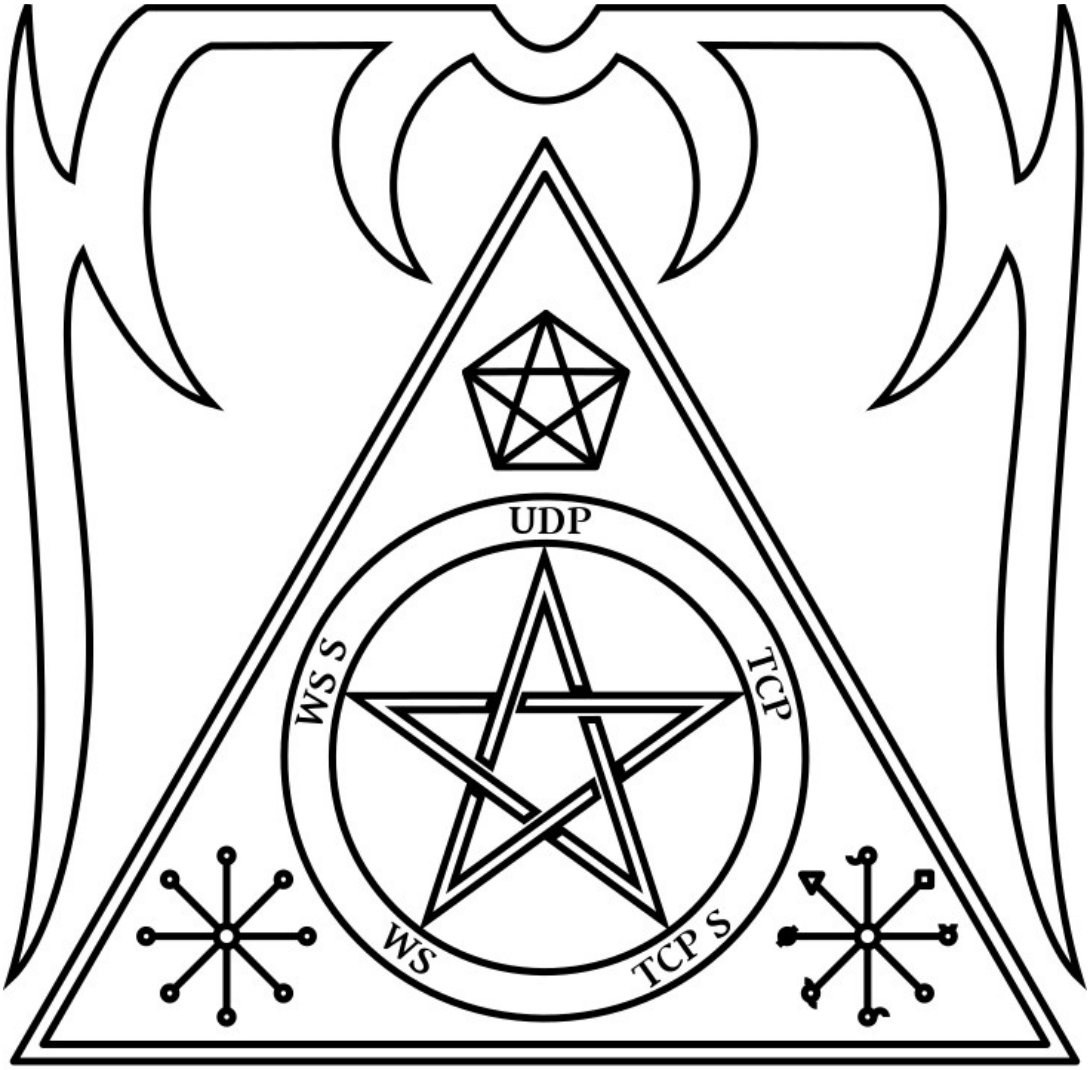

Sockets Under Control

Advanced documentation

eToile

(eToile 2020) V: 1.4

Index

Introduction

Welcome to the Sockets Under Control advanced documentation.

