* will be omitted when the message is packed to a JSON representation.

**timelineSelector = OMIT**
(str) The timeline selector

**unitsPerTick = OMIT**
(int) The units per tick of the timeline

**unitsPerSecond = OMIT**
(int) The units per second of the timeline

**accuracy = OMIT**
(*OMIT* or float) The accuracy of the timeline with respect to the content in seconds.

**private = OMIT**
(*OMIT* or list of dict ) Private data as a list of dict objects that can be converted to JSON by json.dumps(). Each dict must contain at least a key called "type" with a URI string as its value.

**classmethod decode**(*struct*)
Internal method used by a `CII` message object when unpacking to JSON format.

**classmethod encode**(*item*)
Internal class method used by a `CII` message object when packing to JSON format.

**pack**()
**Returns** string containing JSON presentation of this message.

**classmethod unpack**(*msg*)
Convert JSON string representation of this message encoded as a `TimelineOption` object.
**Throws ValueError** if not possible.

### 2.1.3 CSS-CII Clients

Module: *dvbcss.protocol.client.cii*



There are two classes provided for implementing CSS-CII clients:

- `CIIClient` connects to a CII server and provides a CII message object representing the complete state of the server and notifies you of changes of state.
- `CIIClientConnection` provides a lower level connection to a CII server and only provides the messages received from the server. It does not maintain a model of the server state and does not work out when a received message constitutes a change.

An *example* client is provided in this package that uses the `CIIClient` class.

#### Using CIIClient

This is the simplest class to use. Create it, passing the URL of the server to connect to, then call `connect()` and `disconnect()` to connect and disconnect from the server.

CIIClient maintains a local copy of the state of CII data the `CIIClient.cii` property and the most recently received CII message in `CIIClient.latestCII`.

You can use the class either by subclassing and overriding the various stub methods or by creating an instance and replacing the stub methods with your own function handlers dynamically.

Pass the WebSocket URL of the CII server when creating the *CIIClient* then call the `connect()` and `disconnect()` methods to connect and disconnect from the server. The *onXXX()* methods can be overriden to find out when connection or disconnection takes place, if there is a protocol error (e.g. a message was received that could not be parsed as CII) or when properties of the CII data change.

The CII state is kept in the `cii` property of the object. This is updated with properties in CII messages that are received. Properties not included in a CII message are left unchanged.

Properties of the CII state whose value is `dvbcss.protocol.OMIT` have not been defined by the CII server.

```
from dvbcss.protocol.client.cii import CIIClient

class MyCIIClient(CIIClient):

    def onConnected(self):
        print "Connected!"

    def onDisconnected(self, code, reason):
        print "Disconnected :-("

    def onChange(self, propertyNames):
        print "The following CII properties have changed:
        for name in propertyNames:
            value = getattr(conn.cii, name)
            print "    "+name+" is now: "+str(value)

    # one example of a handler for changes to a particular property 'contentId' in␣
↪CII
    def onContentIdChange(self, newValue):
        print "The contentId property has changed to now be: "+str(newValue)



client = MyCIIClient("ws://127.0.0.1/cii")
client.connect()

time.sleep(60)

print "The current contentId is "+client.cii.contentId

time.sleep(60)    # wait only 60 more seconds then disconnect

client.disconnect()
```

The client runs in a separate thread managed by the websocket client library, so the *onXXX* methods are called while the main thread sleeps.

### Using CIIClientConnection

This is a lower level class, that only implements parsing of the incoming CII messages from the server. It does not detect if a message actually constitutes a change of state or not.

You can use the class either by subclassing and overriding the various stub methods or by creating an instance and replacing the stub methods with your own function handlers dynamically.

Pass the WebSocket URL of the CII server when creating the *CIIClientConnection* object then call the `connect()` and `disconnect()` methods to connect and disconnect from the server. The *onXXX()* methods can be overridden to find out when connection or disconnection takes place, if there is a protocol error (e.g. a message was received that could not be parsed as CII) or when a new CII message is received.

```
from dvbcss.protocol.client.cii import CIIClientConnection

class MyCIIClientConnection(CIIClientConnection):

    def onConnected(self):
        print "Connected!"
```

```
    def onDisconnected(self, code, reason):
        print "Disconnected :-("

    def onCii(self, cii):
        print "Received a CII message: "+str(cii)



client = MyCIIClientConnection("ws://127.0.0.1/cii")
client.connect()

time.sleep(60)      # run only for 60 seconds then disconnect

client.disconnect()
```

## Classes

### CIIClient

**class** dvbcss.protocol.client.cii.**CIIClient**(*ciiUrl*)

Manages a CSS-CII protocol connection to a CSS-CII Server and notifies of changes to CII state.

Use by subclassing and overriding the following methods:

- *onConnected()*
- *onDisconnected()*
- *onChange()*
- individual *onXXXXChange()* methods named after each CII property
- *onCiiReceived()* (do not use, by preference)

If you do not wish to subclass, you can instead create an instance of this class and replace the methods listed above with your own functions dynamically.

The *connect()* and *disconnect()* methods connect and disconnect the connection to the server and `getStatusSummary()` provides a human readable summary of CII state.

This object also provides properties you can query:

- *cii* represents the current state of CII at the server
- *latestCII* is the most recently CII message received from the server
- *connected* indicates whether the connection is currently connect

**Initialisation takes the following parameters:**

**Parameters** **ciiUrl** – (*str*) The WebSocket URL of the CSS-CII Server (e.g. "ws://127.0.0.1/myservice/cii")

**connected**

True if currently connected to the server, otherwise False.

**cii**

(*CII*) CII object representing the CII state at the server

**latestCII**
(*CII* or None) The most recent CII message received from the server or None if nothing has yet been received.

**connect**()
Start the client by trying to open the connection.

**Throws ConnectionError** There was a problem that meant it was not possible to connect.

**disconnect**()
Disconnect from the server.

**onChange**(*changedPropertyNames*)
This method is called when a CII message is received from the server that causes one or more of the CII properties to change to a different value.

**Parameters** **changedPropertyNames** – A list of str names of the properties that have changed. Query the *cii* attribute to find out the new values.

**onCiiReceived**(*newCii*)
This method is called when a CII message is received, but before any 'onXXXXChange()' handlers (if any) are called. It is called even if the message does not result in a change to CII state held locally.

By preference is recommended to use the 'onXXXXChange()' handlers instead since these will only be called if there is an actual change to the value of a property in CII state.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **cii** – A *CII* object representing the received message.

**onConnected**()
This method is called when the connection is opened.

*This is a stub for this method. Sub-classes should implement it.*

**onContentIdChange**(*newValue*)
Called when the contentId property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

**onContentIdStatusChange**(*newValue*)
Called when the contentIdStatus property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

**onDisconnected**(*code*, *reason=None*)
This method is called when the connection is closed.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters**

- **code** – (int) The connection closure code to be sent in the WebSocket disconnect frame
- **reason** – (str or None) The human readable reason for the closure

**onMrsUrlChange**(*newValue*)
Called when the mrsUrl property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

**onPresentationStatusChange**(*newValue*)
Called when the presentationStatus property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

**onPrivateChange**(*newValue*)
Called when the private property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

**onProtocolError**(*msg*)
This method is called when there has been an error in the use of the CII protocol - e.g. receiving the wrong kind of message.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **msg** – A `str` description of the problem.

**onProtocolVersionChange**(*newValue*)
Called when the protocolVersion property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

**onTeUrlChange**(*newValue*)
Called when the teUrl property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

**onTimelinesChange**(*newValue*)
Called when the timelines property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

**onTsUrlChange**(*newValue*)
Called when the tsUrl property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

**onWcUrlChange**(*newValue*)
Called when the wcUrl property of the CII message has been changed by a state update from the CII Server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **newValue** – The new value for this property.

### CIIClientConnection

**class** dvbcss.protocol.client.cii.**CIIClientConnection**(*url*)

Simple object for connecting to a CSS-CII server and handling the connection.

Use by subclassing and overriding the following methods:

- *onConnected()*
- *onDisconnected()*
- *onCII()*
- *onProtocolError()*

If you do not wish to subclass, you can instead create an instance of this class and replace the methods listed above with your own functions dynamically.

**Initialisation takes the following parameters:**

**Param** url (str) The WebSocket URL of the CII Server to connect to. E.g. "ws://127.0.0.1/mysystem/cii"

**connect**()

Open the connection.

:throws ConnectionError if there was a problem and the connection could not be opened.

**connected**

True if the connection is connect, otherwise False

**disconnect**(*code=1001*, *reason=''*)

Close the connection.

**Parameters**

- **code** – (optional int) The connection closure code to be sent in the WebSocket disconnect frame
- **reason** – (optional str) The human readable reason for the closure

**onCII**(*cii*)

This method is called when a CII message is received from the server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **cii** – A *CII* object representing the received message.

**onConnected**()

This method is called when the connection is opened.

*This is a stub for this method. Sub-classes should implement it.*

**onDisconnected**(*code*, *reason=None*)

This method is called when the connection is closed.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters**

- **code** – (int) The connection closure code to be sent in the WebSocket disconnect frame
- **reason** – (str or None) The human readable reason for the closure

**onProtocolError**(*msg*)
This method is called when there has been an error in the use of the CII protocol - e.g. receiving the wrong kind of message.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **msg** – A `str` description of the problem.

### 2.1.4 CSS-CII Servers

Module: *dvbcss.protocol.server.cii*



The `CIIServer` class implements a CII server that can be plugged into the cherrypy web server engine.

To create a CII Server, first create and mount the server in a cherrypy web server. Then you can start the cherrypy server and the CII server will start to accept connections from clients. While the server is running, update the CII state maintained by that server and instruct it when to push updates to all connected clients.

An *example* server is provided in this package.

#### Using CII Server

##### 1. Imports and initialisation

To run a CII server, you must import both ws4py's cherrypy server and the *dvbcss.protocol.server.cii* module. When the *dvbcss.protocol.server.cii* module is imported, it will register as a "tool" with cherrypy, so it must be imported after cherrypy is imported.

Next, subscribe the ws4py websocket plugin to cherrypy.

```
import cherrypy
from ws4py.server.cherrypyserver import WebSocketPlugin
from dvbcss.protocol.server.cii import CIIServer

# initialise the ws4py websocket plugin
WebSocketPlugin(cherrypy.engine).subscribe()
```

### 2. Create and mount the CII server

You can now create an instance of a CIIServer and mount it into the cherrypy server at a path of your choosing.

The configuration for that path must turn on the "dvb_cii" tool and pass a "handler_cls" argument whose value is the handler class that the CIIServer instance provides via the *CIIServer.handler* attribute.

For example, to create a CIIServer mounted at the URL path "/cii":

```
# create CII Server
ciiServer = CIIServer(maxConnectionsAllowed=2)

# bind it to the URL path /cii in the cherrypy server
class Root(object):
    @cherrypy.expose
    def cii(self):
        pass

# construct the configuration for this path, providing the handler and turning on the␣
↪tool hook
cfg = {"/cii": {'tools.dvb_cii.on': True,
                'tools.dvb_cii.handler_cls': ciiServer.handler
               }
      }

cherrypy.tree.mount(Root(), "/", config=cfg)
```

### 3. Start cherrypy running

Start cherrypy running and our CII server will start to accept connections from clients:

```
# configure cherrypy to serve on port 7681
cherrypy.config.update({"server.socket_port":7681})

# activate cherrypy web server (non blocking)
cherrypy.engine.start()
```

The cherrypy engine runs in a background thread when the cherrypy engine is started.

### 4. Setting CII state and pushing it to connected clients

The *CIIServer.cii* is a CII message object representing the CII state. Your code can read and alter the attributes of this message object to update the server side state.

When a client first connects, a CII message object will automatically be sent to that client to send it the current CII state. Your code does not need to do this.

If you update the CII state then you need to ask the CII server to push a change to all connected clients. To do this call the *CIIServer.updateClients()* method. By default this will only push changes to CII state, and will not send a message at all if there is no change. However this behaviour can be overridden.

```
ciiServer.cii.contentId = "dvb://233a.1004.1080"
ciiServer.cii.contentIdStatus = "partial"
ciiServer.updateClients()

...
```

```
ciiServer.cii.contentId = "dvb://233a.1004.1080;21af~20131004T1015Z--PT01H00M"
ciiServer.cii.contentIdStatus = "final"
ciiServer.updateClients()
```

### Intelligently setting the host and port in tsUrl, teUrl and wcUrl properties

*CIIServer* has built in support to help in situations where it is difficult to determine the host and port to which clients are connecting in order to contact the CII Server, or where CIIServer might be contacted via more than one network interface.

At initialisation, pass the optional *rewriteHostPort* argument, setting it to a list of properties for which you want it to fix the host/port in URLs. Then within the CII, put *{{host}}* and *{{port}}* in place of the host and port number. The CIIServer will then automatically substitute this in the properties you have listed.

For example:

```
ciiServer = CIIServer(rewriteHostPort=['tsUrl','wcUrl'])
ciiServer.cii = CII(
    tsUrl='ws://{{host}}:{{port}}/ts',
    wcUrl='udp://{{host}}:6677'
)
```

This will be done transparently and individually for each connected client. The *cii* property of the CII server will contain the *{{host}}* and *{{port}}* patterns before the substitution takes place.

### What does CIIServer do for you and what does it not?

*CIIServer* handles the connection and disconnection of clients without requiring any further intervention. It ensure the current state in its *cii* property is sent, in a CII message, to the client as soon as it connects.

The role of your code is to update the *cii* object as state changes, and to inform the CIIServer when it is time to update any connected clients by informing them of the changes to state by calling the *updateClients()* method.

### Classes

### CIIServer - CII Server handler for cherrypy

**class** dvbcss.protocol.server.cii.**CIIServer**(*maxConnectionsAllowed=-1*, *enabled=True*, *initialCII=CII(protocolVersion='1.1')*, *rewriteHostPort=[]*)

The CIIServer class implements a server for the CSS-CII protocol. It transparently manages the connection and disconnection of clients and provides an interface for simply setting the CII state and requesting that it be pushed to any connected clients.

Must be used in conjunction with a cherrypy web server:

1. Ensure the ws4py `WebSocketPlugin` is subscribed, to the cherrypy server. E.g.

   ```
   WebSocketPlugin(cherrypy.engine).subscribe()
   ```

2. Mount the instance onto a particular URL path on a cherrypy web server. Set the config properties for the URL it is to be mounted at as follows:

```
{ 'tools.dvb_cii.on'          : True,
  'tools.dvb_cii.handler_cls': myCiiServerInstance.handler }
```

Update the *cii* property with the CII state information and call the *updateClients()* method to propagate state changes to any connected clients.

When the server is "disabled" it will refuse attempts to connect by sending the HTTP status response 403 "Forbidden".

When the server has reached its connection limit, it will refuse attempts to connect by sending the HTTP status response 503 "Service unavailable".

This object provides properties:

- *enabled* (read/write) controls whether this server is enabled or not
- *cii* (read/write) the CII state that is being shared to connected clients

To allow for servers serving multiple network interfaces, or where the IP address of the interface is not easy to determine, CII Server can be asked to automatically substitute the host and port with the one that the client connected to. Specify the list of property names for which this shoud happen as an optional *rewritheostPort* argument when intialising the CIIServer, then use *{{host}} {{port}}* within those properties.

**Initialisation takes the following parameters:**

**Parameters**

- **maxConnectionsAllowed** – (int, default=-1) Maximum number of concurrent connections to be allowed, or -1 to allow as many connections as resources allow.
- **enabled** – (bool, default=True) Whether the endpoint is initially enabled (True) or disabled (False)
- **initialCII** – (*dvbcss.protocol.cii.CII*, default=CII(protocolVersion="1.1")) Initial value of CII state.
- **rewriteHostPort** – (list) List of CII property names for which the sub-string '{{host}}' '{{port}}' will be replaced with the host and port that the client connected to.

**cii**

**handler**
Handler class for new connections.

When mounting the CII server with cherrypy, include in the config dict a key 'tools.dvb_cii.handler_cls' with this handler class as the value.

**enabled**
(read/write `bool`) Whether this server endpoint is enabled (True) or disabled (False).

Set this property enable or disable the endpoint.

When disabled, existing connections are closed with WebSocket closure reason code 1001 and new attempts to connect will be refused with HTTP response status code 403 "Forbidden".

**getConnections**()

**Returns dict** mapping a `WebSocket` object to connection related data for all connections to this server. This is a snapshot of the connections at the moment the call is made. The dictionary is not updated later if new clients connect or existing ones disconnect.

**onClientConnect**(*webSock*)
If you override this method you must call the base class implementation.

**onClientDisconnect**(*webSock*, *connectionData*)
If you override this method you must call the base class implementation.

**onClientMessage**(*webSock*, *message*)
If you override this method you must call the base class implementation.

**updateClients**(*sendOnlyDiff=True*, *sendIfEmpty=False*)
Send update of current CII state from the *CIIServer.cii* object to all connected clients.

**Parameters**

- **sendOnlyDiff** – (bool, default=True) Send only the properties in the CII state that have changed since last time a message was sent. Set to False to send the entire message.
- **sendIfEmpty** – (bool, default=False) Set to True to force that a CII message be sent, even if it will be empty (e.g. no change since last time)

By default this method will only send a CII message to clients informing them of the differencesin state since last time a message was sent to them. If no properties have changed at all, then no message will be sent.

The two optional arguments allow you to change this behaviour. For example, to force the messages sent to include all properties, even if they have not changed:

```
myCiiServer.updateClients(sendOnlyDiff=False)
```

To additionally force it to send even if the CII state held at this server has no values for any of the properties:

```
myCiiServer.updateClients(sendOnlyDiff=False, sendIfEmpty=True)
```

This package provides objects for representing messages exchanged via the DVB CSS-CII protocol and for implementing clients and servers.

The CII protocol is a mechanism for sending state updates from server to client. The state of the server can be represented by a *CII* message where every property is populated with a value. The server can send complete CII messages or partial ones containing only the properties that have changed value since the last message. The client must track these changes to maintain its own local up-to-date copy of the complete state.

Modules for using the CSS-CII protocol:

- *dvbcss.protocol.cii* : objects for representing and packing/unpacking the CSS-CII protocol messages.
- *dvbcss.protocol.client.cii* : implementations of a client for a CSS-CII connection.
- *dvbcss.protocol.server.cii* : implementations of a server for a CSS-CII connection.

## 2.2 CSS-TS protocol

### 2.2.1 CSS-TS Protocol introduction

**Here is a quick introduction to the CSS-TS protocol. For full details, refer to the** DVB specification ETSI 103 286 part 2.

The CSS-TS protocol is for *Timeline Synchronisation*. Via this protocol, the server (e.g. TV) pushes timestamps to the client (e.g. companion) to keep it up-to-date on the progress of a particular timeline. The timeline to use is requested by the client at the beginning of the interaction.

A client can also report its own timing and what range of timings it can cope with. This allows the client to negotiate a mutually achievable timing with the server, although the server is under no obligation and can choose to ignore this information.

It is a WebSockets based protocol and messages are in JSON format.



### Sequence of interaction

The client is assumed to already know the WebSocket URL for the CSS-TS server (usually from the information received via the *CSS-CII protocol*).

1. The client connects to the CSS-TS server. Either this is refused via an HTTP status code response, or it is accepted.

2. The client then immediately sends an initial `SetupData` message to request the timeline to synchronise with.

3. The server then starts sending back `ControlTimestamp` messages that update the client as to the state of that timeline. This state says either that the timeline is currently unavailable, or that it is available, and here is how to calculate the timeline position from the wall clock position.The server sends as frequently or infrequently as it likes, but will at least send them if there is a meaningful change in the timeline.

4. The client can, optionally, send its own `AptEptLpt` messages to inform the server of what it is doing, and the range of different timings it can achieve for its media (e.g. what is the earliest and latest timings it can achieve). However this is purely informative. A server is not obliged to do anything with this information.

### Determining timeline selection and availability

The `SetupData` message conveys to the CSS-TS server details of what timeline the client wants to synchronise to.

The CSS-TS server determines, at any given moment, if a timeline is available by checking if:

1. the stem matches the current content identifier for what is being presented at the server (meaning that the stem matches the left hand most subset of the content id);
2. and the timeline selector identifies a timeline that exists for the content being presented at the server.

While the above is true, the timeline is "available". While it is not true, it is "unavailable". The CSS-TS connection is kept open irrespective of timeline availability. The server indicates changes in availability via the `ControlTimestamp` messages it sends.

Example `SetupData` message; requesting a PTS timeline for a particular DVB broadcast channel, but not being specific about which event (programme in the EPG):

```
{
        "contentIdStem"     : "dvb://233a.1004.1044",
        "timelineSelector" : "urn:dvb:css:timeline:pts"
}
```

### What does a timestamp convey?

It represents a relationship between Wall Clock time and the timeline of the content being presented by the TV Device. It is sometimes referred to as a *(point of) correlation* between the wall clock and the timeline.

This relationship can be visualised as a line that maps from wall clock time (on one axis) to timeline time (on the other axis). The (content-time, wall-clock-time) correlation is a point on the line. The timelineSpeedMultiplier represents the slope. The tick rates of each timeline are the units (the scale of each axis).

The CSS-TS server sends *ControlTimestamp* messages to clients, and clients can, optionally, send back `AptEptLpt` messages.

A ControlTimestamp can also tell a client if a timeline is unavailable by having null values for the contentTime and timelineSpeedMultiplier properties. Non-null values mean the timeline is available.

*AptEptLpt* messages enables a client to inform a server of what time it is presenting at (the "actual" part of the timestamp) and also to indicate the earliest and latest times it could present. It is, in effect, three correlations bundled into one message, to represent each of these three aspects. Earliest and Latest correlations are allowed to have -infinity and +infinity for the wall clock time to indicate that the client has no limits on how early, or late, it can present.

An example *ControlTimestamp* indicating the timeline is unavailable:

```
{
        "contentTime"             : null,
        "wallClockTime"           : "116012000000",
        "timelineSpeedMultiplier" : null
}
```

An example *ControlTimestamp* providing a correlation for an available timeline:

```
{
        "contentTime"             : "834188",
        "wallClockTime"           : "116012000000",
        "timelineSpeedMultiplier" : 1.0
}
```

An example of an *AptEptLpt* message, indicating the current presentation timing being used by the client; a limit on how early it can present; but no limit on how long it can delay (buffer):

```
{
        "actual" : {
                "contentTime"   : "834190",
                "wallClockTime" : "115992000000"
        },
        "earliest" : {
                "contentTime"   : "834190",
                "wallClockTime" : "115984000000"
        },
        "latest" : {
                "contentTime"   : "834190",
                "wallClockTime" : "plusinfinity"
        }
}
```

### 2.2.2 CSS-TS Message objects

Module: *dvbcss.protocol.ts*

- *Examples*
- *+/- infinity*

A *SetupData* object represents a setup-data message sent by a client to a server immediately after opening a CSS-TS protocol connection.

A *ControlTimestamp* object represents a Control Timestamp message sent by the server to the client.

A *AptEptLpt* object represents an Actual, Earliest and Latest Presentation Timestamp message that may be sent by a client to the server.

The *Timestamp* objects are used in the above message objects to represent the relationship between wall clock time and content (timeline) time.

### Examples

SetupData examples:

```
>>> from dvbcss.protocol.ts import SetupData
>>> from dvbcss.protocol import OMIT

>>> s = SetupData(timelineSelector="urn:dvb:css:timeline:pts", ciStem="dvb://1004")
>>> s.pack()
'{"timelineSelector": "urn:dvb:css:timeline:pts", "contentIdStem": "dvb://1004"}'
```

```
>>> jsonMessage = \"""\
... { "timelineSelector":"urn:dvb:css:timeline:temi:1:1",
...   "contentIdStem":""
... }
... \"""
>>> SetupData.unpack(jsonMessage)
SetupData(ciStem="", timelineSelector="urn:dvb:css:timeline:temi:1:1", private=OMIT)
```

ControlTimestamp examples:

```
>>> from dvbcss.protocol.ts import ControlTimestamp, Timestamp

>>> t = Timestamp(contentTime=12345, wallClockTime=900028432)
>>> ct = ControlTimestamp(timestamp=t, timelineSpeedMultiplier=1)
>>> ct.pack()
'{"timelineSpeedMultiplier": 1.0, "wallClockTime": "900028432", "contentTime": "12345
↪"}'
```

```
>>> jsonMessage = \"""\
... { "contentTime" : "1003847",
...   "wallClockTime" : "348957623498576",
...   "timelineSpeedMultiplier" : 2.0
... }
... \"""
>>> c = ControlTimestamp.unpack(jsonMessage)
>>> c.timestamp.contentTime
```

```
1003847
>>> c.timestamp.wallClockTime
348957623498576
>>> c.timelineSpeedMultiplier
2.0
```

Actual, Earliest and Latest Presentation Timestamp examples:

```
>>> from dvbcss.protocol.ts import AptEptLpt, Timestamp

>>> te = Timestamp(contentTime=123465, wallClockTime=float("-inf"))
>>> tl = Timestamp(contentTime=123465, wallClockTime=float("+inf"))
>>> ael = AptEptLpt(earliest=te, latest=tl)
>>> ael.pack()
'{"earliest": {"wallClockTime": "minusinfinity", "contentTime": "123465"}, "latest": {
↪"wallClockTime": "plusinfinity", "contentTime": "123465"}}'
```

```
>>> jsonMessage = \"""\
... { "earliest" : { "contentTime" : "1000", "wallClockTime": "10059237" },
...   "latest"   : { "contentTime" : "1000", "wallClockTime": "19284782" },
...   "actual"   : { "contentTime" : "1005", "wallClockTime": "10947820" }
... }
... \"""
>>> ael=AptEptLpt.unpack(jsonMessage)
>>> ael.actual.contentTime
1005
>>> ael.actual.wallClockTime
10947820
```

#### +/- infinity

For certain timestamp messages it is permissible to convey a time value that is either plus or minus infinity. Use the python `float` to express these values as follows:

```
>>> float("+inf")
inf

>>> float("-inf")
-inf
```

### Classes

#### setup-data

**class** dvbcss.protocol.ts.**SetupData**(*contentIdStem*, *timelineSelector*, *private=OMIT*)

Object representing a CSS-TS Setup-Data message.

This carries a content identifier stem and a timeline selector string, and is used, in effect, to request the timeline to be synchronised to via the CSS-TS protocol.

**Initialisation takes the following parameters:**

**Parameters**

- **contentIdStem** (*str*) – The content identifier stem.
- **timelineSelector** (*str*) – The timeline selector
- **private** (*OMIT* or *Signalling that a property is to be omitted from a message*) – Optional private data.

The attributes of the object have the same name as the SetupData message properties:

- *contentIdStem*
- *timelineSelector*
- *private*

Converting to and from JSON representation is performed using the *pack()* method and *unpack()* class method. Properties set to equal *OMIT* will be omitted when the message is packed to a JSON representation.

**contentIdStem**
(read/write str) The stem (subset starting from the LHS) of a content identifier

**timelineSelector**
(read/write str) The timeline selector

**private = OMIT**
(read/write *OMIT* or *Signalling that a property is to be omitted from a message*) Optional private data.

**copy**()
**Returns** a copy of this SetupData object. Note that this does NOT deep copy any private data.

**pack**()
**Returns** string containing JSON representation of this message.
**Throws ValueError** if there are values for properties that are not permitted.

**classmethod unpack**(*msg*)
Convert JSON string representation of this message encoded as a *SetupData* object.
**Throws ValueError** if not possible.

### Control Timestamp

**class** dvbcss.protocol.ts.**ControlTimestamp**(*timestamp*, *timelineSpeedMultiplier*)
Object representing a CSS-TS Control Timestamp message.

**Initialisation takes the following parameters:**

**Parameters**

- **timestamp** (Timestamp) – carries the *contentTime* and *wallClockTime* properties of the Control Timestamp
- **timelineSpeedMultiplier** (float or None) – the timeline speed multiplier

The attributes of the object have the following relationship to the message properties:

- *timestamp*
- *timelineSpeedMultiplier*

Converting to and from JSON representation is performed using the *pack()* method and *unpack()* class method.

**timestamp**
(read/write *Timestamp*) *Timestamp* object representing the contentTime and wallClockTime parts of the timestamp)

**timelineSpeedMultiplier**
(read/write `float` or `None`) Timeline speed. For example: 1 = normal, 0 = pause, -0.5 = half speed reverse. Use *None* only when the Control Timestamp is supposed to indicate that the timeline is unavailable.

**copy**()
**Returns** a deep copy of this Control Timestamp object

**pack**()
**Returns** string containing JSON representation of this message.
**Throws ValueError** if there are values for properties that are not permitted.

**classmethod unpack**(*msg*)
Convert JSON string representation of this message encoded as a *ControlTimestamp* object.
**Throws ValueError** if not possible.

### AptEptLpt (Actual, Earliest and Latest Presentation Timestamp)

**class** dvbcss.protocol.ts.**AptEptLpt**(*actual=OMIT*, *earliest=Timestamp(contentTime=0, wallClockTime=-inf)*, *latest=Timestamp(contentTime=0, wallClockTime=inf)*)
Object representing a CSS-TS Actual, Earliest and Latest Presentation Timestamp message.

**Initialisation takes the following parameters:**

**Parameters**

- **actual** (*OMIT* or *Timestamp*) – Optional timestamp representing the actual presentation timing
- **earliest** (*OMIT* or *Timestamp*) – Timestamp representing the earliest possible presentation timing
- **latest** (*OMIT* or *Timestamp*) – Timestamp representing the latest possible presentation timing

For the *actual presentation timestamp*, the contentTime and wallClockTime must both be non-null integer values.

For the *earliest presentation timestamp*, the contentTime must be a non-null integer. wallClockTime can be a non-null integer or *plus infinity*

For the *latest presentation timestamp*, the contentTime must be a non-null integer. wallClockTime can be a non-null integer or *minus infinity*

The attributes of the object have the same names as the Actual, Earliest and Latest presentation timestamp message properties:

- *actual*
- *earliest*

- *latest*

Converting to and from JSON representation is performed using the *pack()* method and *unpack()* class method. If values of properties do not meet the requirements described above, then *pack()* will raise `ValueError` exceptions.

**actual**
(read/write `OMIT` or *Timestamp*) Actual presentation timestamp

**earliest**
(read/write *Timestamp*) Earliest presentation timestamp. The wallClockTime property must be an `int` or `float("+inf")`. It must not be `float("-inf")`.

**latest**
(read/write *Timestamp*) Latest presentation timestamp. The wallClockTime property must be an `int` or `float("-inf")`. It must not be `float("+inf")`.

**copy**()
**Returns** a deep copy of this AptEptLpt object

**pack**()
**Returns** string containing JSON representation of this message.
**Throws ValueError** if there are values for properties that are not permitted.

**classmethod unpack**(*msg*)
Convert JSON string representation of this message encoded as a *AptEptLpt* object.
**Throws ValueError** if not possible.

### Timestamp

This object does not directly represent a message, but is instead used by *ControlTimestamp* and *AptEptLpt* as a representation of a correlation between a content time and a wall clock time.

**class** dvbcss.protocol.ts.**Timestamp**(*contentTime*, *wallClockTime*)
Object representing a Timestamp part(s) of a *ControlTimestamp* or *AptEptLpt* object.

**Initialisation takes the following parameters:**

**Parameters**

- **contentTime** (`None` or `int`) – The content time (time on timeline) part of a timestamp
- **wallClockTime** (`int` or *+/- infinity* `float` ("+inf") or `float` ("-inf")) – The wall clock time part of a timestamp

The values for contentTime and wallClockTime are allowed to be arbitrarily large precision integers because they are carried as a string in the JSON representation.

The attributes of the object have the same name as the corresponding message properties:

- *contentTime*
- *wallClockTime*

Converting to and from JSON representation is performed using the `pack()` method and `unpack()` class method.

**contentTime**
(read/write `None` or large `int`) The content time part of a timestamp

**wallClockTime**
(read/write large `int` or `float("+inf")` or `float("-inf")` ) The wall clock time part of a timestamp

### 2.2.3 CSS-TS Clients

Module: *dvbcss.protocol.client.ts*



There are two classes provided for implementing CSS-TS clients:

- `TSClientClockController` wraps a TSClientConnection and provides a higher level interface that provides information about timeline availability and drives a `clock` object to match the timeline.
- `TSClientConnection` implements the core functionality of connecting to a CSS-TS server and providing methods and callbacks to manage the connection and send and receive `timestamp` messages.

An *example* client is provided in this package that uses the `TSClientClockController` class.

#### Using TSClientClockController

This class provides a high level interface that drives a `CorrelatedClock` object to match the timeline provided by a CSS-TS server. Subclass to get notifications of connection, disconnection and timing changes or changes in timeline availability.

Create it, passing the URL of the server to connect to, and a `connect` and `disconnect()` to connect and disconnect from the server.

The clock you provide must be a `CorrelatedClock` object and the parent clock of that clock must represent the *wall clock*. The tick rate of this clock must match that of the timeline and the tick rate of the wall clock must be 1 tick per nanosecond (1e9 ticks per second). This should therefore be used in conjunction with a `WallClockClient` to ensure the wall clock is meaningfully synchronised to the same server.

The `TSClientClockController` object maintains the connection and automatically adjusts the `correlation` of the clock to synchronise it, using the information in the Control Timestamps received from the server. It adjusts the `speed()` to match speed changes of the timeline. It also sets the availability of the clock to reflect the availability of the timeline.

You can set a minimum threshold for how much the timing must change before the the timeline clock will be adjusted.

You can also provide separate clock objects to represent the earliest and latest presentation timings that your application can achieve (and that you wish to convey to the server). These must also be `CorrelatedClock` objects. Use the `TSClientClockController.sendAptEptLpt()` method to cause that information to be sent to the server.

A simple example:

```
from dvbcss.protocol.client.ts import TSClientClockController
from dvbcss.clock import CorrelatedClock, SysClock
```

```
sysClock = SysClock()
wallClock = CorrelatedClock(parent=sysClock, tickRate=1000000000)  # 1 nanosecond per␣
↪tick
timelineClock = CorrelatedClock(parent=wallClock, tickRate=90000)  # will represent␣
↪PTS

# NOTE: need a Wall Clock Client too (not shown in this example)
#       to ensure wallClock is synchronised

class MyTSClient(TSClientClockController):

    def onConnected(self):
        print "Conected!"

    def onDisconnected(self):
        print "Disconnected :-("

    def onTimelineAvailable(self):
        print "Timeline is available!"

    def onTimelineUnavailable(self):
        print "Timeline is not available :-("

    def onTimingChange(self, speedHasChanged):
        print "Timing of clock has been adjusted."
        if speedHasChanged:
            print "The speed of the clock has altered."

# make connection. Want PTS timeline for any DVB service.
client = MyTSClient("ws://192.168.1.1:7682/ts",
                    "dvb://",
                    "urn:dvb:css:timeline:pts",
                    timelineClock)
client.connect()

for i in range(0,60):
    time.sleep(1)
    print "Timeline available?", timelineClock.isAvailable()
    if timelineClock.isAvailable():
        print "Timeline currently at tick value = ", timelineClock.ticks
        print "          ... and moving at speed = ", timelineClock.speed

client.disconnect()
```

The client runs in a separate thread managed by the websocket client library, so the *onXXX* methods are called while the main thread sleeps.

### Using TSClientConnection

This class provides the lowest level interface to a CSS-TS server. It only implements the parsing of incoming Control Timestamp messages and sending of outgoing Actual, Earliest and Latest Presentation Timestamp messages. It does not try to understand the meaning of the messages sent or received.

You can use the class either by subclassing and overriding the various stub methods or by creating an instance and replacing the stub methods with your own function handlers dynamically.

Pass the WebSocket URL of the CSS-TS server during initialisation then call the *connect()* and *disconnect()*

methods to connect and disconnect from the server. The *onXXX()* methods are called when connection or disconnection takes place, if there is a protocol error (e.g. a message was received that could not be parsed as Control Timestamp) or a new Control Timestamp is received.

A simple example:

```
from dvbcss.protocol.client.ts import TSClientConnection
from dvbcss.protocol.ts import AptEptLpt
from dvbcss.protocol import OMIT

class MyClientConnection(TSClientConnection):
    def onConnected(self):
        print "Connected!"
        e = Timestamp(0, float("-inf"))
        l = Timestamp(0, float("+inf"))
        aptEptLpt = AptEptLpt(earliest=e, latest=l, actual=OMIT)
        client.sendTimestamp(aptEptLpt)

    def onDisconnected(self, code, reason):
        print "Disconnected :-("

    def onControlTimestamp(self, ct):
        print "Received a ControlTimestamp: "+str(ct)

# make connection. Want PTS timeline for any DVB service.
client = MyTSClientConnection("ws://127.0.0.1/ts", "dvb://", "urn:dvb:css:timeline:pts
↪")
client.connect()

time.sleep(60)      # run only for 60 seconds then disconnect

client.disconnect()
```

## Classes

### TSClientConnection

**class** dvbcss.protocol.client.ts.**TSClientConnection**(*url*, *contentIdStem*, *timelineSelector*)

Simple object for connecting to a CSS-TS server (an MSAS) and handling the connection.

Use by subclassing and overriding the following methods or assigning your own functions to them at runtime:

- *onConnected()*
- *onDisconnected()*
- *onControlTimestamp()*
- *onProtocolError()*

If you do not wish to subclass, you can instead create an instance of this class and replace the methods listed above with your own functions dynamically.

Use the *sendTimestamp()* method to send Actual, Earliest and Latest Presentation Timestamps to the server.

This class has the following properties:

- *connected* (read only) whether the client is connected or not

**Initialisation takes the following parameters:**

**Parameters**

- **url** (*str*) – The WebSocket URL of the TS Server to connect to. E.g. "ws://127.0.0.1/mysystem/ts"
- **contentIdStem** (*str*) – The stem of the content id to be included in the SetupData message that is sent as soon as the connection is opened.
- **timelineSelector** (*str*) – The timeline selector to be included in the SetupData message that is sent as soon as the connection is opened.

**connect**()
Open the connection.

**Throws ConnectionError** if there was a problem and the connection could not be opened.

**connected**
This property is True if the connection is connect, otherwise False

**disconnect**(*code=1001*, *reason=''*)
Close the connection.

**Parameters**

- **code** – (optional int) The connection closure code to be sent in the WebSocket disconnect frame
- **reason** – (optional str) The human readable reason for the closure

**onConnected**()
This method is called when the connection is opened and the setup-data message has been sent.

*This is a stub for this method. Sub-classes should implement it.*

**onControlTimestamp**(*controlTimestamp*)
This method is called when a Control Timestamp message is received from the server.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **controlTimestamp** – A *ControlTimestamp* object representing the received message.

**onDisconnected**()
This method is called when the connection is closed.

*This is a stub for this method. Sub-classes should implement it.*

**onProtocolError**(*msg*)
This method is called when there has been an error in the use of the TS protocol - e.g. receiving the wrong kind of message.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **msg** – A str description of the problem.

**sendTimestamp**(*aptEptLpt*)
Send an Actual, Earliest and Latest Presentation Timestamp message to the TS Server.

**Parameters** **aptEptLpt** – The *AptEptLpt* object representing the Actual, Earliest and Latest Presentation Timestamp to be sent.

### TSClientClockController

**class** dvbcss.protocol.client.ts.**TSClientClockController**(*tsUrl*, *contentIdStem*, *timelineSelector*, *timelineClock*, *correlationChangeThresholdSecs=0.0001*, *earliestClock=None*, *latestClock=None*)

This class manages a CSS-TS protocol connection and controls a *CorrelatedClock* to synchronise it to the timeline provided by the server.

Subclass and override the following methods when using this class:

- *onConnected()*
- *onDisconnected()*
- *onTimingChange()*
- *onTimelineAvailable()*
- *onTimelineUnavailable()*
- *onProtocolError()*

If you do not wish to subclass, you can instead create an instance of this class and replace the methods listed above with your own functions dynamically.

Create an instance of this class and use the *connect()* and *disconnect()* methods to start and stop its connection to the server. The *contentIdStem* and *timelineSelector* you specify are sent to the server in a *SetupData* message to choose the timeline to receive from the server.

While connected, while the timeline is available, the timelineClock you provided will have its *correlation* updated to keep it in sync with the timeline received from the server.

The *speed* property of the timeline clock will also be adjusted to match the timeline speed indicated by the server. For example: it will be set to zero when the timeline is paused, or 2.0 when the timeline speed is x2. The *tickRate* property of the clock is not changed.

Requirements for the timeline clock you provide:

- The *tickRate* of the **timeline clock** must match that of the timeline.
- Its parent must represent the **wall clock**
- The **wall clock** must have a *~dvbcss.clock.CorrelatedClock.tickRate* that matches the wall clock tick rate.

The TSClientClockController has the following properties:

- *connected* (read only) is the client connected?
- *timelineAvailable* (read only) is the timeline available?
- *latestCt* (read only) is the most recently received *ControlTimestamp* message
- *earliestClock* (read/write) A clock object representing earliest possible presentation timing, or `None`
- *latestClock* (read/write) A clock object representing latest possible presentation timing, or `None`

**Initialisation takes the following parameters:**

**Parameters**

- **url** (*str*) – The WebSocket URL of the TS Server to connect to. E.g. "ws://127.0.0.1/mysystem/ts"

- **contentIdStem** (*str*) – The stem of the content id to be included in the SetupData message that is sent as soon as the conncetion is opened.
- **timelineSelector** (*str*) – The timeline selector to be included in the SetupData message that is sent as soon as the conncetion is opened.
- **timelineClock** (*CorrelatedClock*) – A clock object whose parent must represent the wall clock.
- **correlationChangeThresholdSecs** (*float*) – Minimum threshold for the change in the timeline (in units of seconds) that will result in the timeline clock being adjusted.
- **earliestClock** (*CorrelatedClock* or None) – An optional clock object representing the earliest possible presentation timing that this client can achieve (expressed on the same timeline)
- **latestClock** (*CorrelatedClock* or None) – An optional clock object representing the latest possible presentation timing that this client can achieve (expressed on the same timeline)

**connected**
(bool) True if currently connected to the server, otherwise False.

**latestCt**
(*ControlTimestamp*) A copy of the most recently received Control Timestamp.

**earliestClock**
None or a *CorrelatedClock* correlated to the WallClock representing the earliest possible presentation timing.

**latestClock**
None or a *CorrelatedClock* correlated to the WallClock representing the latest possible presentation timing.

**connect**()
Start the client by trying to open the connection.

If the connection opens successfully, a *SetupData* message will be sent automatically.

**Throws ConnectionError** if there was a problem and the connection could not be opened.

**disconnect**()
Disconnect from the server.

**getStatusSummary**()

**Returns str** A human readable string describing the state of the timeline and the connection.

**onConnected**()
This method is called when the connection is opened and the setup-data message has been sent.

*This is a stub for this method. Sub-classes should implement it.*

**onDisconnected**()
This method is called when the connection is closed.

*This is a stub for this method. Sub-classes should implement it.*

**onProtocolError**(*msg*)
This method is called when there has been an error in the use of the CII protocol - e.g. receiving the wrong kind of message.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **msg** – A str description of the problem.

**onTimelineAvailable**()
This method is called when the server indicates that the timeline is available.

*This is a stub for this method. Sub-classes should implement it.*

**onTimelineUnavailable**()
This method is called when the server indicates that the timeline is unavailable.

*This is a stub for this method. Sub-classes should implement it.*

**onTimingChange**(*speedChanged*)
This method is called when the server indicates that the timeline timing has changed.

This means that a received Control Timestamp has changed the timing of the clock relative to the wall clock by the threshold amount or more, or that the speed of the timeline has changed. (as indicated by the timelineSpeedMultiplier property of received Control Timestamps).

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **speedChanged** – (bool) True if the speed of the timeline has changed, otherwise False.

**sendAptEptLpt**(*includeApt=True*)
Sends an Actual, Earliest and Latest presentation timestamp to the CSS-TS server.

- The EPT is derived from the *earliestClock* property, if it is not None and it is a clock that is available.
- The LPT is derived from the *latestClock* property, if it is not None and it is a clock that is available.
- The APT is only included if the includeApt argument is True (default=True) and it is a clock that is available.

**Parameters** **includeApt** – (bool) Set to False if the Actual Presentation Timestamp is *not* to be included in the message (default=True)

**timelineAvailable**
(bool) True if the most recently received Control Timestamp indicates that the timeline is available.

Changed in version 0.4: It is now recommended to not use this method. Instead, use the *isAvailable()* method of a *clock* instead.

### 2.2.4 CSS-TS Servers

Module: *dvbcss.protocol.server.ts*

The `TSServer` class implements a CSS-TS server that can be plugged into the cherrypy web server engine.

To create a CSS-TS Server, first create and mount the server in a cherrypy web server. Then you can start the cherrypy server and the CSS-TS server will start to accept connections from clients.

The CSS-TS Server needs a *timeline source* to be given to it in order for it to serve these to clients. A timeline source provides the Control Timestamps given a particular timeline selector. A CSS-TS server can have multiple timelines sources plugged into it, and they can be added or removed dynamically while the server is running.

An *example* server is provided in this package.

### Using TSServer and Timeline Sources

#### 1. Imports and initialisation

To run a CSS-TS server, you must import both ws4py's cherrypy server and the *dvbcss.protocol.server.ts* module. When the *dvbcss.protocol.server.ts* module is imported, it will register as a "tool" with cherrypy, so it must be imported after cherrypy is imported.

Next, subscribe the ws4py websocket plugin to cherrypy.

```
import cherrypy
from ws4py.server.cherrypyserver import WebSocketPlugin
from dvbcss.protocol.server.ts import TSServer

# initialise the ws4py websocket plugin
WebSocketPlugin(cherrypy.engine).subscribe()
```

#### 2. Create and mount the TS server

You can now create an instance of a TSServer and mount it into the cherrypy server at a path of your choosing.

The configuration for that path (see example code below) must turn on the "dvb_ts" tool and pass a "handler_cls" argument whose value is the handler class that the TS Server instance provides via the `TSServer.handler` attribute.

The TS Server needs, at initialisation, to be told the initial contentId that it is providing timelines for and it also needs to be provided with a `clock` object representing the Wall Clock (ticking at the correct rate of 1e9 ticks per second). It

needs the WallClock so it can set the WallClockTime property in Control Timestamps that notify clients of a timeline being unavailable.