This documentation describes the core classes:

1. UDPConnection.cs
2. TCPConnection.cs
3. TCPServer.cs
4. WSConnection.cs
5. WSServer.cs (Not compatible with Unity/Mono)

Sockets Under Control is fully multi-threaded, includes simple Unity wrappers (easy to set in editor) and is ready to operate under IPV4 and IPV6 networks simultaneously in most cases.

Asset Integration

The first step is to import the package from the Asset Store. All Sockets Under Control content is into the "eToile" folder.

The core classes are into the eToile/SocketsUnderControl/CoreClasses folder, copy the cs files to any `.net` compatible platform to use them.

UDPConnection

Here is the complete description of the `UDPConnection` class and how to use it.
You can modify the parameters of this entity dynamically by calling `Setup()`. Once `Setup()` is called, the connection will be closed and must be reconnected again by calling `Connect()`.
All interaction with this entity is done through the 4 events.

UDPConnection methods

The methods are provided to keep control of the entity.

```
Constructor (new UDPConnection())
public UDPConnection(int port, string localIP = "", EventVoid evOpen = null, EventMessage
evMessage = null, EventException evError = null, EventVoid evClose = null)
```

The constructor allows to create the entity and set its parameters all at once, calling `Setup()` internally.
The `port` accepts values between 1 and 65535.
The `localIP` parameter binds the entity to a particular local network adapter.
It accepts the following values:

1. Empty: If an empty `string` is provided (`""`), the entity will automatically bind to the fist default IPV4 and IPV6 addresses simultaneously (this mode is enough in most cases).
2. IPV4: Binds to provided local IPV4 address, this IP must exist in the system or will fail. If a IPV4 address is provided, the simultaneity with IPV6 is not possible.
3. IPV6: Binds to provided local IPV6 address, this IP must exist in the system or will fail. If a IPV6 address is provided, the simultaneity with IPV4 is not possible.
4. URL: Automatically gets the related IP through the available DNS and binds to.
5. `"ipv4"` literal: Binds to the first available IPV4 address and discards the simultaneity with IPV6.
6. `"ipv6"` literal: Binds to the first available IPV6 address and discards the simultaneity with IPV4.

The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.

How to create a `UDPConnection`:

```
UDPConnection _connection = new UDPConnection(60000, "", OnOpen, OnMessage, OnError, OnClose);
```

```
Setup()
public void Setup(int port, string localIP = "", EventVoid evOpen = null, EventMessage evMessage
= null, EventException evError = null, EventVoid evClose = null)
```

This method allows to set or change the internal parameters of the entity. If `Setup()` is called while the entity is connected, then it will be automatically closed. The `UDPConnection` allows to send data once `Setup()` is called, but it will not receive until `Connect()` succeeds. The method will fail if the provided `port`+`localIP` pair is already in use by another entity.
The `port` accepts values between 1 and 65535.
The `localIP` parameter binds the entity to a particular local network adapter.
It accepts the following values:

1. Empty: If an empty `string` is provided (`""`), the entity will automatically bind to the fist default IPV4 and IPV6 addresses simultaneously (this mode is enough in most cases).
2. IPV4: Binds to provided local IPV4 address, this IP must exist in the system or will fail. If a IPV4 address is provided, the simultaneity with IPV6 is not possible.
3. IPV6: Binds to provided local IPV6 address, this IP must exist in the system or will fail. If a IPV6 address is provided, the simultaneity with IPV4 is not possible.
4. URL: Automatically gets the related IP through the available DNS and binds to.
5. `"ipv4"` literal: Binds to the first available IPV4 address and discards the simultaneity with IPV6.

1. `"ipv6"` literal: Binds to the first available IPV6 address and discards the simultaneity with IPV4.

The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.

How to call `Setup()`:

```
_connection.Setup(60000, "", OnOpen, OnMessage, OnError, OnClose);
```

```
Connect()
public void Connect(int port, string remoteIP = "", float keepAliveTimeout = 0f)
```

This method attempts to establish the connection. In `UDPConnection` it means that the entity starts listening and receiving messages.
The `port` accepts values between 1 and 65535.
The `remoteIP` allows to accept messages from a particular remote device or any if left empty (`""`).