For example, to create a TS Server mounted at the URL path "/ts":

```
# create a Wall Clock
from dvbcss.clock import SysClock, CorrelatedClock
sysClock = SysClock()
wallClock = CorrelatedClock(parentClock=sysClock, tickRate=1000000000)

# create TS Server
tsServer = TSServer(contentId="dvb://1004", wallClock=wallClock,␣
↪maxConnectionsAllowed=10)

# bind it to the URL path /ts in the cherrypy server
class Root(object):
    @cherrypy.expose
    def ts(self):
        pass

# construct the configuration for this path, providing the handler and turning on the␣
↪tool hook
cfg = {"/ts": {'tools.dvb_ts.on': True,
               'tools.dvb_ts.handler_cls': tsServer.handler
              }
      }

cherrypy.tree.mount(Root(), "/", config=cfg)
```

### 3. Start cherrypy running

Start cherrypy running and our TS server will start to accept connections from clients:

```
# configure cherrypy to serve on port 7681
cherrypy.config.update({"server.socket_port":7682})

# activate cherrypy web server (non blocking)
cherrypy.engine.start()
```

The cherrypy engine runs in a background thread when the cherrypy engine is started. Callbacks from the *TSServer* happen on the ws4py websocket library's thread(s) for handling received messages.

### 4. Providing timelines through the TS Server

To make a timeline available to clients, we create a source for that timeline. For example, a *SimpleClockTimelineSource* based on a *clock* object that corresponds to the ticking of progress on the timeline. We also decide the timeline selector that clients must use to obtain that timeline.

For example, we might create a Timeline Source to represent ticking of a PTS timeline:

```
from dvbcss.protocol.server.ts import SimpleClockTimelineSource
from dvbcss.clock import Correlation

ptsClock = CorrelatedClock(parentClock=wallClock, tickRate=90000)
ptsTimelineSrc = SimpleClockTimelineSource(timelineSelector="urn:dvb:css:timeline:pts
↪", wallClock=wallClock, clock=ptsClock)
```

```
# set that ptsClock to start counting from zero starting NOW:
ptsClock.correlation = Correlation(wallClock.ticks, 0)
```

When we want that timeline to become available to connected clients we add it to the TS Server:

```
tsServer.attachTimelineSource(ptsTimelineSrc)
tsServer.updateAllClients()
```

When we make any change (adding or removing timlelines, or changing a timeline in any way) then we must call *TSServer.updateAllClients()* to ensure Control Timestamp messages are sent immediately to all connected clients to notify them of any changes.

Any new client connections or existing connections, that ask for that timeline (using the timeline selector and matching the content id) will have the timeline made available to them.

If at some point the relationship between PTS and wall clock changes, or the clock availability changes, then we must instruct the TS Server to send new Control Timestamps out if needed:

```
# lets reset the pts timeline position to zero again
ptsClock.correlation = Correlation(wallClock.ticks, 0)
tsServer.updateAllClients()
```

When we want to make the timeline unavailable, we can simply update the availability of the clock:

```
ptsClock.setAvailability(False)
tsServer.updateAllClients()
```

Or if we wish to stop more permanently, we can instead detatch the timeline source from the TS Server:

```
tsServer.removeTimelineSource(ptsTimelineSrc)
tsServer.updateAllClients()
```

If there are clients connected that have asked for that timeline, then the update call will cause Control Timestamps to be sent to them that indicate that the timeline is no longer available.

We can also change the contentId for which the timelines are being provided. If we do this, then again, it may change whether timelines are available to any currently connected clients:

#### 5. Changing content id

```
tsServer.contentId = "http://myservice.com/new-content-id"
tsServer.updateAllClients()
```

#### What does TSServer do for you and what does it not?

*TSServer* handles the connection and disconnection of clients without requiring any further intervention. It handles the initial setup-data message and uses the information in that to determine which *TimelineSource* it should obtain Control Timestamps from. It will also automatically ensure 'null' Control Timestamp messages are sent if there is no suitable timeline source.

Your code (or a *TimelineSource* that you create) must call the `updateClient()` or `updateAllClients()` to notify the TSServer that it needs to potentialy send updated Control Timestamps to clients.

### More about Timeline Sources

A timeline source is any object that implements the methods defined by the *TimelineSource* base class.

### Exising timeline source implementations

Two implementations are provided:

- *SimpleTimelineSource* is a timeline source where you directly specify the Control Timestamp that will be sent to clients.
- *SimpleClockTimelineSource* is a timeline source where Control Timestamps are generated from *Clock* objects representing the position of the timeline and the Wall Clock.

The CSS-TS server does *not* automatically push out new Control Timestamps to connected clients. It will only do so when the `updateAllClients()` or `updateClient()` methods are called. This allows you to do things like swap out and replace a timeline source object without causing spurious Control Timestamps to be sent.

### Creating new Timeline Sources

Create new Timeline Source types by subclasing *TimelineSource* and, at minimum, implementing the stubs *recognisesTimelineSelector()* and *getControlTimestamp()*.

For example, here is a timeline source that recognises a timeline selector "urn:pretend-timeline:N" where N is the ticks per second of the timeline. The progress of this timeline is controlled by the code outside of this object periodcally calling the `setTimelinePositionNow` method. It needs a *clock* object representing the WallClock to be passed to it. It can serve different tick rates to different clients.

```
from dvbcss.protocol.server.ts import TimelineSource
from dvbcss.protocol.ts import ControlTimestamp, Timestamp
from dvbcss.clock import Correlation
import re

class PretendTimelineSource(TimelineSource):
    def __init__(self, wallClock):
        super(PretendTimelineSource,self).__init__()
        self.wallClock = wallClock
        self.correlation = Correlation(0,0)

    def recognisesTimelineSelector(self, timelineSelector):
        return re.match("^urn:pretend-timeline:([0-9]+)$", timelineSelector)

    def getControlTimestamp(self, timelineSelector):
        match = re.match("^urn:pretend-timeline:([0-9]+)$", timelineSelector)
        if not match:
            raise RuntimeError("This should never happen.")
        elif self.correlation is None:
            # timeline not yet available, so return 'null' control timestamp
            return ControlTimestamp(Timestamp(None, self.wallClock.ticks), None)
        else:
            tickRate = int(match.group(1))
            contentTime = tickRate * self.correlation.childTicks
            wallClockTime = self.correlation.parentTicks
            speed = 1
            return ControlTimestamp(Timestamp(contentTime, wallClockTime), speed)
```

```
    def setTimelinePositionNow(self, timelineSecondsNow):
        self.correlation = Correlation(self.wallClock.nanos, timelineSecondsNow)
```

The base class also has stub methods to support notification of when a sink is attached to the timeline source and also methods to notify of when a particular timeline selector is being requested by at least one client and when it is no longer required by any clients. See the documentation for *TimelineSource* for more details.

### Classes

### TSServer

**class** dvbcss.protocol.server.ts.**TSServer**(*contentId*, *wallClock*, *maxConnectionsAllowed=-1*, *enabled=True*)

Implements a server for the CSS-TS protocol that serves timelines provided by sources that are plugged into this server object.

Use by instantiating. You may subclass if you wish to optionally override the following methods:

- *onClientConnect()*
- *onClientDisconnect()*
- *onClientMessage()*
- *onClientSetup()*

When the server is "disabled" it will refuse attempts to connect by sending the HTTP status response 403 "Forbidden".

When the server has reached its connection limit, it will refuse attempts to connect by sending the HTTP status response 503 "Service unavailable".

This object has the following properties:

- *enabled* (read/write) controls whether the server is enabled
- *contentId* (read/write) controls the contentId that timelines are being served for
- *handler* (read) the class that must be passed as part of the cherrypy configuration for the URL path this server is attached to.

The *contentId* and *enabled* attributes can be changed at runtime. Whether the contentId matches that provided by a client when it initially connects is part of determining whether the timeline is available to the client.

Use *attachTimelineSource()* and *removeTimelineSource()* to add and remove sources of timelines. Adding or removing a source of a timeline will affect availablilty of a timeline to a client.

This server does not automatically send *Control Timestamp* messages to clients. After you make a change (e.g. to the *contentId* or a state change in a timeline source or attaching or removing a timeline source) then you must call updateAllClients() to cause messages to be sent.

The only exception to this is changes to the *enabled* state which takes effect immediately.

**Initialisation takes the following parameters:**

**Parameters**

- **contentId** (*str*) – The content-id for which timelines will be made available.
- **wallClock** (*clock*) – The wall clock

- **maxConnectionsAllowed** (*int*) – (int, default=-1) Maximum number of concurrent connections to be allowed, or -1 to allow as many connections as resources allow.
- **enabled** (*bool*) – Whether this server starts off enabled or disabled.

**contentId**
(read/write str) The content ID for all timelines currently being served. Can be changed at runtime.

**handler**
Handler class for new connections.

When mounting the CII server with cherrypy, include in the config dict a key 'tools.dvb_cii.handler_cls' with this handler class as the value.

**attachTimelineSource**(*timelineSource*)
Attach (add) a source of a timeline to this CSS-TS server. This causes the timeline to become available immediately to any connected clients that might be requesting it.

This causes the `addSink()` method of the timeline source to be called, to notify it that this CSS-TS server is now a recipient (sink) for this timeline.

**Parameters** **timelineSource** – Any object implementing the methods of the *TimelineSource* base class

**enabled**
(read/write bool) Whether this server endpoint is enabled (True) or disabled (False).

Set this property enable or disable the endpoint.

When disabled, existing connections are closed with WebSocket closure reason code 1001 and new attempts to connect will be refused with HTTP response status code 403 "Forbidden".

**getConnections**()

**Returns dict** mapping a WebSocket object to connection related data for all connections to this server. This is a snapshot of the connections at the moment the call is made. The dictionary is not updated later if new clients connect or existing ones disconnect.

**getDefaultConnectionData**()
Internal method. Creates empty 'connection data' for the connection consisting of: a dict: `{ "setup":None, "prevCt":None, "aptEptLpt":None }`

This reflects that no setup-data has been received yet, and no Control Timestamps have been sent, and no AptEptLpt has been received either

**onClientAptEptLpt**(*webSock*, *apteptlpt*)
Called when a client has sent an updated AptEptLpt message

*This is a stub for this method. Sub-classes should implement it.*

**Parameters**

- **webSock** – The connection from the client.
- **aptEptLpt** – (AptEptLpt) object representing the received timestamp message.

**onClientConnect**(*webSock*)
Called when a client establishes a connection.

If you override, make sure you call the superclass implementation of this method.

**Parameters** **webSock** – The connection from the client.

**onClientDisconnect**(*webSock*, *connectionData*)
Called when a client disconnects.

If you override, make sure you call the superclass implementation of this method.

**Parameters**

- **webSock** – The connection from the client.
- **connectionData** – A `dict` containing data relating to this (now closed) connection

**onClientMessage**(*webSock*, *message*)
Called when a message is received from a client.

If you override, make sure you call the superclass implementation of this method.

**Parameters**

- **webSock** – The connection from the client.
- **msg** – (`Message`) WebSocket message that has been received. Will be either a `Text` or a `Binary` message.

**onClientSetup**(*webSock*)
Called when a client has connected and submitted its SetupData message to provide context for the connection.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **webSock** – The connection from the client.

**removeTimelineSource**(*timelineSource*)
Remove a source of a timeline from this CSS-TS server. This causes the timeline to become unavailable immediately to any connected clients that are using it.

This causes the `removeSink()` method of the timeline source to be called, to notify it that this CSS-TS server is no longer a customer (sink) for this timeline.

**Parameters** **timelineSource** – Any object implementing the methods of the *TimelineSource* base class

**updateAllClients**()
Causes an update to be sent to all clients that need it (i.e. if the ControlTimestamp that would be sent now is different to the one most recently sent to that client)

**updateClient**(*webSock*)
Causes an updated `ControlTimestamp` to be sent to the WebSocket connection specified.

The ControlTimestamp is only sent if it is different to the last time this was done for this connection.

The value of the Control Timestamp is determined by searching all attached timeline sources to find one that can supply a Control Timestamp for this connection.

### TimelineSource

**class** dvbcss.protocol.server.ts.**TimelineSource**
Base class for timeline sources.

Subclass and implement the stub methods to create Timeline Source:

- *recognisesTimelineSelector()*
- *getControlTimestamp()*

If your source needs to be informed of when a timeline is needed and when it becomes no longer needed (e.g. so you can allocate/deallocate resources needed to extract it) then also implement these stub methods:

- *timelineSelectorNeeded()*
- *timelineSelectorNotNeeded()*

You can also optionally override the following methods, provided your code still calls through to the base class implementations:

- *attachSink()* (see note)
- *removeSink()* (see note)

The *attachSink()* and *removeSink()* methods will be called by parties interested in the TimelineSource (such as a *TSServer*) when they wish to use the timeline source. The base class implementations of these methods maintain the `sinks` attribute as a dictionary indexed by sink.

**Note:** When subclassing *attachSink()* and *removeSink()* remember to call the base class implementations of these methods.

**attachSink**(*sink*)
Called to notify this Timeline Source that there is a sink (such as a *TSServer*) that wishes to use this timeline source.

**Parameters** **sink** – The sink object (e.g. a *TSServer*)

A TimelineSource implementation can use knowledge of what sinks are attached in whatever way it wishes. For example: it might use this to proactively call `updateAllClients()` on the attached TSServer when its Timeline's relationship to wall clock changes in some way. It is up to the individual implementation whether it chooses to do this or not.

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

**getControlTimestamp**(*timelineSelector*)
Get the Control Timestamp from this Timeline Source given the supplied timeline selector.

This method will only be called with timelineSelectors for which the *recognisesTimelineSelector()* method returned True.

*This is a stub for this method. Sub-classes should implement it.*

The return value should be a *ControlTimestamp* object. If the timeline is known to be unavailable, then the contentTime and speed properties of that Control Timestamp must be None.

If, however, you want the TS Server to not send any Control Timestamp to clients at all, then return None instead of a Control Timestamp object. Use this when, for example, you are still awaiting an result from code that has only just started to try to extract the timeline and doesn't yet know if there is one available or not.

**Parameters** **timelineSelector** – (`str`) A timeline selector supplied by a CSS-TS client

**Returns** A *ControlTimestamp* object for this timeline source appropriate to the timeline selector, or None if the timleine is recognised no Control Timestamp should yet be provided.

**recognisesTimelineSelector**(*timelineSelector*)
*This is a stub for this method. Sub-classes should implement it.*

**Parameters** **timelineSelector** – (`str`) A timeline selector supplied by a CSS-TS client

**Returns** True if this Timeline Source can provide a Control Timestamp given the specified timeline selector

**removeSink**(*sink*)
Called to notify this Timeline Source that there is a sink that no longer wishes to use this timeline source.

**Parameters** **sink** – The sink object

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

**timelineSelectorNeeded**(*timelineSelector*)
Called to notify this Timeline Source that there is a need to provide a timeline for the specified timeline selector.

**Parameters** **timelineSelector** – (str) A timeline selector supplied by a CSS-TS client that has not been specified by any other currently connected clients.

This is useful to, for example, initiate processes needed to extract the timeline for the specified timeline selector. You will not receive repeats of the same notification.

If the timeline is no longer needed then the *timelineSelectorNotNeeded()* function will be called to notify of this. After this, then you might be notified again in future if a timeline for the timeline selector becomes needed again.

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

**timelineSelectorNotNeeded**(*timelineSelector*)
Called to notify this Timeline Source that there is no longer a need to provide a timeline for the specified timeline selector.

**Parameters** **timelineSelector** – (str) A timeline selector that was previously needed by one or more CSS-TS client(s) but which is no longer needed by any.

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

### SimpleTimelineSource

**class** dvbcss.protocol.server.ts.**SimpleTimelineSource**(*timelineSelector*, *controlTimestamp*, **args*, ***kwargs*)
A simple Timeline Source implementation for a fixed timeline selector and where the Control Timestamp is specified manually.

**Initialisation takes the following parameters:**

**Parameters**

- **timelineSelector** – (str) The exact timeline selector that this Timeline Source will provide Control Timestamps for.
- **controlTimestamp** – (*ControlTimestamp*) The initial value of the Control Timestamp

**attachSink**(*sink*)
Called to notify this Timeline Source that there is a sink (such as a *TSServer*) that wishes to use this timeline source.

**Parameters** **sink** – The sink object (e.g. a *TSServer*)

A TimelineSource implementation can use knowledge of what sinks are attached in whatever way it wishes. For example: it might use this to proactively call `updateAllClients()` on the attached TSServer when its Timeline's relationship to wall clock changes in some way. It is up to the individual implementation whether it chooses to do this or not.

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

**removeSink**(*sink*)
Called to notify this Timeline Source that there is a sink that no longer wishes to use this timeline source.

**Parameters** **sink** – The sink object

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

**timelineSelectorNeeded**(*timelineSelector*)

Called to notify this Timeline Source that there is a need to provide a timeline for the specified timeline selector.

**Parameters** **timelineSelector** – (str) A timeline selector supplied by a CSS-TS client that has not been specified by any other currently connected clients.

This is useful to, for example, initiate processes needed to extract the timeline for the specified timeline selector. You will not receive repeats of the same notification.

If the timeline is no longer needed then the *timelineSelectorNotNeeded()* function will be called to notify of this. After this, then you might be notified again in future if a timeline for the timeline selector becomes needed again.

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

**timelineSelectorNotNeeded**(*timelineSelector*)

Called to notify this Timeline Source that there is no longer a need to provide a timeline for the specified timeline selector.

**Parameters** **timelineSelector** – (str) A timeline selector that was previously needed by one or more CSS-TS client(s) but which is no longer needed by any.

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

### SimpleClockTimelineSource

**class** dvbcss.protocol.server.ts.**SimpleClockTimelineSource**(*timelineSelector*, *wallClock*, *clock*, *speedSource=None*, *autoUpdateClients=False*)

Simple subclass of *TimelineSource* that is based on a *clock* object.

The Control Timestamp returned by the `getControlTimestamp()` method reflects the state of the clock.

Note that this does **not** result in a new Control Timestamp being pushed to clients, unless you set the auto update parameter to True when initialising this object. By default (with no auto-updating), you still have to call *updateAllClients()* manually yourself to cause that to happen.

Use auto-updating with caution: if you have multiple Timeline Sources driven by a common clock, then a change to that clock will cause each Timeline Source to call *updateAllClients()*, resulting in multiple unnecessary calls.

The tick rate is fixed to that of the supplied clock and timeline selectors are only matched as an exact match.

The speed property of the clock will be used as the timelineSpeedMultiplier value, unless a different clock is provided as the optional speedSource argument at initialisation. This is useful if the speed of the timeline is set by setting the speed property of a parent clock, and not the speed property of this clock (e.g. in situations where a single clock represents timeline progress but there are multiple clocks as children of that to represent the timeline on different scales - e.g. PTS, TEMI etc).

The availability of the clock is mapped to whether the timeline is available.

SimpleClockTimelineSource generates its correlation by observing the current tick value of the wallClock and the provided clock whenever a ControlTimestamp needs to be provided.

**Initialisation takes the following parameters:**

**Parameters**

- **timelineSelector** – (str) The timeline selector that this TimelineSource will provide Control Timestamps for

- **wallClock** – (*ClockBase*) A clock object representing the Wall Clock
- **clock** – (*ClockBase*) A clock object representing the flow of ticks of the timeline.
- **speedSource** – (None or *ClockBase*) A different clock object from which the timelineSpeedMultiplier is determined (from the clock's speed property), or None if the clock for this is not different.
- **autoUpdateClients** – (bool) Automatically call updateAllClients() if there is a change in the clock or wallClock

**notify**(*cause*)

Called by clocks to notify of changes (because this class binds itself to the clock object).

If auto-updating is enabled then this will result in a call to `updateAllClients()` on all sinks.

**timelineSelectorNeeded**(*timelineSelector*)

Called to notify this Timeline Source that there is a need to provide a timeline for the specified timeline selector.

**Parameters** **timelineSelector** – (str) A timeline selector supplied by a CSS-TS client that has not been specified by any other currently connected clients.

This is useful to, for example, initiate processes needed to extract the timeline for the specified timeline selector. You will not receive repeats of the same notification.

If the timeline is no longer needed then the *timelineSelectorNotNeeded()* function will be called to notify of this. After this, then you might be notified again in future if a timeline for the timeline selector becomes needed again.

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

**timelineSelectorNotNeeded**(*timelineSelector*)

Called to notify this Timeline Source that there is no longer a need to provide a timeline for the specified timeline selector.

**Parameters** **timelineSelector** – (str) A timeline selector that was previously needed by one or more CSS-TS client(s) but which is no longer needed by any.

NOTE: If you override this in your subclass, ensure you still call this implementation in the base class.

## Functions

### ciMatchesStem

dvbcss.protocol.server.ts.**ciMatchesStem**(*ci*, *stem*)

Checks if a content identifier stem matches a content identifier. A match is when the content identifier starts with the stem and is the same length as the stem, or longer.

**Parameters**

- **ci** (str or *OMIT*) – Content identifier
- **stem** (*str*) – Content identifier stem

**Returns** True if the supplied content identifier stem (stem) matches the content identifier. If the ci is *OMIT* then always returns False.

#### isControlTimestampChanged

dvbcss.protocol.server.ts.**isControlTimestampChanged**(*prev*, *latest*)

Checks whether a new (latest) Control Timestamp is different when compared to a old (previous) Control Timestamp.

Note that this does not check equivalency (if two Control Timestamps represent the same mapping between Wall clock time and content time) but instead checks if the property values comprising the two timestamps are exact matches.

**Parameters**

- **prev** – None or a previous *ControlTimestamp*
- **latest** – A new *ControlTimestamp*

**Returns** True if the previous Control Timestamp is supplied is None, or if any of its properties differ in value to that of the latest Control Timestamp

**Throws ValueError** The new Control Timestamp supplied is None instead of a Control Timetamp

This package provides objects for representing messages exchanged via the DVB CSS-TS protocol and for implementing clients and servers.

The TS protocol is a mechanism for a server to share timeline position and playback speed with a client. In effect it enables a client to synchronise its understanding of the progress of media presentation with that of a server, in terms of a particular timeline.

The client initially sends a *SetupData* message to specify what timeline it wants to synchronise in terms of. The server then periodically sends *ControlTimestamp* messages to inform the client of the state of presentation timing. The client can also send AptEptLpt (Actual, Earliest and Latest Presentation Timestamp) messages to the server to inform it of its playback timing and range of playback timings it can achieve.

The client implementation in this library can control a *CorrelatedClock*, synchronising it to the timeline. The server implementation in this library uses *CorrelatedClock* objects as its source of timelines that it is to share with clients.

Modules for using the CSS-TS protocol:

- *dvbcss.protocol.ts* : objects for representing and packing/unpacking the CSS-TS protocol messages.
- *dvbcss.protocol.client.ts* : implementation of a client for a CSS-TS connection.
- *dvbcss.protocol.server.ts* : implementation of a server for a CSS-TS connection.

## 2.3 CSS-WC protocol

### 2.3.1 CSS-WC Protocol Introduction

**Here is a quick introduction to the CSS-WC protocol. For full details, refer to the** DVB specification ETSI 103 286 part 2.

The CSS-WC protocol is for establishing *Wall clock synchronisation* - meaning that there is a common synchronised sense of time (a "wall clock") between the server (e.g. TV) and client (e.g. companion). This common wall clock is used in the CSS-TS protocol to make it immune to network delays.