How to call `Connect()` to listen all incoming messages to port 60000 from any IP:

```
_connection.Connect(60000);
```

```
public void Disconnect()
```

In `UDPConnection` it stops the listening thread instead of disconnecting. The entity remains ready to `Connect()` again.

```
public void Dispose()
```

This method disconnects the entity and also releases all internal data and resources. The entity remains unusable, a new instance should be created. Use this method when exit the application.

```
public bool DataAvailable()
```

If the `_onMessage` event is not set, then the incoming messages will be stored in an internal buffer. Use this method to know if there are incoming messages available.

```
public byte[] GetMessage()
```

If the `onMessage` event is not set, then the incoming messages will be stored in an internal buffer. Use this method to get the next available incoming message. If there are not available messages, an empty `byte[]` (`null`) will be returned.

```
public void ClearInputBuffer()
```

If the `onMessage` event is not set, then the incoming messages will be stored in an internal buffer. Use this method to flush the incoming messages buffer.

```
public void SendData(IPEndPoint remoteIPEndpoint, byte[] data)
public void SendData(IPEndPoint remoteIPEndpoint, string data)
public void SendData(string ip, byte[] data, int port = 0)
public void SendData(string ip, string data, int port = 0)
```

Use this method to send data. This four overloads allows to send data to a specific remote IP+Port. If the port is not provided the `port` provided in `Connect()` will be used. The `string data` is always encoded to UTF8.

```
public void SendData(byte[] data)
public void SendData(string data)
```

Use this method to send data. This two overloads allows to send data to the `port+remoteIP` provided in `Connect()` automatically. Will do nothing if no `remoteIP` was provided at `Connect()`. The `string data` is always encoded to UTF8.

```
public string GetIP(bool secondary = false)
```

Gets the local IP to which the entity es bind. There can be two addresses simultaneously (IPV4 and IPV6) the primary is IPV4 and the secondary is IPV6. The `secondary` argument determines which IP should be returned. If there is no secondary IP, then an empty `string` will be returned.

```
public string GetIPv4BroadcastAddress()
```
Dynamically calculates the broadcast IP depending on the current local IPV4 address. An empty `string` will be returned if no IPV4 is available.

```
public int GetPort()
```
Returns the current local listening port.

```
public bool IsConnected()
```
Use this method to check if the entity is currently listening.

```
public int GetIOBufferSize()
```
Gets the available input/output buffer. Never send data bigger than this size or the message will be automatically dropped without prompt.

```
public IPEndPoint AddressParser(string ipOrUrl = "ipv4", int port = 0)
```
Converts the provided data into a `System.Net.IPEndPoint` object.
If the `"ipv4"` or `"ipv6"` literals are provided as `ipOrUrl` argument, the `IPEndPoint` will be generated using the default IPV4 or default IPV6 respectively.

```
public string GetDefaultIPAddress(string ipMode = "")
```
This method returns the first available IPV4 or IPV6 address of your device.
The `ipMode` argument also allows the `"ipv4"` and `"ipv6"` literals to force the return.

```
public string[] GetMacAddress()
```
Use this method to list all the MAC addresses of your device.

```
public string ByteArrayToString(byte[] content)
```
Converts a `byte[]` into a `string` using UTF8 encoding.

```
public byte[] StringToByteArray(string content)
```
Converts a `string` into a `byte[]` using UTF8 encoding.

```
public int SecondsToMiliseconds(float seconds)
```
Converts a `float` in seconds into an `int` in milliseconds. Useful when working with timers (`System.Threading.Timer`).