The client uses the information carried in the protocol to estimate the server wall clock and attempt to compensate for network latency. This is a connectionless UDP protocol similar to NTP's client-server mode of operation, but much simplified and not intended to set the system real-time clock.

### Sequence of interaction

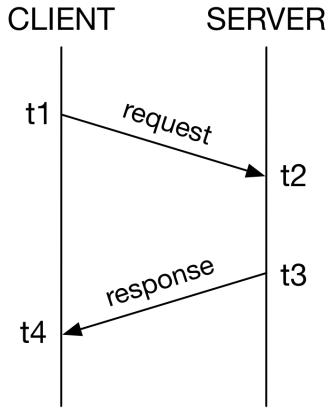
The client is assumed to already know the host and port number of the CSS-WC server (usually from the information received via the *CSS-CII protocol*).

1. The client sends a Wall Clock protocol "request" message to the server.
2. The server sends back a Wall Clock protocol "response" message to the client.
3. If the server is able to more accurately measure when it sent a message *after* it has done so, then it can optionally send a "follow-up response" with this information.

The client repeats this process as often as it needs to.

### Synchronising the wall clock

The client notes the time at which the request is sent and the response received, and by the server including the times at which it received the request and sent its response. Using this information the client can estimate the relationship between the time of its clock and that of the server. It can also calculate an error bound on this (known as dispersion):




**Relationship expressed as an estimated offset:**

- Offset between local clock and server wall clock is (( *t3* + *t2* ) - ( *t4* + *t1* )) / 2

**Relationship expressed as a correlation:**

- When local clock is ( *t1* + *t4* )/2
- . . . the server wall clock is estimated to be ( *t2* + *t3* )/2

The DVB specification (part 2) contains an annex that goes into more detail on the theory of how to calculate dispersion and how a client can use this as part of a simple algorithm to align its wall clock.

### Message format

Requests and responses both have the same fixed 32 byte binary message format. A *WCMessage* carries the following fields:

- Protocol **version** identifier
- Message **type** (request / response / response-before-follow-up / follow-up)
- The **precision** of the server's wall clock
- The **maximum frequency error** of the server's wall clock
- Timevalues (in NTP 64-bit time format, comprising a 32bit word carrying the number of nanoseconds and another 32bit word containing the number of seconds)
  - **Originate timevalue:** when the client sent the request.
  - **Receive timevalue:** when the server received the request.
  - **Transmit timevalue:** when the server sent the response.

The *precision*, *max freq error*, *receive timevalue* and *transmit timevalue* fields only have meaning in a response from a server. Their values do not matter in requests.

## 2.3.2 CSS-WC Message objects

Module: *dvbcss.protocol.wc*



The *WCMessage* class represents a CSS-WC protocol message.

The *Candidate* class represents the measurement resulting from a request-response exchange of messages with a server. This is a "candidate" that could be used to update the client's estimate of the Wall Clock and is therefore used by a Wall Clock Client algorithm. A candidate is calculated from a *WCMessage* that represents a Wall Clock protocol response message received from a server.

A candidate can calculate the correlation needed to set a *CorrelatedClock* to model the server's Wall Clock.

### Example usage

Creating a request message at a Wall Clock Client:

```
>>> from dvbcss.protocol.wc import WCMessage
>>> import time

>>> t1 = time.time() * 1000000000

>>> msg = WCMessage(msgtype=WCMessage.TYPE_REQUEST, precision=-10,␣
↪maxFreqError=256*50, originateNanos=t1, receiveNanos=0, transmitNanos=0)
>>> packedMessage = msg.pack()
```

```
>>> packedMessage
"\x00\x00\xf6\x00\x00\x002\x00TvH
↪'3\xf5\xfc\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
```

Processing a received response message at a Wall Clock Client:

```
>>> from dvbcss.protocol.wc import Candidate

>>> t4 = <nanoseconds-at-which-message-was-received>
>>> msg = WCMessage.unpack(receivedData)
>>> msg.msgtype
1
>>> c = Candidate(msg, t4)
>>> c.rtt
459734573
```

Creating a correlation to configure a :class:‘‘dvbcss.clock.CorrelatedClock‘ representing the client estimate of the server’s wall clock:

```
>>> corr = c.calcCorrelationFor(wallClock, localMaxFreqErrorPpm=wallClock.
↪getRootMaxFreqError())
>>> wallClock.correlation = corr
```

### Classes

### WCMessage

**class** dvbcss.protocol.wc.**WCMessage**(*msgtype*, *precision*, *maxFreqError*, *originateNanos*, *receiveNanos*, *transmitNanos*, *originalOriginate=None*)

Create object representing a CSS-WC wall clock request or response message.

**Initialisation takes the following parameters:**

**Parameters**

- **msgtype** (*int*) – Type of message. One of: *TYPE_REQUEST*, *TYPE_RESPONSE*, *TYPE_RESPONSE_WITH_FOLLOWUP* and *TYPE_FOLLOWUP*
- **precision** (*int*) – Precision (of the server’s wall clock) encoded in log base 2 seconds between -128 and +127 inclusive.
- **maxFreqError** (*int*) – Maximum frequency error (of the server’s wall clock) in units of 1/256ths ppm.
- **originateNanos** (*int*) – Originate timevalue in integer number of nanoseconds
- **receiveNanos** (*int*) – Receive timevalue in integer number of nanoseconds
- **transmitNanos** (*int*) – Transmit timevalue in integer number of nanoseconds
- **originalOriginate** (None or (int, int)) – Optional original encoding of the originate timevalue as (seconds, nanos). Overrides *originateNanos* if not *None*.

The originalOriginate parameter, if not None, overrides the originateNanos parameter.

Convert to and from a string containing the binary encoding of this message using the *pack()* method and *unpack()* class method.

**msgtype**
(read/write int) Type of message. 0=request, 1=response, 2=response-with-followup, 3=followup

**precision**
(read/write int) Precision encoded in log base 2 seconds between -128 and +127 inclusive. For example: -10 encodes a precision value of roughly 0.001 seconds.

**maxFreqError**
(read/write int) Maximum frequency error in units of 1/256ths ppm. For example: 12800 encodes a max freq error of 50ppm.

**originateNanos**
(read/write int) Originate timevalue in integer number of nanoseconds

**receiveNanos**
(read/write int) Receive timevalue in integer number of nanosecond

**transmitNanos**
(read/write int) Transmit timevalue in integer number of nanosecond

**originalOriginate**
(read/write None or (int, int)) Optional original encoding of the originate timevalue as (seconds, nanos). Overrides *originateNanos* when the message is packed if the value is not *None*.

**TYPE_FOLLOWUP = 3**
Constant: Message type 3 "follow-up response"

**TYPE_REQUEST = 0**
Constant: Message type 0 "request"

**TYPE_RESPONSE = 1**
Constant: Message type 1 "response with no follow-up"

**TYPE_RESPONSE_WITH_FOLLOWUP = 2**
Constant: Message type 2 "response to be followed by a follow-up response"

**copy**()
Duplicate this wallclock message object

**classmethod encodePrecision**(*precisionSecs*)
Convert a precision value in seconds to the format used in these messages

**getMaxFreqError**()
Get frequency error in ppm

**getPrecision**()
Get precision value in fractions of a second

**pack**()
Pack wall clock message into binary representation.
**Returns** String containing the wall clock message in final bitstream form.

**setMaxFreqError**(*maxFreqErrorPpm*)
Set freq error given a freq error represented as ppm

**setPrecision**(*precisionSecs*)
Set precision value given a precision represented as factions of a second

**classmethod unpack**(*data*)
Class method that takes a string containing a wall clock message and unpacks it to a *WCMessage* object.

**Parameters** **data** (*str*) – String containing binary representation of a Wall Clock message as received from a client or server.
**Returns** *WCMessage* object representing the wall clock message.

### Candidate

**class** dvbcss.protocol.wc.**Candidate**(*msg*, *nanosRx*)

This object represents a measurement "candidate" to be fed into a Wall Clock Client's algorithm. It is calculated from a *WCMessage* received as a response from a Wall Clock server.

This object requires that response comes from a Wall Clock Client that measured the *parent* of the clock that will be used to model the wall clock locally.

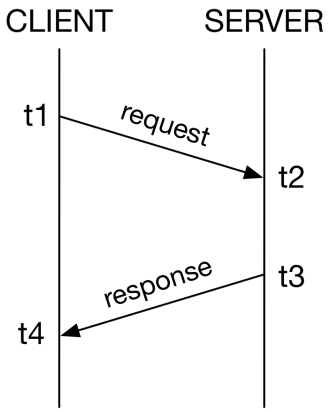
**Initialisation takes the following parameters:**

**Parameters**

- **msg** (*WCMessage*) – Response message received from server
- **nanosRx** (*int*) – the time, in nanoseconds, at which it was received (from the server)

Pass in a received WallClockMessage that is a response and this will represent the candidate data derived from that request-response interaction.

Populates properties of this objects with the candidate information. *t1*, *t2*, *t3* and *t4* represent the times of message sending and receiving as shown below:




It also calculates and provides, as properties, the round-trip time (*rtt*) and clock offset estimate (*offset*) based on this measurement.

All data is in units of nanoseconds, except for precision which is measured in seconds, and maximum frequency error which is measured in ppm.

A helper function *calcCorrelationFor()* makes it easy to calculate the *Correlation* needed to take this measurment candidate and use it to control a *CorrelatedClock*.

**t1 = msg.originateNanos**
(read only) The time "t1" at which the request was sent in the request-response measurement (nanoseconds)

**t2 = msg.receiveNanos**
(read only) The time "t2" at which the request was received in the request-response measurement (nanoseconds)

**t3 = msg.transmitNanos**
(read only) The time "t3" at which the response was sent in the request-response measurement (nanoseconds)

**t4 = nanosRx**
(read only) The time "t4" at which the response was received in the request-response measurement (nanoseconds)

**offset = ((t3+t2)-(t4+t1))/2**
(read only) Server<->client clock offset (nanoseconds)

**rtt = (t4-t1)-(t3-t2)**
(read only) Round trip time (nanoseconds)

**isNanos = True**

**precision = msg.getPrecision()**
(read only) The precision reported by the server in its response (units of factions of a second)

**maxFreqError = msg.getMaxFreqError()**
(read only) The maximum frequency error reported by the server in its response (units of ppm)

**msg = WCMessage**
(read only *WCMessage*) The response message from which this candidate was derived

**calcCorrelationFor**(*clock*, *localMaxFreqErrorPpm=None*)
Calculates and returns the *Correlation* for a *CorrelatedClock* that is equivalent to this candidate.

The returned correlation can then be applied to the clock to model the time at the server. This includes the error bounds information needed to enable the clock to correctly calculate dispersion.

**Parameters**
- **clock** – *CorrelatedClock* that will model the server clock. Its parent must be the one that was measured for *t1* and *t4* this candidate.
- **localMaxFreqErrorPpm** – Optional. By defeault the *getRootMaxFreqError()* of the *clock* is used. Provide this value to override that.

**Returns** *Correlation* representing this *candidate*, and that can be used with the *CorrelatedClock*.

---

**Note:** The parameters of the correlation are calculated by this function as follows:
- **parentTicks** = (t1' + t4') / 2
- **childTicks** = (t2' + t3') / 2
- **initialError** = precision + ( rtt/2 + mfeC * (t4 - t1) + mfeS * (t3 - t2) ) / $10^9$
- **errorGrowthRate** = mfeC + mfeS

Where:
- **t1**, **t2**, **t3** and **t4** are in units of nanoseconds
- **t1'** and **t4'** are the same as t1 and t4 but converted to ticks of the parent of the specified clock
- **t2'** and **t3'** are the same as t2 and t3 but converted to ticks of the specified clock
- **mfeC** is the clock's *getRootMaxFreqError()*, converted from ppm to a fraction by dividing by $10^6$
- **mfeS** is the max freq error reported by the server, converted from ppm to a fraction by dividing by $10^6$

---

New in version 0.4.

**toTicks**(*clock*)

Returns a new Candidate object the same as this one but whose measurements have been converted to match the timescale of a clock.

**Raises** **NotImplementedError** – This function has been deprecated.

> **Warning:** Deprecated since version 0.4: This function has been deprecated because of the architectural change to taking taking readings from a different clock to the one that is adjusted.
>
> Use *calcCorrelationFor()* instead as this will perform the necessary conversion to clock tick rate units as well as creating the correlation needed to configure that clock.

### 2.3.3 CSS-WC Clients

Modules: *dvbcss.protocol.client.wc* | *dvbcss.protocol.client.wc.algorithm*



The *WallClockClient* class provides a complete Wall Clock Client implementation. It is used in conjunction with an *algorithm* for adjusting a *clock* object so that it mirrors the server's Wall Clock.

### Using a Wall Clock Client

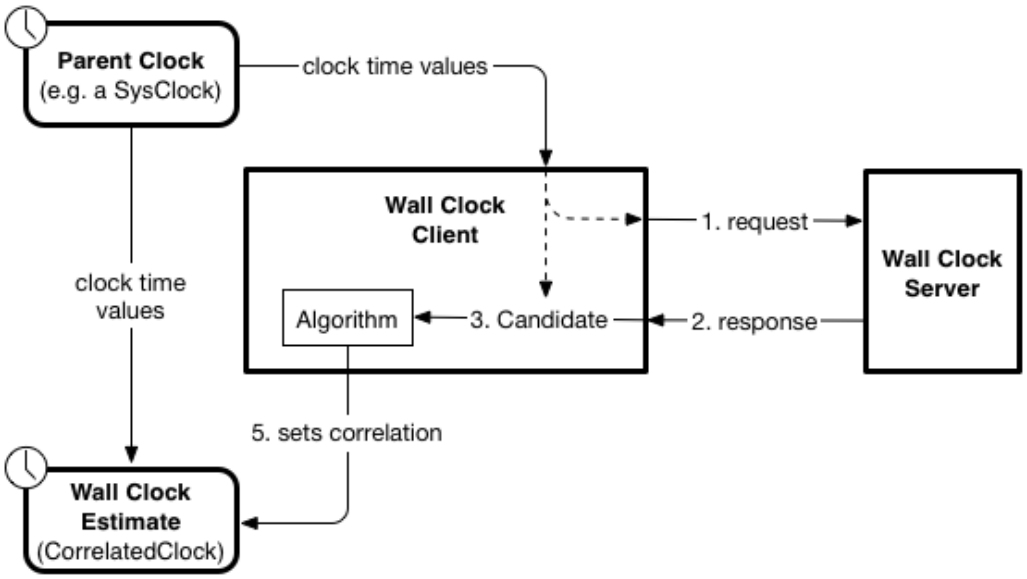
Recommended simplest use is to instantiate a *WallClockClient* and provide it with an instance of the *LowestDispersionCandidate* algorithm to control how it updates a clock.

A simple example that connects to a wall clock server at 192.168.0.115 port 6677 and sends requests once per second:

```
from dvbcss.clock import SysClock as SysClock
from dvbcss.protocol.client.wc import WallClockClient
from dvbcss.protocol.client.wc.algorithm import LowestDispersionCandidate

sysClock=SysClock()
wallClock=CorrelatedClock(sysClock,tickRate=1000000000)

algorithm = LowestDispersionCandidate(wallClock,repeatSecs=1,timeoutSecs=0.5)

bind = ("0.0.0.0", 6677)
server = ("192.168.0.115", 6677)

wc_client=WallClockClient(bind, server, wallClock, algorithm)
wc_client.start()

# wall clock client is now running in the background
```

After the *start()* method is called, the WallClockClient runs in a separate thread and execution continues.

*WallClockClient* is used by providing it with an object implementing the clock synchronisation *algorithm*. The Wall Clock client handles the protocol interaction (sending the requests and parsing the responses) at the times specified by the algorithm and passes the results of each request-response measurement (a *Candidate*) to the algorithm. The algorithm then adjusts the clock.




Changed in version 0.4: The measurement process involves taking a reading from the local clock when the request is about to be sent, and when the response is received. This measurement is taken from the *parent* of the clock you provide. The candidate represents a possible relationship between that (parent) clock and the Wall Clock of the server given the results of the request-response measurement. The algorithm processes this and makes a decision as to the *Correlation* that is to be used.

Although the WallClockClient class does not require the tickrate of the Wall Clock to be 1 tick per nanosecond, it is recommended to set it as such. This is because the next step (using the CSS-TS protocol) assumes a Wall Clock

ticking at this rate.

### Algorithms

The algorithm object determines when (and how often) a WallClockClient sends CSS-WC protocol requests and processes the results of requests and responses to adjust the *clock* object representing the Wall Clock.

### Dispersion algorithm

The *LowestDispersionCandidate* algorithm is the recommended algorithm. It uses the candidate with the lowest dispersion.

### Simple algorithm

The *MostRecent* algorithm is a simple naive algorithm that uses the most recent candidate irrespective of its quality (e.g. how long the round-trip time of the measurement was)

### Writing new algorithms

An algorithm is an object that has the following method:

**.algorithm**(*self*)

A python generator function that yields whenever it wants a Wall Clock protocol measurement to be taken:

```
candidateOrNone = yield timeoutSecs
```

The yield must pass a timeout in seconds. This is the maximum amount of time the Wall Clock client will wait for a response to its request before timing out.

Changed in version 0.4: The yield statement will return either `None` or a *Candidate* object representing the result of the measurement.

The algorithm can then use the Candidate object in its algorithm for estimating the wall clock (and in the case of most practical implementations: adjusting a *clock* object).

Here is an example of a simple naive algorithm that adjusts a *CorrelatedClock* object using the most recent measurement, irrespective of influencing factors such as previous measurements or network latency (round trip time). It makes requests at most once per second and has a timeout on waiting for responses of 0.5 seconds:

```
class NaiveAlgorithm(object):

    def __init__(self,clock):
        self.clock = clock

    def algorithm(self):
        while True:
            candidate=(yield 0.5)
            if candidate is not None:
                self.clock.correlation = candidate.calcCorrelationFor(self.clock)
                time.sleep(1.0)
```

### Composable Filter and prediction algorithm

There is also a simple modular framework for building up an algorithm out of two parts:

- *Filters* - processes measurement candidates, determining whether to reject them.
- *A Predictor* - takes the measurement candidates outputted from the filtering step and use them to adjust the clock.

Use the *FilterAndPredict()* function to compose zero, one or more Filters, and a Predictor into an algorithm that can be used with a *WallClockClient*.

Changed in version 0.4: When using this algorithm, you provide a *CorrelatedClock* object to it, whose parent is the clock given to the *WallClockClient*. This algorithm controls the *CorrelatedClock* object by settings its *correlation* property to that returned by the predictor. The parent of this clock is the clock used by the *WallClockClient* in generating the measurement candidates. So the job of the predictor is to determine the best correlation for modelling the relationship between that (parent) clock and the wall clock of the server.

Here is a simple example that uses a simple predictor and a round-trip-time threshold filter:

```
from dvbcss.clock import CorrelatedClock, SysClock
from dvbcss.protocol.client.wc.algorithm import FilterAndPredict, FilterRttThreshold,␣
↪PredictSimple
from dvbcss.protocol.client.wc import WallClockClient

sysClock = SysClock(tickRate=1000000000)
wallClock = CorrelatedClock(parentClock=sysClock, tickRate=1000000000)

filters = [ FilterRttThreshold(thresholdMillis=10.0) ]
predictor = PredictSimple(wallClock)

algorithm = FilterAndPredict(wallClock, repeatSecs, timeoutSecs, filters, predictor)

bind = ("0.0.0.0", 6677)
server = ("192.168.0.115", 6677)

wc_client=WallClockClient(bind, server, wallClock, algorithm)
wc_client.start()
```

The Clock object given to the algorithm must be *CorrelatedClock* whose parent is the clock object provided to the WallClockClient object.

### Round-trip time threshold Filter

The `FilterRttThreshold` class implements a filter that eliminates any candidate where the round-trip time exceeds a threshold.

### Lowest dispersion candidate filter

The `FilterLowestDispersionCandidate` class implements a filter that eliminates any candidate if it does not have a lower dispersion that any candidate that came before it.

### Simple Predictor

The `PredictSimple` class is a simple predictor that uses the candidate most recently provided to it and directly transforms that into a *dvbcss.clock.Correlation*.

### Writing your own Filter

You can write your own Filter by creating a class with the following method defined:

**.checkCandidate**(*self*, *candidate*)

**Parameters** **candidate** – A (`dict`) containing two *Candidate* objects representing the result of the measurement in units of ticks (dict key *"ticks"*) and units of nanoseconds (dict key *"nanos"*):

### Writing your own Predictor

You can write your own Predictor by creating a class with the following methods defined:

**.addCandidate**(*self*, *candidate*)

**param candidate** A (`dict`) containing two *Candidate* objects representing the result of the measurement in units of of nanoseconds.

This method is called whenever a measurement candidate resulting from a request-response measurement survives the filtering process.

**.predictCorrelation**(*self*)

**Returns** A *Correlation*

The returned Correlation must represent the relationship between the clock and its parent, such that the clock becomes an estimate of the server's Wall Clock. This must be in the correct units (tick rate) for the clock and its parent.

## Classes

### WallClockClient

**class** dvbcss.protocol.client.wc.**WallClockClient**(*(bindaddr, bindport)*, *(dstaddr, dstport)*, *wallClock*, *wcAlgorithm*)

Simple object implementing the client side of the CSS-TS protocol.

Use by initialising and supplying with an algorithm object and Clock object.

The *Candidate* objects provided to the algorithm represent the relationship between the parent of the provided clock and the server's Wall Clock.

The algorithm is then expected to set the *dvbcss.clock.Correlation* of the clock object such that it becomes an estimate of the server's Wall Clock.

It is recommended to use the *LowestDispersionCandidate* algorithm.

**Initialisation takes the following parameters:**

**Parameters**

- **(bindaddr,bindport)** – (`str`, `int`) A tuple containing the IP address (as a string) and port (as an int) to bind to to listen for incoming packets

- **(dstaddr,dstport)** – (str, int) A tuple containing the IP address (as a string) and port (as an int) of the Wall Clock server
- **wallClock** – (*clock*) The local clock that will be controlled to be a Wall Clock. Measurements will be taken from its parent and candidates provided to the algorithm will represent the relationship between that (parent) clock and the server's wall clock.
- **wcAlgorithm** – (*algorithm*) The algorithm for the client to use to update the clock.

**algorithm = None**
(read only) The *algorithm* object being used with this WallClockClient

**start**()
Call this method to start the client running. This function call returns immediately and the client proceeds to run in a thread in the background.

Does nothing if the client is already running.

**stop**()
Call this method to stop the client running. Returns once the client has stopped.

If the client is not running then nothing happens and this call returns immediately.

### Dispersion algorithm

**class** dvbcss.protocol.client.wc.algorithm.**LowestDispersionCandidate**(*clock*, *repeatSecs=1.0*, *timeoutSecs=0.2*, *localMaxFreqErrorPpm=None*)
Algorithm that selects the candidate (request-response measurement result) with the lowest dispersion.

Dispersion is a formal measure of the error bounds of the Wall Clock estimate. This value is the sum of possible error due to:

- measurement precision limitations (at both client and server)
- round-trip time
- maximum potential for oscillator frequency error at the client and at the server server This grows as the candidate (that was used to most recently adjust the clock) ages.

---

**Note:** The Clock object must be the same one that is provided to the WallClockClient, otherwise this algorithm will not synchronise correctly.

---

The tick rate of the Clock can be any tick rate (it does not have to be one tick per nanosecond), but too low a tick rate will limit clock synchronisation precision.

There is a stub callback function provided that you can override (e.g. by subclassing) that will be called whenever the clock is adjusted:

- *onClockAdjusted()*

The arguments to this method provide details of when the adjustment took place and what adjustment was made. It also reports the dispersion before and after the adjustment and gives information needed to extrapolate future dispersions. You can use this, for example, to record the clock dispersion over time.

*Initialisation takes the following parameters:*

**Parameters**

- **clock** – A `Correlated` object representing that will be adjusted to match the Wall Clock.
- **repeatSecs** – (`float`) The rate at which Wall Clock protocol requests are to be sent (in seconds).
- **timeoutSecs** – (`float`) The timeout on waiting for responses to requests (in seconds).
- **localMaxFreqErrorPpm** – Optional. Override using the *getRootMaxFreqError()* of the clock as the max freq error of the local clock, and instead use this value. It is the clock maximum frequency error in parts-per-million

**getCurrentDispersion**()

**Returns** Current dispersion at this moment in time in units of nanoseconds.

**getWorstDispersion**()

Returns the worst (greatest) dispersion encountered since the previous time this method was called.

The first time it is called, the value reported will be very large, reflecting the fact that initially dispersion is high because the client is not yet synchronised.

**Returns** dispersion

**onClockAdjusted**(*timeAfterAdjustment*, *adjustment*, *oldDispersionNanos*, *newDispersionNanos*, *dispersionGrowthRate*)

This method is called immediately after a clock adjustment has been made, and gives details on how the clock was changed and the effect on the dispersion.

**Parameters**

- **timeAfterAdjustment** – The wall clock time (in ticks) immedaitely after the adjustment took place
- **adjustment** – The amount by which the wall clock instaneously changed (in ticks)
- **oldDispersionNanos** – The dispersion (in nanoseconds) just prior to adjustment
- **newDispersionNanos** – The dispersion (in nanoseconds) immediately after adjustment
- **dispersionGrowthRate** – The rate at which dispersion will continue to grow.

**To calculate a future dispersion at time T:** futureDispersion = newDispersionNanos + (t-timeOfAdjustment) * dispersionGrowthRate

*This is a stub for this method. Sub-classes should implement it.*

### Most recent measurement algorithm

**class** dvbcss.protocol.client.wc.algorithm.**MostRecent**(*clock*, *repeatSecs=1.0*, *timeoutSecs=0.2*)

Simple (naive) Wall Clock client algorithm.

Crash locks the supplied clock based on the offset observed in the result of every successful request/response.

**Note:** The Clock object must be the same one that is provided to the WallClockClient, otherwise this algorithm will not synchronise correctly.

The tick rate of the Clock can be any tick rate (it does not have to be one tick per nanosecond), but too low a tick rate will limit clock synchronisation precision.

*Initialisation takes the following parameters:*

**Parameters**

- **clock** – A *CorrelatedClock* object representing that will be adjusted to match the Wall Clock.
- **repeatSecs** – (float) The rate at which Wall Clock protocol requests are to be sent (in seconds).
- **timeoutSecs** – (float) The timeout on waiting for responses to requests (in seconds).

**getCurrentDispersion**()
Calling this method will always result in *ValueError* being raised.

**Raises** **ValueError** – This algorithm does not model clock synchronisation accuracy.

### Filter and Prediction composable algorithms

### Filters

**class** dvbcss.protocol.client.wc.algorithm._filterpredict.**FilterRttThreshold**(*thresholdMillis=1.0*)
Simple filter that rejects all candidates where round trip time exceeds a specified threshold.

**Parameters** **thresholdMillis** – (float) The threshold to use (in milliseconds)

**class** dvbcss.protocol.client.wc.algorithm._filterpredict.**FilterLowestDispersionCandidate**(*cl*
Simple filter that will reject a candidate unless its dispersion is lower than that currently being used by the clock.

Note that at initialisation, the clock's dispersion is forced to +infinity.

**Parameters** **clock** – A *CorrelatedClock* object representing that will be adjusted to match the Wall Clock.

### Predictors

**class** dvbcss.protocol.client.wc.algorithm._filterpredict.**PredictSimple**(*clock*)
Simple naive predictor that chooses the correlation represented by the most recent candidate

**Parameters** **clock** – The *CorrelatedClock* that is to be set.

Note that this predictor does not actually set the clock. It just needs it in order to calculate the correct correlation for it.

### Functions

#### Filter and Prediction algorithm creator

dvbcss.protocol.client.wc.algorithm.**FilterAndPredict**(*clock*, *repeatSecs=1.0*, *timeoutSecs=0.2*, *filters=[]*, *predictor=None*)

Combines zero, one or more Filters and a Predictor and returns an algorithm for a WallClockClient.

**Parameters**

- **clock** – A *CorrelatedClock* object that will be adjusted to match the Wall Clock.
- **repeatSecs** – (float) The rate at which Wall Clock protocol requests are to be sent (in seconds).
- **timeoutSecs** – (float) The timeout on waiting for responses to requests (in seconds).
- **filters** – (list of Filters) A list of zero, one or more Filters
- **predictor** – (Predictor) The Predictor to use, or None to default to PredictSimple

This algorithm controls the *CorrelatedClock* object by settings its *correlation* property to that provided by the predictor.

The parent of this *clock* is the clock used by the *WallClockClient* in generating the measurement candidates. So the job of the predictor is always to estimate the *Correlation* needed by the *clock*.

Requests are made at the repetition rate specified. If a response is received within the timeout period it is then it is transformed into a measurement candidate and passed to the filters. Filters are applied in the order you list them. If a candidate survives filtering, then it is passed to the predictor. Every time a candidate is provided to the predictor, the correlation returned by the predictor replaces the previous correlation.

---

**Note:** The Clock object must be *CorrelatedClock* whose parent is the clock object that is measured when generating the candidates.

---

The tick rate of the Clock can be any tick rate (it does not have to be one tick per nanosecond), but too low a tick rate will limit clock synchronisation precision.

#### Candidate quality calculator

This function is used internally by the *WallClockClient* class.

dvbcss.protocol.client.wc.algorithm.**calcQuality**(*reqMsg*, *respMsg*)

Generate measure of how good the response was. Quality < 0 means response corresponded to a different request.

Quality = 3 or 4 means it was a response for which no follow-up is expected or a follow-up response.

Quality = 2 means it was a response for which a follow-up is expected

## 2.3.4 CSS-WC Servers

Modules: *dvbcss.protocol.server.wc*



The `WallClockServer` class implements a standalone CSS-WC server.

The server can be started and stopped. While running it runs in its own separate thread of execution in the background.

An *example* server is provided in this package.

### Example usage

To use it, you need to provide a `clock` object that represents the "wall clock". Although it is not required, it is recommended to set the tick rate of that clock to match the required tick rate of the wall clock (1e9 ticks per second).

First, create the clock:

```
from dvbcss.clock import SysClock

mfe = 45

sysClock = SysClock(tickRate=1000000000, maxFreqErrorPpm=mfe)
wallClock = sysClock
```

The server will need to know the potential maximum frequency error and measurement precision of the clock. Fortunately the SysClock already estimates this and it is passed through any dependent clocks.

Fortunately the SysClock internally estimates the measurement precision automatically when it is created. It also defaults to assuming the maximum frequency error is 500ppm, unless you specify otherwise.

Maximum frequency error will depend on oscillator accuracy in the hardware the code is running on, and whether an NTP client is running (and which therefore may slew the clock).

For example, above we have guessed that the combined worst case of NTP client slew and hardware oscillator accuracy is approx 45 ppm:

So next we create and start the wall clock server.

```
from dvbcss.protocol.wc.server import WallClockServer

wcServer = WallClockServer(wallClock)
wcServer.start()
```

### Classes

### WallClockServer

**class** dvbcss.protocol.server.wc.**WallClockServer**(*wallClock*, *precision=None*, *maxFreqError=None*, *bindaddr='0.0.0.0'*, *bindport=6677*, *followup=False*)

A CSS-WC server.

Pass it a *clock* object and information on clock precision and frequency stability, and tell it which network interface to listen on.

Call start() and stop() to start and stop the server. It runs in its own separate thread in the background.

You can optionally ask this server to operate in a mode where it will send *follow-up* responses. Note, however, that in this implementation the transmit-timevalue reported in the follow-up response is not guaranteed to be more accurate. This option exists primarily to check whether a Wall Clock Client has implemennted handling of *follow-up* responses at all.

**Parameters**

- **wallClock** – (:class:dvbcss.clock.ClockBase) The clock to be used as the wall clock for protocol interactions
- **precisionSecs** – (float) Optional. Override using the precision of the provided clock and instead use this value. It is the precision (in seconds) to be reported for the clock in protocol interactions
- **maxFreqErrorPpm** – (float) Optional. Override using the `rootMaxFreqError()` of the clock and instead use this value. It is the clock maximum frequency error in parts-per-million
- **bindaddr** – (str, ip address) The ip address of the network interface to bind to, e.g. "127.0.0.1". Defaults to "0.0.0.0" which binds to all interfaces.
- **bindport** – (int) The port number to bind to (defaults to 6677)
- **followup** – (bool) Set to True if the Wall Clock Server should send follow-up responses. Defaults to False.

**run**()

Internal method - the main runloop of the thread.

Runs in a loop calling the `handle()` method of the object assigned to the `handler` property whenever a UDP packet is received.

Does not return until the _pleaseStop attribute of the object has been set to True

**start**()

Starts the wall clock server running. It runs in a thread in the background.

**stop**()

Stops the wall clock server running. Does not return until the thread has terminated.

This package provides objects for representing messages exchanged via the DVB CSS-WC protocol and for implementing clients and servers.

The WC protocol is a simple UDP request response-protocol that enables simple clock synchronisation algorithms to be used to establish a common *wall clock* between a server and client.

There is a *WallClockServer* class providing a self contained Wall Clock server. The *WallClockClient* is designed to allow different algorithms to be plugged in for acting on the results of the request-response interaction to adjust a local *clock* object to match the Wall Clock of the server.

Modules for using the CSS-WC protocol:

- *dvbcss.protocol.wc* : objects for representing and packing/unpacking the protocol messages.
- *dvbcss.protocol.client.wc* : implementation of a client for a CSS-WC connection.
- *dvbcss.protocol.client.wc.algorithm* : algorithms to be used with a CSS-WC client.
- *dvbcss.protocol.server.wc* : implementation of a server for a CSS-WC connection.

See *Protocol server implementation details* for information on how the servers are implemented.

## 2.4 Common types and objects

### 2.4.1 Signalling that a property is to be omitted from a message

dvbcss.protocol.**OMIT object**

When this object is assigned to an attribute of a protocol message object this indicates that the corresponding property is not included in the JSON representation of that message (it is omitted).

Here is an example. By default nearly all properties of a freshly created CII message object are 'OMIT':

```
>>> from dvbcss.protocol.cii import CII
>>> from dvbcss.protocol import OMIT

>>> c=CII()
>>> print repr(c.contentId)
OMIT

>>> c.wcUrl = "udp://192.168.1.1:6677"
>>> print repr(c.wcUrl)
'udp://192.168.1.1:6677'

>>> c.wcUrl = OMIT
>>> print repr(c.wcUrl)
OMIT
```

### 2.4.2 Private data

Some protocol messages contain optional properties to carry private data.

Private data is encoded in message objects here as a `list` of `dict` objects where each has a key "type" whose value is a URI.

Each dict can contain any other keys and values you wish so long as they can be parsed by the python `json` module's encoder. For example:

```
example_private_data = [
    { "type" : "urn:bbc.co.uk:pid", "pid":"b00291843",
      "entity":"episode"
    },
    { "type" : "tag:bbc.co.uk/programmes/clips/link-url",
      "http://www.bbc.co.uk/programmes/b1290532/"
    }
]
```

### 2.4.3 Exceptions

**class** dvbcss.protocol.client.**ConnectionError**
Exception that is raised when it was not possible to open a connection to a server.

CHAPTER 3

# Clocks, Time and Scheduling modules

This library contains a range of tools for dealing with timing, clocks, and timelines and scheduling code to run at set times.

Contents:

## 3.1 Montonic time functions

Module: *dvbcss.monotonic_time*



This module implements operating system specific access to high resolution monotonic system timers for the following operating systems:

- Windows 2000 Pro or later
- Linux
- Mac OS X

It implements a *time()* function and *sleep()* function that work the same as the time.time() and time.sleep() functions in the python standard library.

It also adds a *timeNanos()* and *timeMicros()* variants that report time in units of nanoseconds or microseconds instead of seconds.

See operating system specific implementation details below to understand the limitations of the functions provided in this module.

**Note:** For all supported operating systems, the sleep() function is not guaranteed to use the same underlying timer as the time() and therefore should be considered inaccurate.

### 3.1.1 Example use

```
>>> import dvbcss.monotonic_time as monotonic_time
>>> monotonic_time.time()
4695.582637038
>>> monotonic_time.timeNanos()
4700164952506L
>>> monotonic_time.timeMicros()
4703471405L
>>> monotonic_time.sleep(0.5)    # sleep 1/2 second
```

### 3.1.2 Operating system implementation details

The precision and accuracy of the clocks and sleep functions are dependent on the host operating system and hardware. This module can therefore provide no performance guarantees.

#### Windows

Windows NT 5.0 (Windows 2000 Professional) or later is supported, including the cygwin environment.

The *time()* function and its variants are based on the QueryPerformanceCounter() high resolution timer system call. This clock is guaranteed to be monotonic and have 1 microsecond precision or better.

The *sleep()* function is based on the CreateWaitableTimer() and SetWaitableTimer() system calls.

Note that the *sleep()* function for Windows is not guaranteed to be accurate because it is not possible to create blocking (non polling) delays based on the clock source used. However it should have significantly higher precision than the standard 15ms windows timers and will be fine for short delays.

#### Mac OS X

The *time()* function and its variants are based on the mach_absolute_time() system call.

The clock is guaranteed to be monotonic. Apple provides no guarantees on precision, however in practice it is usually based on hardware tick counters in the processor or support chips and so is extremely high precision (microseconds or better).

The *sleep()* function is based on the nanosleep() system call. It is unclear whether this uses the same underlying counter as *mach_absolute_time()*.

#### Linux

The `time()` function and its variants are based on the clock_gettime() system call requesting *CLOCK_MONOTONIC*.

The `sleep()` function is based on the nanosleep() system call. It is unclear whether this uses the same underlying counter as *CLOCK_MONOTONIC*.

### 3.1.3 Functions

#### time(), timeMicros() and timeNanos()

dvbcss.monotonic_time.**time**()
Return monotonic time in seconds and fractions of seconds (as a float). The precision is operating system dependent.

dvbcss.monotonic_time.**timeMicros**()
Return monotonic time in integer microseconds. The precision is operating system dependent.

dvbcss.monotonic_time.**timeNanos**()
Return monotonic time in integer nanoseconds. The precision is operating system dependent.

#### sleep()

dvbcss.monotonic_time.**sleep**(*t*)
Sleep for specified number of second and fractions of seconds (as a float). The precision is operating system dependent.

**Throws TimeoutError** if the underlying system call used to sleep reported a timeout (OS dependent behaviour)

**Throws InterruptedException** if a signal or other interruption is received while sleeping (OS dependent behaviour)

---

**Note:** For all supported operating systems, the sleep() function is not guaranteed to use the same underlying timer as the time() and therefore should be considered inaccurate.

---

## 3.2 Synthesised clocks (dvbcss.clock)

Module: *dvbcss.clock*

The dvbcss.clock module provides software synthesised clock objects that can be chained together into dependent hierarchies. Use in conjunction with `dvbcss.task` to sleep and run code at set times on these clocks.

### 3.2.1 Introduction

The classes in this module implement software synthesised clocks from which you can query a current time value. A clock counts in whole numbers of ticks (unlike the standard library `time.time()` function which counts in seconds and fractions of a second) and has a tick rate (expresed in ticks per second).

To use clocks as timers or to schedule code to run later, you must use them with the functions of the `dvbcss.task` module.

To use clocks begin with a root clock that is based on a underlying timing source. The following root clocks are provided:

- `SysClock` **is an root clock based on the** `dvbcss.monotonic_time.time()` **function as the underlying time source**

Other dependent clocks can then be created that have the underlying clock as their parent. and further dependent clocks can be created with those dependents as their parents, creating chains of clocks, each based on their parent and leading back to an underlying clock. The following dependent clocks are provided:

- `CorrelatedClock` **is a fixed tick rate clock where you define the point of correlation between it and its parent.**
- `OffsetClock` ** is a clock that is the same as its parent but with an offset amount of root time. Useful to calibrate for rendering pipeline delays.
- `TunableClock` **is a clock that can have its tick count tweaked and its frequency slewed on the fly.**
- `RangleCorrelatedClock` **is a clock where the relationship to the parent is determined from two points of correlation.**

Dependent clocks can have a different tick rate to their parent and count their ticks from a different starting point.

Dependent clocks allow their relationship to its parent clock to be changed dynamically - e.g. the absolute value, or the rate at which the clock ticks. If your code needs to be notified of such a change, it can bind itself as a dependent to that clock.

The base *ClockBase* class defines the set of methods common to all clocks (both underlying and dependent)

### 3.2.2 Limitations on tick resolution (precision)

The Clock objects here do not run a thread that counts individual ticks. Instead, to determine the current tick value they query the parent clock for its current tick value and then calculate what the tick value should be.

A clock is therefore **limited in the precision and resolution of its tick value by its parents.** *SysClock*, for example, is limited by the resolution of the underlying time source provided to it by *dvbcss.monotonic_time* module's *dvbcss.monotonic_time.time()* function. And this will be operating system dependent. SysClock also outputs ticks as integer values.

If a parent of a clock only reports whole number (integer) tick values then that also limits the resolution of any clocks that depend on it. For example, a clock that counts in integer ticks only at 25 ticks per second will cause a clock descended from it, with a tick rate of 100 ticks per second, to report tick values that increment 4 ticks at a time ... or worse if a parent of both has an even lower tick rate.

With the clocks provided by this module, only *SysClock* limits itself to integer numbers of ticks. *CorrelatedClock* and *TunableClock* are capable of fractional numbers of ticks provided that the parameters provided to them (e.g. the tickRate) are passed as floating point values (this will force python to do the maths in floating point instead of integer maths).

### 3.2.3 Timers and Sleep functions

Use the functions of the *dvbcss.task* in conjunction with Clock objects to create code that sleeps, or which triggers callbacks, based on time as measured by these clocks.

### 3.2.4 Accounting for error (dispersion)

These clocks also support tracking and calculating error bounds on the values they report. This is known as dispersion. It is useful if, for example, a clock hierarchy is being used to estimate time based on imperfect measurements - such as when a companion tries to estimate the Wall Clock of a TV.

Each clock in a hierarchy makes its own contribution to the overall dispersion. When a clock is asked to calculate dispersion (using the `dispersionAtTime()` method), the answer it gives is the sum of the error contributions from itself and its parents, all the way back up to the root system clock.

A system clock has error bounds determined by the precision with which it can be measured (the smallest amount by which its values change).

When using a dependent clock, such as a *CorrelatedClock*, the correlation being used to set its relationship to its parent might also have some uncertainty in it. This information can be included in the *Correlation*. Uncertainty based on measurements/estimates can grow over time - e.g. because the clocks in a TV and companion gradually drift. So there are two parameters that you can provide in a *Correlation* - the *initial error* and the *growth rate*. As time passes, the error for the clock using this correlation is calculated as the initial error plus the growth rate multiplied by how much time has passed since the point of correlation.

### 3.2.5 Usage examples

#### Simple hierarchies of clocks

Here is a simple example where a clock represents a timeline and another represents a timeline related to the first by a correlation:

```
from dvbcss.clock import SysClock
from dvbcss.clock import CorrelatedClock
from dvbcss.clock import Correlation

# create a clock based on dvbcss.monotonic_time.time() that ticks in milliseconds
sysClock = SysClock(tickRate=1000)

# create a clock to represent a timeline
corr = Correlation(parentTicks=0, childTicks=0)
baseTimeline = CorrelatedClock(parentClock=sysClock, tickRate=25, correlation=corr)

# create a clock representing another timeline, where time zero corresponds to time␣
↪100
# on the parent timeline
corr2 = Correlation(parentTicks=100, childTicks=0)
subTimeline = CorrelatedClock(parentClock=baseTimeline, tickRate=25,␣
↪correlation=corr2)
```

At some point later in time during the program, we query the values of all the clocks, confirming that the sub timeline is always 100 ticks ahead of the base timeline.

```
def printTimes():
    sys  = sysClock.ticks()
    base = baseTimeline.ticks()
    sub  = subTimeline.ticks()
    print "SysClock      ticks = %d", sys
    print "Base timeline ticks = %d", base
    print "Sub timeline  ticks = %d", sub

>>> printTimes()
SysClock      ticks = 20000
Base timeline ticks = 500
Sub timeline  ticks = 600
```

*Note that in these examples, for clarity, the tick count on the sysClock is artificially low. It would likely be a much larger value.*

We then change the correlation for the base timeline, declaring tick 25 on its baseline to correspond to tick 0 on its parent timeline, and both the base timeline and the sub timeline reflect this:

```
>>> baseTimeline.correlation = Correlation(0,25)
>>> printTimes()
SysClock      ticks = 30000
Base timeline ticks = 775
Sub timeline  ticks = 875
```

#### Clock speed adjustment

All clocks have a `speed` property. Normally this is 1.0. Some clock classes support changing this value. This scales the rate at which the clock ticks relative to its parent. For example, 0.5 corresponds to half speed; 2.0 = double speed,

0 = frozen and -1.0 = reverse.

Clocks will take speed into account when returning their current tick position or converting it to or from the tick value of a parent clock. However it does not alter the tickRate property. A child clock will similarly ignore the speed property of a parent clock. In this way, the speed property can be used to tweak the speed of time, or to emulate speed control (fast forward, rewind, pause) for a media timeline.

Here is an example where we create 3 clocks in a chain and all tick initially at 100 ticks per second:

```
>>> import time
>>> baseClock = Sysclock(tickRate=100)
>>> clock1 = TunableClock(parent=baseClock, tickRate=100)
>>> clock2 = TunableClock(parent=clock1, tickRate=100)
```

We confirm that both clock1 and its child - clock2 - tick at 100 ticks per second:

```
>>> print clock1.ticks; time.sleep(1.0); print clock1.ticks
5023
5123
>>> print clock2.ticks; time.sleep(1.0); print clock2.ticks
2150
2250
```

If we change the tick rate of clock1 this affects clock1, but its child - clock2 - continues to tick at 100 ticks every second:

```
>>> clock1.tickRate = 200
>>> print clock1.ticks; time.sleep(1.0); print clock1.ticks
5440
5640
>>> print clock2.ticks; time.sleep(1.0); print clock2.ticks
4103
4203
```

But if we *instead* change the speed multiplier of clock1 then this not only affects the ticking rate of clock1 but also of its child - clock2:

```
>>> clock1.tickRate = 100
>>> clock1.speed = 2.0
>>> print clock1.ticks; time.sleep(1.0); print clock1.ticks
5740
5940
>>> print clock2.ticks; time.sleep(1.0); print clock2.ticks
4603
4803
```

### Translating tick values between clocks

The clock classes provide mechanisms to translate a tick value from one clock to a tick value of another clock such that it still represents the same moment in time. So long as both clocks share a common ancestor clock, the conversion will be possible.

*toParentTicks()* and *fromParentTicks()* converts tick values for a clock to/from its parent clock. *toOtherClockTicks()* will convert a tick value for this clock to the corresponding tick value for any other clock with a common ancestor to this one.

```
from dvbcss.clock import SysClock
from dvbcss.clock import CorrelatedClock
from dvbcss.clock import Correlation

# create a clock based on dvbcss.monotonic_time.time() that ticks in milliseconds
sysClock = SysClock(tickRate=1000)

#                                       +-------------+
#                                  .--  | mediaClock  |
# +-----------+     +------------+ <--'   +-------------+
# | sysClock  | <-- | wallClock  |
# +-----------+     +------------+ <--.   +-----------------+
#                                  '--  | otherMediaClock |
#                                       +-----------------+

wallClock = CorrelatedClock(parentClock=sysClock, tickRate=1000000000,␣
↪correlation=Correlation(0,0))
mediaClock = CorrelatedClock(parentClock=wallClock, tickRate=25,␣
↪correlation=Correlation(500021256, 0))
otherMediaClock = CorrelatedClock(parentClock=wallClock, tickRate=30,␣
↪correlation=Correlation(21093757, 0))

# calculate wall clock time 'w' corresponding to a mediaClock time 1582:
t = 1582
w = mediaClock.toParentTicks(t)
print "When MediaClock ticks =", t, " wall clock ticks =", w

# calculate mediaClock time 'm' corresponding to wall clock time 1920395
w = 1920395
m = mediaClock.fromParentTicks(w)
print "When wall clock ticks =", w, " media clock ticks =", m

# calculate other media clock time corresponding to media clock time 2248
t = 2248
o = mediaClock.toOtherClockTicks(otherMediaClock, t)
print "When MediaClock ticks =", t, " other media clock ticks =", o
```

Let us now suppose that the wall clock is actually an estimate of a Wall Clock on another device. When we set the correlation we therefore include error information that is calculated from the proces by which the Wall Clock of the other device is estimated:

```
wallClock.correlation = Correlation(24524535, 34342, initialError=0.012,␣
↪errorGrowthRate=0.00005)
```

Here we are saying that the error bounds of the estimate at the point in time represented by the correlation is +/- 0.012 seconds. This will grow by 0.00005 seconds for every tick of the parent clock.

```
print "Dispersion is +/-", wallClock.dispersionAtTime(wallClock.ticks), "seconds"

print "In 1000 ticks from now, dispersion will be +/-", wallClock.
↪dispersionAtTime(wallClock.ticks + 1000), "seconds"
```

### Implementing new clocks

Implement a new clock class by subclassing *ClockBase* and implementing the stub methods.

For example, here is a clock that is the same as its parent (same tick rate) except that its current tick value differs by a fixed offset.

```
from dvbcss.clock import ClockBase

class FixedTicksOffsetClock(ClockBase):

    def __init__(self, parent, offset):
        super(FixedTicksOffsetClock,self).__init__()
        self._parent = parent
        self._offset = offset

    def calcWhen(self, ticksWhen):
        return self._parent.calcWhen(ticksWhen - self._offset)

    def fromParentTicks(self, ticks):
        return ticks + self._offset

    def getParent(self):
        return self._parent

    @property
    def tickRate(self):
        return self._parent.tickRate

    @property
    def ticks(self)
        return self._parent.ticks + self._offset

    def toParentTicks(self, ticks):
        return ticks - self._offset

    def _errorAtTime(self,t):
        return 0
```

In use:

```
>>> f = FixedTicksOffsetClock(parentClock, 100)
>>> print parentClock.ticks, f.ticks
216 316
```

When doing this, you must decide whether to allow the speed to be adjusted. If you do, then it must be taken into account in the calculations for the methods: `calcWhen()`, `fromParentTicks()` and `toParentTicks()`.

### 3.2.6 Not a number (nan)

"Not a number" value of a float. Check if a value is NaN like this:

```
>>> import math
>>> math.isnan(nanValue)
True
```

Converting tick values to a parent clock or to the root clock may result in this value being returned if one or more of the clocks involved has speed zero.

### 3.2.7 Functions

#### measurePrecision - estimate measurement precision of a clock

dvbcss.clock.**measurePrecision**(*clock*, *sampleSize=10000*)

Do a very rough experiment to work out the precision of the provided clock.

Works by empirically looking for the smallest observable difference in the tick count.

**Parameters** **sampleSize** – (int) Number of iterations (sample size) to estimate the precision over

**Returns** (float) estimate of clock measurement precision (in seconds)

### 3.2.8 Classes

#### ClockBase - base class for clocks

**class** dvbcss.clock.**ClockBase**(***kwargs*)

Base class for all clock classes.

By default, adjusting tickRate and speed are not permitted unless a subclass overrides and implements a property setter.

**bind**(*dependent*)

Bind for notification if this clock changes.

**Parameters** **dependent** – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

**calcWhen**(*ticksWhen*)

**Return** "when" in terms of the underlying clock behind the root clock implementation (e.g. monotonic_time.time() in the case of *SysClock*). Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

*This is a stub for this method. Sub-classes should implement it.*

**clockDiff**(*otherClock*)

Calculate the potential for difference between this clock and another clock.

**Parameters** **otherClock** – The clock to compare with.

**Returns** The potential difference in units of seconds. If effective speeds or tick rates differ, this will always be float ("+inf").

If the clocks differ in effective speed or tick rate, even slightly, then this means that the clocks will eventually diverge to infinity, and so the returned difference will equal +infinity.

New in version 0.4.

**dispersionAtTime**(*t*)

Calculates the dispersion (maximum error bounds) at the specified clock time. This takes into account the contribution to error of this clock and its ancestors.

**Returns** Dispersion (in seconds) at the specified clock time.

New in version 0.4.

**fromParentTicks**(*ticks*)

*This is a stub for this method. Sub-classes should implement it.*

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**Returns** The specified tick value for the parent clock converted to the timescale of this clock.

**Throws StopIteration** if this clock has no parent

**fromRootTicks**(*t*)
Return the time for this clock corresponding to a given time of the root clock.

**Parameters** **t** – Tick value of the root clock.

**Returns** Corresponding tick value of this clock.

New in version 0.4.

**getAncestry**()
Retrieve the ancestry of this clock as a list.

**Returns** A list of clocks, starting with this clock, and proceeding to its parent, then its parent's parent etc, ending at the root clock.

New in version 0.4.

**getEffectiveSpeed**()
Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

**getParent**()
*This is a stub for this method. Sub-classes should implement it.*

**Returns** *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

**getRoot**()

**Returns** The root clock for this clock (or itself it has no parent).

New in version 0.4.

**getRootMaxFreqError**()
Return potential error of underlying clock (e.g. system clock).

**Returns** The maximum potential frequency error (in parts-per-million) of the underlying root clock.

*This is a partial stub method. It must be re-implemented by root clocks.*

For a clock that is not the root clock, this method will pass through the call to the same method of the root clock.

New in version 0.4.

**isAvailable**()
Returns whether this clock is available, taking into account the availability of any (parent) clocks on which it depends.

**Returns** True if available, otherwise False.

New in version 0.4.

**nanos**
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

**nanosToTicks**(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

**Parameters** **nanos** – nanoseconds value

**Returns** number of ticks equivalent to the supplied nanosecond value

**notify**(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

**Parameters** **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

**setAvailability**(*availability*)
Set the availability of this clock.

**Parameters** **availability** – True if this clock is available, otherwise False.

If setting the availability of this clock changes the overall availability of this clock (as returned by *isAvailable()*) then dependents will be notified of the change.

New in version 0.4.

**setParent**(*newParent*)
*This is a stub for this method. Sub-classes should implement it.*

Change the parent of this clock (if supported). Will generate a notification of change.

**speed**
(read/write `float`) The speed at which the clock is running. Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

**tickRate**
(read only)The tick rate (in ticks per second) of this clock.

*This is a stub for this method. Sub-classes should implement it.*

**ticks**
(read only) The tick count for this clock.

*This is a stub for this method. Sub-classes should implement it.*

**toOtherClockTicks**(*otherClock*, *ticks*)
Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters**

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

**Returns** The tick value of the *otherClock* that represents the same moment in time, or *Not a number (nan)*

**Throws NoCommonClock** if there is no common ancestor clock (meaning it is not possible to convert

**toParentTicks**(*ticks*)
*This is a stub for this method. Sub-classes should implement it.*

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's `tickRate` and `speed` properties when performing the conversion.

**Returns** The specified tick value of this clock converted to the timescale of the parent clock.

**Throws StopIteration** if this clock has no parent

**toRootTicks**(*t*)
Return the time for the root clock corresponding to a given time of this clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters** **t** – Tick value for this clock.

**Returns** Corresponding tick value of the root clock, or *Not a number (nan)*

New in version 0.4.

**unbind**(*dependent*)
Unbind from notification if this clock changes.

**Parameters** **dependent** – The dependent to unbind from receiving notifications.

### SysClock - Clock based on time module

**class** dvbcss.clock.**SysClock**(*tickRate=1000000*, *maxFreqErrorPpm=500*, ***kwargs*)
A clock based directly on the standard library timer function `monotonic_time.time()`. Returns *integer* ticks when its `ticks` property is queried.

The default tick rate is 1 million ticks per second, but a different tick rate can be chosen during initialisation.

**Parameters**

- **tickRate** – Optional (default=1000000). The tick rate of this clock (ticks per second).
- **maxFreqErrorPpm** – Optional (default=500). The maximum frequency error (in units of parts-per-million) of the clock, or an educated best-estimate of it.

The precision is automatically estimated using the `measurePrecision()` function when this clock is created. So creating a SysClock, particularly one with a low tickrate, may incur a noticeable delay at initialisation.

The measured precision is then reported as the dispersion of this clock.

It is not permitted to change the `tickRate` or `speed` property of this clock because it directly represents a system clock.

**Parameters** **tickRate** – (int) tick rate for this clock (in ticks per second)

**bind**(*dependent*)
Bind for notification if this clock changes.

**Parameters** **dependent** – When this clock changes, the `notify()` method of this dependent will be called, passing a single argument that is this clock.

**calcWhen**(*ticksWhen*)

**Return** "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*). Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

(Documentation inherited from *ClockBase*)

**clockDiff**(*otherClock*)

Calculate the potential for difference between this clock and another clock.

**Parameters** **otherClock** – The clock to compare with.

**Returns** The potential difference in units of seconds. If effective speeds or tick rates differ, this will always be `float` ("+inf").

If the clocks differ in effective speed or tick rate, even slightly, then this means that the clocks will eventually diverge to infinity, and so the returned difference will equal +infinity.

New in version 0.4.

**dispersionAtTime**(*t*)

Calculates the dispersion (maximum error bounds) at the specified clock time. This takes into account the contribution to error of this clock and its ancestors.

**Returns** Dispersion (in seconds) at the specified clock time.

New in version 0.4.

**fromParentTicks**(*ticks*)

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value for the parent clock converted to the timescale of this clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**fromRootTicks**(*t*)

Return the time for this clock corresponding to a given time of the root clock.

**Parameters** **t** – Tick value of the root clock.

**Returns** Corresponding tick value of this clock.

New in version 0.4.

**getAncestry**()

Retrieve the ancestry of this clock as a list.

**Returns** A list of clocks, starting with this clock, and proceeding to its parent, then its parent's parent etc, ending at the root clock.

New in version 0.4.

**getEffectiveSpeed**()

Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

**getParent**()

**returns** *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

(Documentation inherited from *ClockBase*)

**getRoot**()

**Returns** The root clock for this clock (or itself it has no parent).

New in version 0.4.

**getRootMaxFreqError**()

Return potential error of underlying clock (e.g. system clock).

**returns** The maximum potential frequency error (in parts-per-million) of the underlying root clock.

*This is a partial stub method. It must be re-implemented by root clocks.*

For a clock that is not the root clock, this method will pass through the call to the same method of the root clock.

New in version 0.4.

(Documentation inherited from *ClockBase*)

**isAvailable**()

Returns whether this clock is available, taking into account the availability of any (parent) clocks on which it depends.

**Returns** True if available, otherwise False.

New in version 0.4.

**nanos**

(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

**nanosToTicks**(*nanos*)

Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

**Parameters** **nanos** – nanoseconds value

**Returns** number of ticks equivalent to the supplied nanosecond value

**notify**(*cause*)

Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

**Parameters** **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

**setAvailability**(*availability*)

SysClock is always available, and so availability cannot be set.

**Raises** **NotImplementedError** – Always raised.

New in version 0.4.

**setParent**(*newParent*)

*This is a stub for this method. Sub-classes should implement it.*

Change the parent of this clock (if supported). Will generate a notification of change.

**speed**
(read/write `float`) The speed at which the clock is running. Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

**tickRate**
(read only)The tick rate (in ticks per second) of this clock.

(Documentation inherited from *ClockBase*)

**ticks**
(read only) The tick count for this clock.

(Documentation inherited from *ClockBase*)

**toOtherClockTicks**(*otherClock*, *ticks*)
Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters**

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

**Returns** The tick value of the *otherClock* that represents the same moment in time, or *Not a number (nan)*

**Throws NoCommonClock** if there is no common ancestor clock (meaning it is not possible to convert

**toParentTicks**(*ticks*)

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value of this clock converted to the timescale of the parent clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**toRootTicks**(*t*)
Return the time for the root clock corresponding to a given time of this clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters** **t** – Tick value for this clock.

**Returns** Corresponding tick value of the root clock, or *Not a number (nan)*

New in version 0.4.

**unbind**(*dependent*)
Unbind from notification if this clock changes.

**Parameters** **dependent** – The dependent to unbind from receiving notifications.

### CorrelatedClock - Clock correlated to another clock

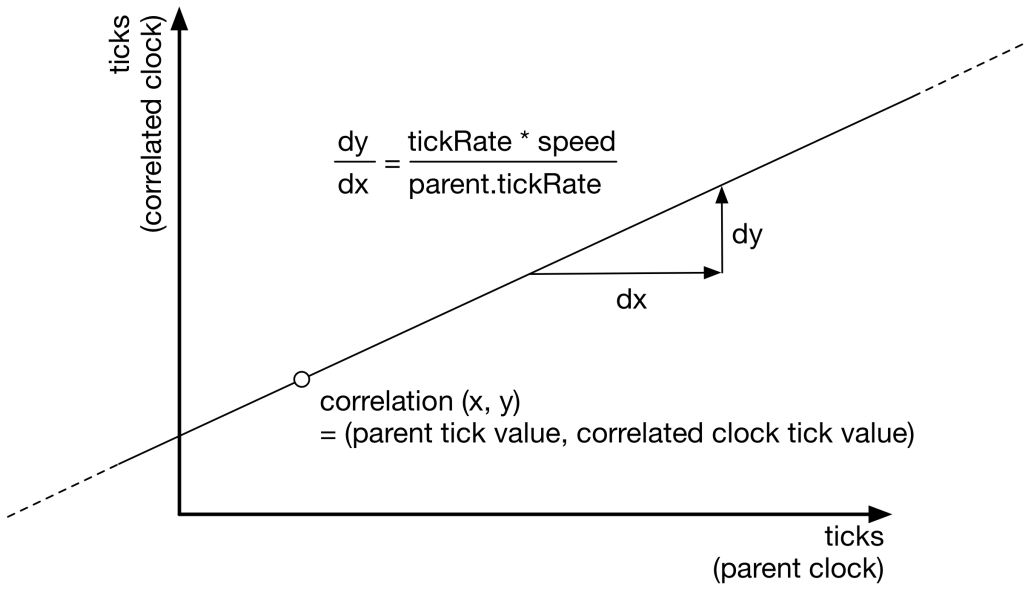
**class** dvbcss.clock.**CorrelatedClock**(*parentClock*, *tickRate*, *correlation=Correlation(parentTicks=0*, *childticks=0*, *initialError=0*, *errorGrowthRate=0)*, *speed=1.0*, ***kwargs*)

A clock locked to the tick count of the parent clock by a correlation and frequency setting.

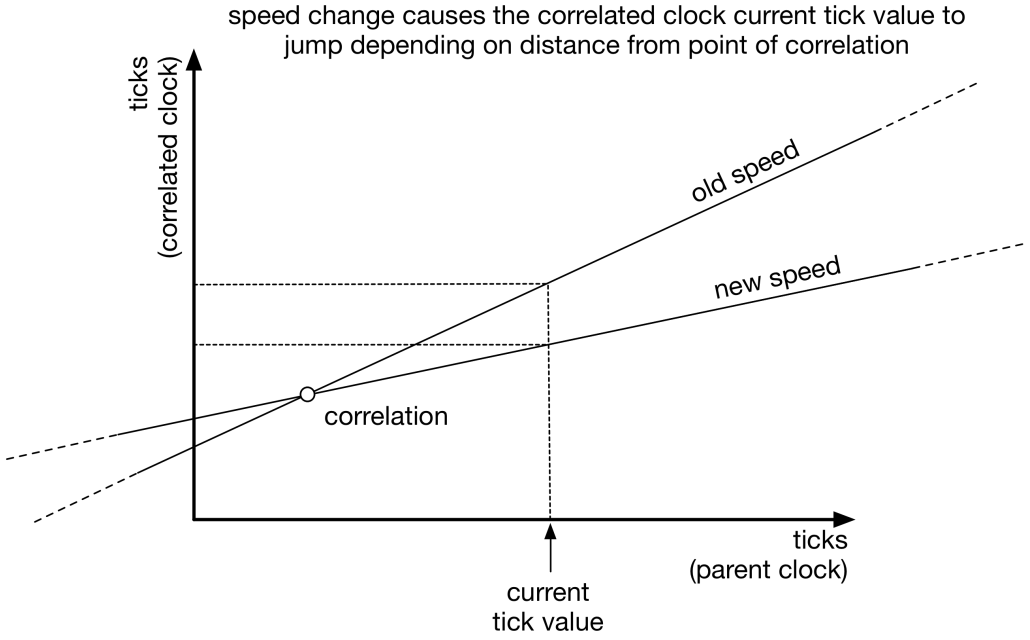
Correlation is either a `Correlation` object or a simple tuple *(parentTicks, selfTicks)*. The object form also allows you to specify error bounds information that this clock will track and propagate to other clocks.

When the parent clock ticks property has the value *parentTicks*, the ticks property of this clock shall have the value *selfTicks*.

This relationship can be illustrated as follows:




You can alter the correlation and tickRate and speed of this clock dynamically. Changes to tickRate and speed will not shift the point of correlation. This means that a change in tickRate or speed will probably cause the current tick value of the clock to jump. The amount it jumps by will be proportional to the distance the current time is from the point of correlation:




If you want a speed change to only affect the ticks from a particular point (e.g. the current tick value) onwards then you must re-base the correlation. There is a function provided to do that in some circumstances:

```
c = CorrelatedClock(parentClock=parent, tickRate=1000, correlation=(50,78))

   ... time passes ...

# now freeze the clock AT ITS CURRENT TICK VALUE

c.rebaseCorrelationAtTicks(c.ticks)
c.speed = 0

# now resume the clock but at half speed, but again without the tick value jumping
c.correlation = Correlation( parent.ticks, c.ticks )
c.speed = 0.5
```

**Note:** The maths to calculate and convert tick values will be performed, by default, as integer maths unless the parameters controlling the clock (tickRate etc) are floating point, or the ticks property of the parent clock supplies floating point values.

**Parameters**

- **parentClock** – The parent clock for this clock.
- **tickRate** – (int) tick rate for this clock (in ticks per second)
- **correlation** – (*Correlation*) or tuple *(parentTicks, selfTicks)*. The intial correlation for this clock.
- **speed** – Initial speed for this clock.

**bind**(*dependent*)
Bind for notification if this clock changes.

**Parameters** **dependent** – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

**calcWhen**(*ticksWhen*)

**Return** "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*). Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

(Documentation inherited from *ClockBase*)

**clockDiff**(*otherClock*)
Calculate the potential for difference between this clock and another clock.

**Parameters** **otherClock** – The clock to compare with.

**Returns** The potential difference in units of seconds. If effective speeds or tick rates differ, this will always be `float` ("+inf").

If the clocks differ in effective speed or tick rate, even slightly, then this means that the clocks will eventually diverge to infinity, and so the returned difference will equal +infinity.

New in version 0.4.

**correlation**
Read or change the correlation of this clock to its parent clock.

Assign a new *Correlation* object to change the correlation.

You can also pass a tuple *(parentTicks, selfTicks)* which will be converted to a correlation automatically. Reading this property after setting it with a tuple will return the equivalent *Correlation*.

**dispersionAtTime**(*t*)
Calculates the dispersion (maximum error bounds) at the specified clock time. This takes into account the contribution to error of this clock and its ancestors.

**Returns** Dispersion (in seconds) at the specified clock time.

New in version 0.4.

**fromParentTicks**(*ticks*)

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value for the parent clock converted to the timescale of this clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**fromRootTicks**(*t*)
Return the time for this clock corresponding to a given time of the root clock.

**Parameters** **t** – Tick value of the root clock.

**Returns** Corresponding tick value of this clock.

New in version 0.4.

**getAncestry**()
Retrieve the ancestry of this clock as a list.

**Returns** A list of clocks, starting with this clock, and proceeding to its parent, then its parent's parent etc, ending at the root clock.

New in version 0.4.

**getEffectiveSpeed**()
Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

**getParent**()

**returns** *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

(Documentation inherited from *ClockBase*)

**getRoot**()

**Returns** The root clock for this clock (or itself it has no parent).

New in version 0.4.

**getRootMaxFreqError**()
Return potential error of underlying clock (e.g. system clock).

**Returns** The maximum potential frequency error (in parts-per-million) of the underlying root clock.

*This is a partial stub method. It must be re-implemented by root clocks.*

For a clock that is not the root clock, this method will pass through the call to the same method of the root clock.

New in version 0.4.

**isAvailable**()
Returns whether this clock is available, taking into account the availability of any (parent) clocks on which it depends.

**Returns** True if available, otherwise False.

New in version 0.4.

**isChangeSignificant**(*newCorrelation*, *newSpeed*, *thresholdSecs*)
Returns True if the potential for difference in tick values of this clock (using a new correlation and speed) exceeds a specified threshold.

**Parameters**

- **newCorrelation** – A *Correlation*
- **newSpeed** – New speed as a `float`.

**Returns** True if the potential difference can/will eventually exceed the threshold.

This is implemented by applying a threshold to the output of *quantifyChange()*.

New in version 0.4.

**nanos**
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

**nanosToTicks**(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

**Parameters** **nanos** – nanoseconds value

**Returns** number of ticks equivalent to the supplied nanosecond value

**notify**(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

**Parameters** **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

**quantifyChange**(*newCorrelation*, *newSpeed*)
Calculate the potential for difference in tick values of this clock if a different correlation and speed were to be used.

**Parameters**

- **newCorrelation** – A *Correlation*
- **newSpeed** – New speed as a float.

**Returns** The potential difference in units of seconds. If speeds differ, this will always be float ("+inf").

If the new speed is different, even slightly, then this means that the ticks reported by this clock will eventually differ by infinity, and so the returned value will equal +infinity. If the speed is unchanged then the returned value reflects the difference between old and new correlations.

New in version 0.4.

**rebaseCorrelationAtTicks**(*tickValue*)
Changes the *correlation* property to an equivalent correlation (that does not change the timing relationship between parent clock and this clock) where the tick value for this clock is the provided tick value.

**setAvailability**(*availability*)
Set the availability of this clock.

**Parameters** **availability** – True if this clock is available, otherwise False.

If setting the availability of this clock changes the overall availability of this clock (as returned by *isAvailable()*) then dependents will be notified of the change.

New in version 0.4.

**setCorrelationAndSpeed**(*newCorrelation*, *newSpeed*)
Set both the correlation and the speed to new values in a single operation. Generates a single notification for descendents as a result.

**Parameters**

- **newCorrelation** – A *Correlation* representing the new correlation. Or a tuple *(parentTicks, selfTicks)* representing the correlation.
- **newSpeed** – New speed as a float.

New in version 0.4.

**setParent**(*newParent*)

Change the parent of this clock (if supported). Will generate a notification of change.

(Documentation inherited from *ClockBase*)

**speed**

**(read/write float) The speed at which the clock is running.** Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

(Documentation inherited from *ClockBase*)

**tickRate**

Read or change the tick rate (in ticks per second) of this clock. The value read is not affected by the value of the *speed* property.

**ticks**

(read only) The tick count for this clock.

(Documentation inherited from *ClockBase*)

**toOtherClockTicks**(*otherClock*, *ticks*)

Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters**

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

**Returns** The tick value of the *otherClock* that represents the same moment in time, or *Not a number (nan)*

**Throws NoCommonClock** if there is no common ancestor clock (meaning it is not possible to convert

**toParentTicks**(*ticks*)

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value of this clock converted to the timescale of the parent clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**toRootTicks**(*t*)

Return the time for the root clock corresponding to a given time of this clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters** **t** – Tick value for this clock.

**Returns** Corresponding tick value of the root clock, or *Not a number (nan)*

New in version 0.4.

**unbind**(*dependent*)
Unbind from notification if this clock changes.

**Parameters** **dependent** – The dependent to unbind from receiving notifications.

### OffsetClock - A clock that is offset by a fixed amount of root time

**class** dvbcss.clock.**OffsetClock**(*parentClock*, *offset=0*)
A clock that applies an offset such that reading it is the same as reading its parent, but as if the current time is slightly offset by an amount ahead (+ve offset) or behind (-ve offset).

*OffsetClock* inherits the tick rate of its parent. Its speed is always 1. It takes the effective speed into account when applying the offset, so it should always represent the same amount of time according to the root clock. In practice this means it will be a constant offset amount of real-world time.

This can be used to compensate for rendering delays. If it takes N seconds to render some content and display it, then a positive offset of N seconds will mean that the rendering code thinks time is N seconds ahead of where it is. It will then render the correct content that is needed to be displayed in N seconds time.

For example: A correlated clock (the "media clock") represents the time position a video player needs to currently be at.

The video player has a 40 milisecond (0.040 second) delay between when it renders a frame and the light being emitted by the display. We therefore need the video player to render 40 milliseconds in advance of when the frame is to be displayed. An *OffsetClock* is used to offset time in this way and is passed to the video player:

```
mediaClock = CorrelatedClock(...)

PLAYER_DELAY_SECS = 0.040
oClock = OffsetClock(parent=mediaClock, offset=PLAYER_DELAY_SECS)

videoPlayer.syncToClock(oClock)
```

If needed, the offset can be altered at runtime, by setting the *offset* property. For example, perhaps it needs to be changed to a 50 millisecond offset:

```
oClock.offset = 0.050
```

Both positive and negative offsets can be used.

**bind**(*dependent*)
Bind for notification if this clock changes.

**Parameters** **dependent** – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

**calcWhen**(*ticksWhen*)

**Return** "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*). Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

*This is a stub for this method. Sub-classes should implement it.*

**clockDiff**(*otherClock*)
Calculate the potential for difference between this clock and another clock.

**Parameters** **otherClock** – The clock to compare with.

**Returns** The potential difference in units of seconds. If effective speeds or tick rates differ, this will always be `float` ("+inf").

If the clocks differ in effective speed or tick rate, even slightly, then this means that the clocks will eventually diverge to infinity, and so the returned difference will equal +infinity.

New in version 0.4.

**dispersionAtTime**(*t*)

Calculates the dispersion (maximum error bounds) at the specified clock time. This takes into account the contribution to error of this clock and its ancestors.

**Returns** Dispersion (in seconds) at the specified clock time.

New in version 0.4.

**fromParentTicks**(*ticks*)

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value for the parent clock converted to the timescale of this clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**fromRootTicks**(*t*)

Return the time for this clock corresponding to a given time of the root clock.

**Parameters** **t** – Tick value of the root clock.

**Returns** Corresponding tick value of this clock.

New in version 0.4.

**getAncestry**()

Retrieve the ancestry of this clock as a list.

**Returns** A list of clocks, starting with this clock, and proceeding to its parent, then its parent's parent etc, ending at the root clock.

New in version 0.4.

**getEffectiveSpeed**()

Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

**getParent**()

**returns** *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

(Documentation inherited from *ClockBase*)

**getRoot**()

**Returns** The root clock for this clock (or itself it has no parent).

New in version 0.4.

**getRootMaxFreqError**()
Return potential error of underlying clock (e.g. system clock).

**Returns** The maximum potential frequency error (in parts-per-million) of the underlying root clock.

*This is a partial stub method. It must be re-implemented by root clocks.*

For a clock that is not the root clock, this method will pass through the call to the same method of the root clock.

New in version 0.4.

**isAvailable**()
Returns whether this clock is available, taking into account the availability of any (parent) clocks on which it depends.

**Returns** True if available, otherwise False.

New in version 0.4.

**nanos**
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

**nanosToTicks**(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

**Parameters** **nanos** – nanoseconds value

**Returns** number of ticks equivalent to the supplied nanosecond value

**notify**(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

**Parameters** **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

**offset**
Read or change the number of seconds by which this clock is ahead (the offset).

**setAvailability**(*availability*)
Set the availability of this clock.

**Parameters** **availability** – True if this clock is available, otherwise False.

If setting the availability of this clock changes the overall availability of this clock (as returned by *isAvailable()*) then dependents will be notified of the change.

New in version 0.4.

**setParent**(*newParent*)

Change the parent of this clock (if supported). Will generate a notification of change.

(Documentation inherited from *ClockBase*)

**speed**
Read the clock's speed. It is always 1.

**tickRate**
Read the tick rate (in ticks per second) of this clock. The tick rate is always that of the parent clock.

**ticks**
(read only) The tick count for this clock.

(Documentation inherited from *ClockBase*)

**toOtherClockTicks**(*otherClock*, *ticks*)
Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters**

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

**Returns** The tick value of the *otherClock* that represents the same moment in time, or *Not a number (nan)*

**Throws NoCommonClock** if there is no common ancestor clock (meaning it is not possible to convert

**toParentTicks**(*ticks*)

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value of this clock converted to the timescale of the parent clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**toRootTicks**(*t*)
Return the time for the root clock corresponding to a given time of this clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters** **t** – Tick value for this clock.

**Returns** Corresponding tick value of the root clock, or *Not a number (nan)*

New in version 0.4.

**unbind**(*dependent*)
Unbind from notification if this clock changes.

**Parameters** **dependent** – The dependent to unbind from receiving notifications.

### TunableClock - Clock with dynamically adjustable frequency and tick offset

**class** dvbcss.clock.**TunableClock**(*parentClock*, *tickRate*, *ticks=0*, ***kwargs*)
A clock whose tick offset and speed can be adjusted on the fly. Must be based on another clock.

Advancement of time of this clock is based on the tick count and rates reported by the supplied parent clock.

If you adjust the tickRate or speed, then the change is applied going forward from the moment it is made. E.g. if you are observing the rate of increase of the ticks property, then doubling the speed wil cause the ticks property to start increasing faster but will not cause it to suddenly jump value.

Changed in version 0.4: TunableClock has been reimplemented as a subclass of *CorrelatedClock*. The behaviour is the same as before, however it now also includes all the methods defined for *CorrelatedClock* too.

The maths to calculate and convert tick values will be performed, by default, as integer maths unless the parameters controlling the clock (tickRate etc) are floating point, or the ticks property of the parent clock supplies floating point values.

**Parameters**

- **parentClock** – The parent clock for this clock.
- **tickRate** – The tick rate (ticks per second) for this clock.
- **ticks** – The starting tick value for this clock.

The specified starting tick value applies from the moment this object is initialised.

**adjustTicks**(*offset*)
Change the tick count of this clock by the amount specified.

**bind**(*dependent*)
Bind for notification if this clock changes.

**Parameters** **dependent** – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

**calcWhen**(*ticksWhen*)

**Return** "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*). Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

(Documentation inherited from *ClockBase*)

**clockDiff**(*otherClock*)
Calculate the potential for difference between this clock and another clock.

**Parameters** **otherClock** – The clock to compare with.

**Returns** The potential difference in units of seconds. If effective speeds or tick rates differ, this will always be `float` ("+inf").

If the clocks differ in effective speed or tick rate, even slightly, then this means that the clocks will eventually diverge to infinity, and so the returned difference will equal +infinity.

New in version 0.4.

**correlation**
Read or change the correlation of this clock to its parent clock.

Assign a new *Correlation* object to change the correlation.

You can also pass a tuple *(parentTicks, selfTicks)* which will be converted to a correlation automatically. Reading this property after setting it with a tuple will return the equivalent *Correlation*.

**dispersionAtTime**(*t*)
Calculates the dispersion (maximum error bounds) at the specified clock time. This takes into account the contribution to error of this clock and its ancestors.

**Returns** Dispersion (in seconds) at the specified clock time.

New in version 0.4.

**fromParentTicks**(*ticks*)

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value for the parent clock converted to the timescale of this clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**fromRootTicks**(*t*)

Return the time for this clock corresponding to a given time of the root clock.

**Parameters** **t** – Tick value of the root clock.

**Returns** Corresponding tick value of this clock.

New in version 0.4.

**getAncestry**()

Retrieve the ancestry of this clock as a list.

**Returns** A list of clocks, starting with this clock, and proceeding to its parent, then its parent's parent etc, ending at the root clock.

New in version 0.4.

**getEffectiveSpeed**()

Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

**getParent**()

**returns** *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

(Documentation inherited from *ClockBase*)

(Documentation inherited from *CorrelatedClock*)

**getRoot**()

**Returns** The root clock for this clock (or itself it has no parent).

New in version 0.4.

**getRootMaxFreqError**()

Return potential error of underlying clock (e.g. system clock).

**Returns** The maximum potential frequency error (in parts-per-million) of the underlying root clock.

*This is a partial stub method. It must be re-implemented by root clocks.*

For a clock that is not the root clock, this method will pass through the call to the same method of the root clock.

New in version 0.4.

**isAvailable**()

Returns whether this clock is available, taking into account the availability of any (parent) clocks on which it depends.

**Returns** True if available, otherwise False.

New in version 0.4.

**isChangeSignificant**(*newCorrelation*, *newSpeed*, *thresholdSecs*)
Returns True if the potential for difference in tick values of this clock (using a new correlation and speed) exceeds a specified threshold.

**Parameters**

- **newCorrelation** – A *Correlation*
- **newSpeed** – New speed as a `float`.

**Returns** True if the potential difference can/will eventually exceed the threshold.

This is implemented by applying a threshold to the output of *quantifyChange()*.

New in version 0.4.

**nanos**
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

**nanosToTicks**(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

**Parameters** **nanos** – nanoseconds value

**Returns** number of ticks equivalent to the supplied nanosecond value

**notify**(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

**Parameters** **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

**quantifyChange**(*newCorrelation*, *newSpeed*)
Calculate the potential for difference in tick values of this clock if a different correlation and speed were to be used.

**Parameters**

- **newCorrelation** – A *Correlation*
- **newSpeed** – New speed as a `float`.

**Returns** The potential difference in units of seconds. If speeds differ, this will always be `float` ("+inf").

If the new speed is different, even slightly, then this means that the ticks reported by this clock will eventually differ by infinity, and so the returned value will equal +infinity. If the speed is unchanged then the returned value reflects the difference between old and new correlations.

New in version 0.4.

**rebaseCorrelationAtTicks**(*tickValue*)
Changes the *correlation* property to an equivalent correlation (that does not change the timing relationship between parent clock and this clock) where the tick value for this clock is the provided tick value.

**setAvailability**(*availability*)
Set the availability of this clock.

**Parameters** **availability** – True if this clock is available, otherwise False.

If setting the availability of this clock changes the overall availability of this clock (as returned by *isAvailable()*) then dependents will be notified of the change.

New in version 0.4.

**setCorrelationAndSpeed**(*newCorrelation*, *newSpeed*)
Set both the correlation and the speed to new values in a single operation. Generates a single notification for descendents as a result.

**Parameters**

- **newCorrelation** – A *Correlation* representing the new correlation. Or a tuple *(parentTicks, selfTicks)* representing the correlation.
- **newSpeed** – New speed as a float.

New in version 0.4.

**setError**(*current*, *growthRate=0*)
Set the current error bounds of this clock and the rate at which it grows per tick of the parent clock.

**Parameters**

- **current** – Potential error (in seconds) of the clock at this time.
- **growthRate** – Amount by which error will grow for every tick of the parent clock.

New in version 0.4.

**setParent**(*newParent*)

Change the parent of this clock (if supported). Will generate a notification of change.

(Documentation inherited from *ClockBase*)

**slew**
This is an alternative method of querying or adjusting the speed property.

The slew (in ticks per second) currently applied to this clock.

Setting this property will set the speed property to correspond to the specified slew.

For example: for a clock with tickRate of 100, then a slew of -25 corresponds to a speed of 0.75

**speed**

**(read/write float) The speed at which the clock is running.** Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

(Documentation inherited from *ClockBase*)

**tickRate**
Read or change the tick rate (in ticks per second) of this clock. The value read is not affected by the value of the *speed* property.

**ticks**
(read only) The tick count for this clock.

(Documentation inherited from *ClockBase*)

**toOtherClockTicks**(*otherClock*, *ticks*)
Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters**

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

**Returns** The tick value of the *otherClock* that represents the same moment in time, or *Not a number (nan)*

**Throws NoCommonClock** if there is no common ancestor clock (meaning it is not possible to convert

**toParentTicks**(*ticks*)

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value of this clock converted to the timescale of the parent clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**toRootTicks**(*t*)

Return the time for the root clock corresponding to a given time of this clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters** **t** – Tick value for this clock.

**Returns** Corresponding tick value of the root clock, or *Not a number (nan)*

New in version 0.4.

**unbind**(*dependent*)

Unbind from notification if this clock changes.

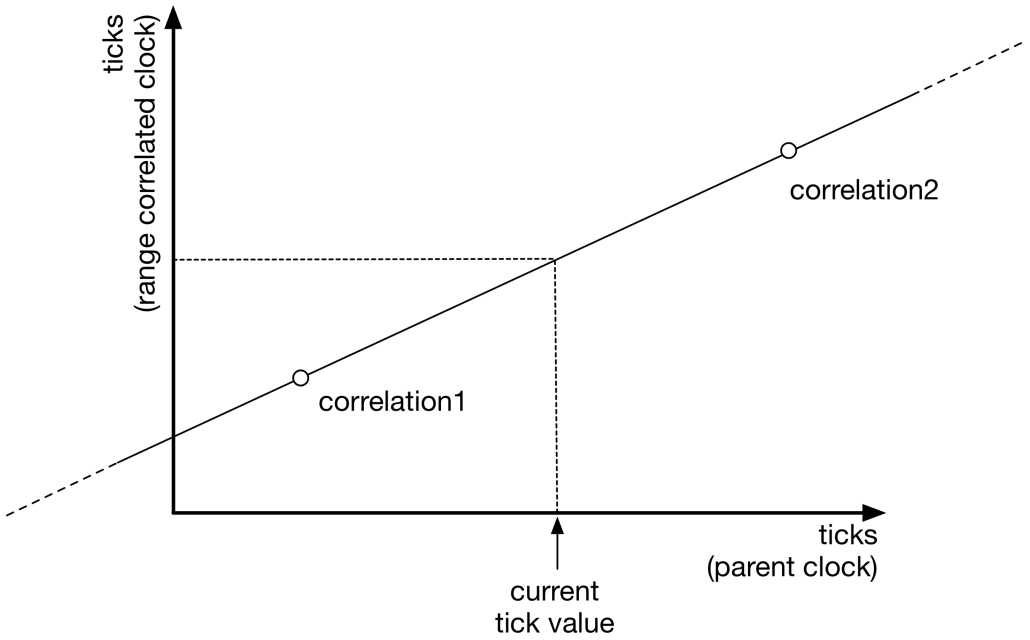
**Parameters** **dependent** – The dependent to unbind from receiving notifications.

### RangeCorrelatedClock - Clock correlated to another clock

**class** dvbcss.clock.**RangeCorrelatedClock**(*parentClock*, *tickRate*, *correlation1*, *correlation2*, ***kwargs*)

A clock locked to the tick count of the parent clock by two different points of correlation.

This relationship can be illustrated as follows:




The tickRate you set is purely advisory - it is the tickRate reported to clocks that use this clock as the parent, and may differ from what the reality of the two correlations represents!

**Parameters**

- **parentClock** – The parent clock for this clock.
- **tickRate** – The advisory tick rate (ticks per second) for this clock.
- **correlation1** – (*Correlation*) or tuple *(parentTicks, selfTicks)*. The first point of correlation for this clock.
- **correlation2** – (*Correlation*) or tuple *(parentTicks, selfTicks)*. The second point of correlation for this clock.

**bind**(*dependent*)

Bind for notification if this clock changes.

**Parameters** **dependent** – When this clock changes, the *notify()* method of this dependent will be called, passing a single argument that is this clock.

**calcWhen**(*ticksWhen*)

**Return** "when" in terms of the underlying clock behind the root clock implementation (e.g. `monotonic_time.time()` in the case of *SysClock*). Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

(Documentation inherited from *ClockBase*)

**clockDiff**(*otherClock*)

Calculate the potential for difference between this clock and another clock.

**Parameters** **otherClock** – The clock to compare with.

**Returns** The potential difference in units of seconds. If effective speeds or tick rates differ, this will always be `float` ("+inf").

If the clocks differ in effective speed or tick rate, even slightly, then this means that the clocks will eventually diverge to infinity, and so the returned difference will equal +infinity.

New in version 0.4.

**correlation1**
Read or change the first correlation of this clock to its parent clock.

Assign a new *Correlation* or tuple *(parentTicks, childTicks)* to change the correlation.

**correlation2**
Read or change the first correlation of this clock to its parent clock.

Assign a new *Correlation* to change the correlation.

**dispersionAtTime**(*t*)
Calculates the dispersion (maximum error bounds) at the specified clock time. This takes into account the contribution to error of this clock and its ancestors.

**Returns** Dispersion (in seconds) at the specified clock time.

New in version 0.4.

**fromParentTicks**(*ticks*)

Method to convert from a tick value for this clock's parent to the equivalent tick value (representing the same point in time) for this clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value for the parent clock converted to the timescale of this clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**fromRootTicks**(*t*)
Return the time for this clock corresponding to a given time of the root clock.

**Parameters** **t** – Tick value of the root clock.

**Returns** Corresponding tick value of this clock.

New in version 0.4.

**getAncestry**()
Retrieve the ancestry of this clock as a list.

**Returns** A list of clocks, starting with this clock, and proceeding to its parent, then its parent's parent etc, ending at the root clock.

New in version 0.4.

**getEffectiveSpeed**()
Returns the 'effective speed' of this clock.

This is equal to multiplying together the speed properties of this clock and all of the parents up to the root clock.

**getParent**()

**returns** *ClockBase* representing the immediate parent of this clock, or None if it is a root clock.

(Documentation inherited from *ClockBase*)

**getRoot**()

**Returns** The root clock for this clock (or itself it has no parent).

New in version 0.4.

**getRootMaxFreqError**()
Return potential error of underlying clock (e.g. system clock).

**Returns** The maximum potential frequency error (in parts-per-million) of the underlying root clock.

*This is a partial stub method. It must be re-implemented by root clocks.*

For a clock that is not the root clock, this method will pass through the call to the same method of the root clock.

New in version 0.4.

**isAvailable**()
Returns whether this clock is available, taking into account the availability of any (parent) clocks on which it depends.

**Returns** True if available, otherwise False.

New in version 0.4.

**nanos**
(read only) The tick count of this clock, but converted to units of nanoseconds, based on the current tick rate (but ignoring the *speed* property).

**nanosToTicks**(*nanos*)
Convert the supplied nanosecond to number of ticks given the current tick rate of this clock (but ignoring the *speed* property).

**Parameters** **nanos** – nanoseconds value

**Returns** number of ticks equivalent to the supplied nanosecond value

**notify**(*cause*)
Call to notify this clock that the clock on which it is based (its parent) has changed relative to the underlying timing source.

**Parameters** **cause** – The clock that is calling this method.

Will notify all dependents of this clock (entities that have registered themselves by calling *bind()*).

**setAvailability**(*availability*)
Set the availability of this clock.

**Parameters** **availability** – True if this clock is available, otherwise False.

If setting the availability of this clock changes the overall availability of this clock (as returned by *isAvailable()*) then dependents will be notified of the change.

New in version 0.4.

**setParent**(*newParent*)

Change the parent of this clock (if supported). Will generate a notification of change.

(Documentation inherited from *ClockBase*)

**speed**

**(read/write float) The speed at which the clock is running.** Does not change the reported tickRate value, but will affect how ticks are calculated from parent clock ticks. Default = 1.0 = normal tick rate.

(Documentation inherited from *ClockBase*)

**tickRate**
Read the tick rate (in ticks per second) of this clock. The value read is not affected by the value of the *speed* property.

**ticks**
(read only) The tick count for this clock.

(Documentation inherited from *ClockBase*)

**toOtherClockTicks**(*otherClock*, *ticks*)
Converts a tick value for this clock into a tick value corresponding to the timescale of another clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters**

- **otherClock** – A *clock* object representing another clock.
- **ticks** – A time (tick value) for this clock

**Returns** The tick value of the *otherClock* that represents the same moment in time, or *Not a number (nan)*

**Throws NoCommonClock** if there is no common ancestor clock (meaning it is not possible to convert

**toParentTicks**(*ticks*)

Method to convert from a tick value for this clock to the equivalent tick value (representing the same point in time) for the parent clock.

Implementations should use the parent clock's *tickRate* and *speed* properties when performing the conversion.

**returns** The specified tick value of this clock converted to the timescale of the parent clock.

**throws StopIteration** if this clock has no parent

(Documentation inherited from *ClockBase*)

**toRootTicks**(*t*)
Return the time for the root clock corresponding to a given time of this clock.

Will return *Not a number (nan)* if the conversion is not possible (e.g. when current *speed* is zero and *ticksWhen* does not match the current *correlation.childTicks*).

**Parameters** **t** – Tick value for this clock.

**Returns** Corresponding tick value of the root clock, or *Not a number (nan)*

New in version 0.4.

**unbind**(*dependent*)
Unbind from notification if this clock changes.

**Parameters** **dependent** – The dependent to unbind from receiving notifications.

### Correlation - represents a Correlation

**class** dvbcss.clock.**Correlation**(*parentTicks*, *childTicks*, *initialError=0*, *errorGrowthRate=0*)

Immutable object representing a correlation. This can also optionally include bounds for the potential for error if the correlation is used to model a clock.

The correlation (*parentTicks*, *ticks*) represents a relationship between a (child) clock and its parent. The time *parentTicks* of the parent corresponds to the time *ticks* of the child.

**Parameters**

- **parentTicks** – Time of the parent clock.
- **childTcks** – Corresponding time of the clock using this correlation.
- **initialError** – Optional (default=0). The amount of potential error (in seconds) at the moment represented by the correlation.
- **errorGrowthRate** – Optional (default=0). The rate of growth of error (e.g. how many seconds error increases by per second of the parent clock)

This class is intended to be immutable. Instead of modifying a correlation, create a new one based on an existing one. The *butWith()* method is designed to assist with this.

..note:

```
For backwards compatibility with pydvbcss v0.3 and earlier, this object can
also be treated as if it is a tuple (parentTicks, childTicks), for example:

..code-block:: python

    c = Correlation(1,5, 0.1, 0.005)

    # unapacking using an expression assignment
    parentTicks, childTicks = c

    # accessing by index
    parentTicks = c[0]
    childTicks = c[1]
```

New in version 0.4.

**butWith**(*parentTicks=None*, *childTicks=None*, *initialError=None*, *errorGrowthRate=None*)

Return a new correlation the same as this one but with the specified changes.

**Parameters**

- **parentTicks** – Optional. A new Time of the parent clock.
- **childTcks** – Optional. The corresponding time of the clock using this correlation.
- **initialError** – Optional. The amount of potential error (in seconds) at the moment represented by the correlation.
- **errorGrowthRate** – Optional. The rate of growth of error (e.g. how many seconds error increases by per second of the parent clock)

**Returns** New *Correlation* based on this one, but with the changes specified by the parameters.

If a parameter is set to *None* or not provided, then the existing value is taken from this correlation object.

**childTicks**

Number representing a time of the child clock, that corresponds to the *parentTicks* time of the parent clock.

**errorGrowthRate**
Number representing the amount that the potential for error will grow by (in units of seconds) for every tick of the parent clock. Default value is 0 if not set.

**initialError**
Number representing the amount of potential error (in units of seconds) at the moment represented by the correlation. Default value is 0 if not set.

**parentTicks**
Number representing a time of the parent clock.

## 3.3 Task scheduling for clocks

Module: *dvbcss.task*



### 3.3.1 Introduction

The *dvbcss.task* module provides sleep and scheduling functions for use with the *dvbcss.clock* module. These functions track adjustments to clocks (such as changes in the tick rate or tick value/offset) to ensure that the sleep or scheduled event happen when the clock actually reaches the target tick count value.

To use this module, just import it and directly call the functions *sleepFor()*, *sleepUntil()*, *scheduleEvent()* or *runAt()*.

---

**Note:** Scheduling happens on a single thread, so if you use the *runAt()* function, try to keep the callback code as fast and simple as possible, so that it returns control as quickly as possible.

---

See *How the dvbcss.task module works internally* for information on how the internals of the Task module work.

### 3.3.2 Example

A simple example:

```
from dvbcss.clock import SysClock
from dvbcss.clock import CorrelatedClock
from dvbcss.task import sleepFor, runAt

s = SysClock()
c = CorrelatedClock(parentClock=s, tickRate=1000)

# wait 1 second
sleepFor(c, numTicks=1000)

# schedule callback in 5 seconds
def foo(message):
```

```
    print "Callback!", message

runAt(clock=c, whenTicks=c.ticks+5000, foo, "Tick count progressed by 5 seconds")

# ... but change the correlation to make the clock jump 1 second forward
#     causing the callback to happen one second earlier
c.correlation = (c.correlation[0], c.correlation[1] + 1000)

# ... the callback will now happen in 4 seconds time instead
```

### 3.3.3 Functions

dvbcss.task.**sleepUntil**(*clock*, *whenTicks*)
Sleep until the specified *clock* reaches the specified tick value.

**Parameters**

- **clock** – (*dvbcss.clock.ClockBase*) Clock to sleep against the ticks of.
- **whenTicks** – (int) The tick value of the clock at which this function returns.

Returns after the specified tick value is reached.

dvbcss.task.**sleepFor**(*clock*, *numTicks*)
Sleep for the number of ticks of the specified clock.

**Parameters**

- **clock** – (*dvbcss.clock.ClockBase*) Clock to sleep against the ticks of.
- **numTicks** – (int) The number of ticks to sleep for.

Returns after the elapsed number of ticks of the specified clock have passed.

dvbcss.task.**scheduleEvent**(*clock*, *whenTicks*, *event*)
Schedule the threading.Event to be called when the specified clock reaches (or passes) the specified tick value.

**Parameters**

- **clock** – (*dvbcss.clock.ClockBase*) Clock to schedule the event against
- **whenTicks** – (int) The tick value of the clock at which the event is to be triggered.
- **event** – (threading.Event) python Event object that the :method:threading.Event.set method will be called on at the scheduled time

dvbcss.task.**runAt**(*clock*, *whenTicks*, *callBack*, *args=None*, *kwargs=None*)
Call the specified callback function when the specified clock reaches (or passes) the specified tick value.

The callback happens on the single thread used within the clock scheduling system. You should avoid writing code that hogs this thread to do substantial processing.

**Parameters**

- **clock** – (*dvbcss.clock.ClockBase*) Clock to schedule the callback against
- **whenTicks** – (int) The tick value of the clock at which the callback is to be called.
- **callback** – (callable) Function to be called
- **args** – A list of positional arguments to be passed to the callback function when it is called.

- **kwargs** – A `dict` of keyword arguments to be pased to the callback function when it is called.

The *dvbcss.monotonic_time* module provides a *time()* and *sleep()* functions equivalent to those in the standard python library `time` module. However these are guaranteeed to be monotonic and use the highest precision time sourcecs available (depending on the host operating system).

The *dvbcss.clock* module provides high level abstractions for representing clocks and timelines and the relationships between them. The *client and server implementations* for the DVB-CSS protocols use these objects to represent clocks and timelines.

The:mod:*Task* module provides sleep and task scheduling functions that work with *clock* objects and allow code to be called when a clock reaches a particular tick value, even if that clock is adjusted in some way after the task is scheduled.

CHAPTER 4

# Internal implementation details

## 4.1 Protocol server implementation details



### 4.1.1 CSS-WC

#### Overview

The CSS-WC server is based on a simple generic framework for building UDP servers

#### Classes

**class** dvbcss.protocol.server.wc.**UdpRequestServer**(*socket*, *handler*, *maxMsgSize*)

Simple request handling server framework.

Pass it an open UDP socket object that blocks on receive, and it will call your handler, passing the socket, plus the received data.

This all happens in a separate thread.

Use start() and stop() methods to start and stop the handling thread.

Socket must have set blocking and a timeout.

**Parameters**

- **socket** (socket.socket) – Bound socket ready to receive (and send) UDP packets
- **handler** – (object) Object providing a *handle()* method.
- **maxMsgSize** (*int*) – The maximum message size (sets the UDP receive buffer size)

The *handler* object must have a method with the following signature:

**handle**(*socket*, *received_data*, *src_addr*)

**Parameters**

- **socket** – The socket object for the connection on which the packet was received.
- **received_data** (*str*) – The received UDP packet payload
- **src_addr** – The source address. For an AF_INET connection this will be a tuple (str host, int port)

**run**()

Internal method - the main runloop of the thread.

Runs in a loop calling the *handle()* method of the object assigned to the handler property whenever a UDP packet is received.

Does not return until the _pleaseStop attribute of the object has been set to True

**start**()

Starts the wall clock server running. It runs in a thread in the background.

**stop**()

Stops the wall clock server running. Does not return until the thread has terminated.

**class** dvbcss.protocol.server.wc.**WallClockServerHandler**(*wallClock*, *precisionSecs=None*, *maxFreqErrorPpm=None*, *followup=False*, ***kwargs*)

Simple Wall Clock Server Handler function.

Provides a handle() method. Designed to be used with *UdpRequestServer*

**Parameters**

- **wallClock** (dvbcss.clock.ClockBase) – The clock to be used as the wall clock for protocol interactions
- **precisionSecs** – (float) Optional. Override using the precision of the provided clock and instead use this value. It is the precision (in seconds) to be reported for the clock in protocol interactions
- **maxFreqErrorPpm** – (float or None) Optional. Override using the *getRootMaxFreqError()* of the clock and instead use this value. It is the clock maximum frequency error in parts-per-million
- **followup** (*bool*) – Set to True if the Wall Clock Server should send follow-up responses

### 4.1.2 CSS-CII and CSS-TS

Module: *dvbcss.protocol.server*

### Overview

The CSS-CII and CSS-TS servers subclass the WebSocket server functionality for cherrypy implemented by ws4py in the `cherrypyserver` module.

*`CIIServer`* and *`TSServer`* both inherit from a common base implementation *`WSServerTool`* provided in the *dvbcss.protocol.server* module.

The Tool provides the hook into cherrypy for handling the connection request and upgrading it to a WebSocket connection, spawning an object representing the WebSocket connection and which implements the WebSocket protocols.

The base server object class is intended to manage all WebSocket connections for a particular server endpoint. It therefore provides its own customised WebScoket class that is bound to that particular server object instance.

The tool is enabled via an "on" configuration when setting up the mount point in cherrypy. The tool also expects to a "handler_cls" property set in the configuration at the mount point. This property points to a WebSocket class which can be instantiated to handle the connection.

Example usage: creating a server at "ws://<host>:80/endpoint" just using the base classes provided here:

```
import cherrypy
from ws4py.server.cherrypyserver import WebSocketPlugin
from dvbcss.protocol.server import WSServerBase, WSServerTool

# plug the tool into cherrypy as "my_server"
cherrypy.tools.my_server = WSServerTool()

WebSocketPlugin(cherrypy.engine).subscribe()

# create my server
myServer = WSServerBase()

# bind it to the URL path /endpoint in the cherrypy server
class Root(object):
    @cherrypy.expose
    def endpoint(self):
        pass

cfg = {"/endpoint": {'tools.my_server.on': True,
                'tools.my_server.handler_cls': myServer.handler
               }
      }

cherrypy.tree.mount(Root(), "/", config=cfg)

# activate cherrypy web server on port 80
cherrypy.config.update({"server.socket_port":80})
cherrypy.engine.start()
```

See documentation for *`WSServerBase`* for information on creating subclasses to implement specific endpoints.

### Classes

**class** dvbcss.protocol.server.**ConnectionIdGen**

Object for generating unique connection id strings.

**classmethod next**()

**Returns** new unique connection id string

**class** dvbcss.protocol.server.**WSServerTool**

Subclass of the ws4py.server.cherrypyserver.WebSocketTool tool to do a simultaneous connections limit check and enabled/disabled check for the end point before the handler is invoked

**cleanup_headers**()

Some clients aren't that smart when it comes to headers lookup.

**complete**()

Override of the base implementation of the code to install a callback to notify the websocket once the upgrade is complete.

Cherrypy's normal "completion" callbacks fire /before/ cherrypy has flushed outgoing data to the socket connection, creating a race-condition between the flush happening and the websocket class starting to write outgoing messages.

This replacement handler is guaranteed to fire /after/ cherrypy has flushed and is no longer using the socket connection.

**start_handler**()

Runs at the end of the request processing by calling the opened method of the handler.

**upgrade**(**args*, ***kwargs*)

Override of base implementation of the code to handle a connection "upgrade" (part of the WebSocket handshake).

The "handler_cls" for this tool is passed as an argument and it implements methods to check whether:

1. The end point is enabled. If not then a 403 "Forbidden" response is returned.
2. A connection can be allocated (the connection limit is not yet reached). If not then a 503 "Service Unavailable" response is returned.

If neither of these tests fail, then the superclass upgrade operation is allowed to proceed.

**class** dvbcss.protocol.server.**WSServerBase**(*maxConnectionsAllowed=-1*, *enabled=True*)

Base class for WebSocket server endpoint implementation.

**Parameters**

- **maxConnectionsAllowed** (*int*) – -1 to allow unlimited connections, otherwise sets the maximum number of concurrent connections from clients.
- **enabled** (*bool*) – Whether this server starts off enabled or disabled.

When the server is "disabled" it will refuse attempts to connect by sending the HTTP status response 403 "Forbidden".

When the server has reached its connection limit, it will refuse attempts to connect by sending the HTTP status response 503 "Service unavailable".

Protocol specific implementations inherit from this base class and override stub methods and class attributes. When subclassing you will want to override:

- *getDefaultConnectionData()*
- *onClientConnect()*
- *onClientDisconnect()*
- *onClientMessage()*

The WebSocket connection object representing each connection is used as a handle for the connection.

This class stores per connection connection data. What that data is is entirely up to the subclass. The *getDefaultConnectionData()* method provides the initial data for a connection when it is opened. The *_connections* instance variable then keeps a mapping from websocket connections to that data.

**handler**
Handler class for new connections. Should be provided as a configuration argument to cherrypy.

**_connections**
dict mapping WebSocket objects to connection data. Connection data is for use by subclasses to store data specific to each individual connection.

**_addConnection**(*webSock*)
Internal method. Called to notify this class of a newly connected websocket.

**Parameters** **webSock** – (WebSocket) The newly connected Websocket.

**_makeHandlerClass**(*connectionIdPrefix*)

**Parameters** **connectionIdPrefix** – (str) Human-readable prefix to be put on connection-ids

**Returns** *WebSocketHandler* class, which is a subclass of ws4py.websocket.WebSocket that is unique to this server instance.

**_receivedMessage**(*webSock*, *message*)
Internal method. Called to notify this class of a websocket message arrival.

**Parameters**

- **webSock** – (WebSocket) The WebSocket connection on which the message arrived.
- **msg** – (Message) WebSocket message that has been received. Will be either a Text or a Binary message.

**_removeConnection**(*webSock*)
Internal method. Called to notify this class of a websocket disconnection.

**Parameters** **webSock** – (WebSocket) The now disconnected Websocket.

**connectionIdPrefix = 'serverbase'**
prefix to be used for connection ids

**enabled**
(read/write bool) Whether this server endpoint is enabled (True) or disabled (False).

Set this property enable or disable the endpoint.

When disabled, existing connections are closed with WebSocket closure reason code 1001 and new attempts to connect will be refused with HTTP response status code 403 "Forbidden".

**getConnections**()

**Returns dict** mapping a WebSocket object to connection related data for all connections to this server. This is a snapshot of the connections at the moment the call is made. The dictionary is not updated later if new clients connect or existing ones disconnect.

**getDefaultConnectionData**()
This function is called to create new server-specific connection data when a new client connects. This is stored against the websocket connection and can be retrieved using the *getConnections()* method.

Override with your own function that returns data to be stored against new connections :returns: empty dict, but can return anything you like.

**loggingName = 'dvb-css.protocol.server.WSServerBase'**
name used for logging messages

**onClientConnect**(*webSock*)
This method is called when a new client connects.

*This is a stub for this method. Sub-classes should implement it.*

:param webSock:(`WebSocket`) The object representing the WebSocket connection of the newly connected client

**onClientDisconnect**(*webSock*, *connectionData*)
This method is called after a client is disconnected.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters**

- **webSock** – (`WebSocket`) The object representing the WebSocket connection of the now-disconnected client
- **connectionData** – (`dict`) of connection data relating to this connection.

**onClientMessage**(*webSock*, *msg*)
This method is called when a message is received from a client.

*This is a stub for this method. Sub-classes should implement it.*

**Parameters**

- **webSock** – (`WebSocket`) The object representing the WebSocket connection from which the message has been received.
- **msg** – (`Message`) WebSocket message that has been received. Will be either a `Text` or a `Binary` message.

**class .WebSocketHandler**(*WebSocket*)
This class is created and returned by the *`WSServerBase._makeHandlerClass()`* method and each class returned is bound to the instance of *`WSServerBase`* that created it.

It is intended to be provided to cherrypy as the "handler_cls" configuration parameter for the WebSocket tool. It is instantiated for every connection made.

These are subclasses of the ws4py `WebSocket` class and represent an individual WebSocket connection.

Instances of this class call through to *`WSServerBase._addConnection()`* and *`WSServerBase._removeConnection()`* and *`WSServerBase._receivedMessage()`* to inform the parent server of the WebSocket opening, closing and receiving messages.

**classmethod isEnabled**(*cls*)

**Returns** True if the server endpoint is enabled, otherwise False.

**classmethod canAllocateConnection**(*cls*)

**Returns** True only if the connection limit of the parent server has not yet been reached. Otherwise False.

**id**(*self*)

**Returns** A human readable connection ID

## 4.2 How the dvbcss.task module works internally

### 4.2.1 Introduction

The *dvbcss.task* module internally implements a task scheduler based around a single daemon thread with an internal priority queue.

Sleep and callback methods cause a task objet to be queued. The scheduler picks up the queued task and adds it to the priority queue and binds to the Clock so that it is notified of adjustments to the clock. When a task is added to the queue, the clock is queried to calculate the true time at which the tick count is expected to be reached by calling *dvbcss.clock.ClockBase.calcWhen()*

If a clock is adjusted the affected tasks are marked as deprecated (but remain in the priority queue) and new tasks are rescheduled with a recalculated time.

When one of the clocks involved has speed 0, then it may not be possible to calculate the time at which the task is to be scheduled. This happens when *dvbcss.clock.ClockBase.calcWhen()* returns *Not a number (nan)*. The task will not be immediately added to the priority queue, however it will be added later once the clock speed returns to a non-zero value. This happens automatically as part of the rescheduling process when a clock is adjusted.

### 4.2.2 Objects

dvbcss.task.**scheduler = <dvbcss.task._Scheduler object>**

Task scheduler. Starts an internal `threading.Thread` with `theading.Thread.daemon` set to True.

This is an internal of the Task module. For normal use you should not need to access it.

**Variables**

- **taskheap** – the priority queue of tasks
- **addQueue** – threadsafe queue of tasks to be added to the priority queue
- **rescheduleQueue** – thereadsafe queue of clocks that have been adjusted and therefore which need to trigger rescheduling of tasks
- **updateEvent** – `theading.Event` used to wake the scheduler thread whenever there is work pending (items added to addQueue or rescheduleQueue)
- **clock_Tasks** – mapping of clocks to takss that depend on them

Running instance of the *dvbcss.task._Scheduler*

### 4.2.3 Classes

**class** dvbcss.task.**_Scheduler**(**args*, ***kwargs*)

Task scheduler. Starts an internal `threading.Thread` with `theading.Thread.daemon` set to True.

This is an internal of the Task module. For normal use you should not need to access it.

**Variables**

- **taskheap** – the priority queue of tasks
- **addQueue** – threadsafe queue of tasks to be added to the priority queue
- **rescheduleQueue** – thereadsafe queue of clocks that have been adjusted and therefore which need to trigger rescheduling of tasks

- **updateEvent** – `theading.Event` used to wake the scheduler thread whenever there is work pending (items added to addQueue or rescheduleQueue)
- **clock_Tasks** – mapping of clocks to takss that depend on them

Starts the scheduler thread at initialisation.

**notify**(*causeClock*)
Callback entry point for when a clock is adjusted

**Parameters** **causeClock** – (*dvbcss.clock.ClockBase*) The clock that was adjusted and is therefore causing this notification of adjustment.

**run**()
Main runloop of the scheduler.

While looping:

1. Checks the queue of tasks to be added to the scheduler

   The time the task is due to be executed is calculated and used as the sort key when the task is inserted into a priority queue.

2. Checks any queued requests to reschedule tasks (due to clock adjustments)

   The existing task in the scheduler priority queue is "deprecated" And a new task is scheduled with the revised time of execution

3. checks any tasks that need to now be executed

   Dequeues them and executes them, or ignores them if they are marked as deprecated

**schedule**(*clock*, *whenTicks*, *callBack*, *args*, *kwargs*)
Queue up a task for scheduling

**Parameters**

- **clock** – (*dvbcss.clock.ClockBase*) the clock against which the task is scheduled
- **whenTicks** – (int) The tick value of the clock at which the scheduled task is to be executed
- **callback** – (func) The function (the task) that will be called at the scheduled time
- **args** – (list) List of arguments to be passed to the function when it is invoked
- **kwargs** – (dict) Dictionary of keyword arguments to be passed to the function when it is invoked

**stop**()
Stops the scheduler if it is running.

**class** dvbcss.task.**_Task**(*clock*, *whenTicks*, *callBack*, *args*, *kwargs*, *n=0*)
Representation of a scheduled task. This is an internal of the Task module. For normal use you should not need to acess it.

Initialiser

**Parameters**

- **clock** – (*dvbcss.clock.ClockBase*) the clock against which the task is scheduled
- **whenTicks** – (int) The tick value of the clock at which the scheduled task is to be executed
- **callback** – (func) The function (the task) that will be called at the scheduled time

- **args** – (list) List of arguments to be passed to the function when it is invoked
- **kwargs** – (dict) Dictionary of keyword arguments to be passed to the function when it is invoked
- **n** – (int) Generation count. Incremented whenever the task is based on a previous task (i.e. it is a rescheduled task)

**regenerateAndDeprecate**()
Sets the deleted flag of this task to True, and returns a new task the same as this one but not deleted and with the scheduled time 'when' recalculated from the clock

Here are some details on parts of the internal implementation of aspects of this library.

- modindex | Full Index

This collection of Python modules provides clients and servers for the network protocols defined in the DVB "Companion Screens and Streams" (CSS) specification ETSI 103 286 part 2. There are also supporting classes that model clocks (e.g. to represent timelines) and their inter-relationships.

Use it to build clients and servers for each of the protocols (CSS-WC, CSS-CII and CSS-TS) that mock or simulate the roles of TV Devices and Companion Screen Applications for testing and prototyping.

*To use this library you need to have a working understanding of these protocols and, of course, the Python programming language.*

CHAPTER 5

# Getting started

1. Install, following the instructions in the README.
2. Try to *Run the examples*.
3. Read the docs for the *DVB CSS Protocol modules*.
4. Use the library in you own code

CHAPTER 6

# State of implementation

This library does not currently implement the *CSS-TE* or *CSS-MRS* protocols (from the DVB specification).

There are some unit tests but these mainly only cover the calculations done within clock objects and the packing and unpacking of JSON messages.

CHAPTER 7

# Upgrading from previous versions

See the release notes / change log for details of what is new in this version of pydvbcss.

CHAPTER 8

# License and Contributing

*pydvbcss* is licensed as open source software under the terms of the Apache License v2.0.

See the *CONTRIBUTING* and *AUTHORS* files for information on how to contribute and who has contributed to this library.

CHAPTER 9

# Contact and discuss

There is a pydvbcss google group for announcements and discussion of this library.

# Python Module Index

## d



## e

# Index

## Symbols



## A



## B



## C

## D



## E

## F



## G

## H



## I



## L



## M



## N



## O

## P



## Q



## R

## S



## T

## U

## W