UDPConnection events

All events are invoked as they happen without delays, and all events always runs in a new thread to grant application responsiveness and push at maximum the device performance. Keep this in mind, because it may be an issue when working with some sort of graphic interface as Unity, Windows Forms, Xamarin, etc. As usual, always keep the code of the events as short as possible.

```
public volatile EventVoid onOpen
```
This event is invoked when the `UDPConnection` starts listening. If any problem occurs and the listening can't be started, then a `onError` event is invoked instead.
The signature is: `void OnOpen(UDPConnection)`
The `UDPConnection` is the reference to the entity which invoked the event.

```
public volatile EventMessage onMessage
```
This event is invoked when a message has been received.
The signature is: `void OnMessage(byte[], string, UDPConnection)`
The `byte[]` is the received message.
The `string` is the remote IP (this connection can receive messages from several other connections).
The `UDPConnection` is the reference to the entity which invoked the event.

```
public volatile EventException onError
```
This event is invoked when an internal error has occurred. Assume that after an internal error the entity is not operative any more, and needs to be reconnected (or restarted in this case).
The signature is: `void OnError(int, string, UDPConnection)`
The `int` is the error code.
The `string` is error description.
The `UDPConnection` is the reference to the entity which invoked the event.

```
public volatile EventVoid onClose
```
This event is invoked when the connection closes unexpectedly. In most cases UDP reconnects automatically and continues operating with the shortest possible interruption.
The signature is: `void OnClose(UDPConnection)`
The `UDPConnection` is the reference to the entity which invoked the event.

Quick reference

This piece of code is included in the `UDPConnection` header to ease the task of defining the events. Copy to your code, complete with what you need and assign to the delegates:

```
    void OnUDPOpen(UDPConnection connection)
    { }
    void OnUDPMessage(byte[] message, string remoteIP, UDPConnection connection)
    { }
    void OnUDPError(int code, string message, UDPConnection connection)
    { }
    void OnUDPClose(UDPConnection connection)
    { }
```

Events are assigned like this:

```
    _connection.onOpen = OnTCPOpen; // Use this if you need only one call.
    _connection.onOpen += OnTCPOpen; // Use this if you need several calls.
```

TCPConnection

Here is the complete description of the `TCPConnection` wrapper and how to use it.
You can modify the parameters of this entity dynamically by calling `Setup()`. Once `Setup()` is called, the connection will be closed and must be reconnected again by calling `Connect()`.
All interaction with this entity is done through the 4 events.
This entity will work as a "stream" by default because it fixes fragmentation problems in all use cases and it's the only way to probe line activity. Disable this feature (`streamIn`) only if you know exactly that it's what you need.

TCPConnection methods

The methods are provided to keep control of the entity.

```
Constructor (new TCPConnection())
public TCPConnection(string localIP = "", EventVoid evOpen = null, EventMessage evMessage = null,
EventException evError = null, EventVoid evClose = null, byte[] eof = null)
```

The constructor allows to create the entity and set its parameters all at once, calling `Setup()` internally.
The `localIP` parameter binds the entity to a particular local network adapter.
It accepts the following values:

1. Empty: If an empty `string` is provided (`""`), the entity will automatically bind to the fist default IPV4 or IPV6 address (this mode is enough in most cases).
2. IPV4: Binds to provided local IPV4 address, this IP must exist in the system or will fail. This `localIP` type must match the remote server IP type to be able to connect.
3. IPV6: Binds to provided local IPV6 address, this IP must exist in the system or will fail. This `localIP` type must match the remote server IP type to be able to connect.
4. URL: Automatically gets the related IP through the available DNS and binds to.
5. `"ipv4"` literal: Binds to the first available IPV4 address automatically.
6. `"ipv6"` literal: Binds to the first available IPV6 address automatically.

The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.
There is a constructor overload, but it's for internal use by `TCPServer`, so there is no point in giving an extended explanation (It's used to set the new incoming connections).
The `eof` array is the message delimiter. Then the `onMessage` event is fired, this sequence is already removed. Leaving this to `null` will not affect the default (or previously set) `_eof`.

How to create a `TCPConnection`:

```
TCPConnection _connection = new TCPConnection("", OnOpen, OnMessage, OnError, OnClose);
```

```
Setup()
public void Setup(string localIP = "", EventVoid evOpen = null, EventMessage evMessage = null,
EventException evError = null, EventVoid evClose = null, byte[] eof = null)
```

This method allows to set or change the internal parameters of the entity. If `Setup()` is called while the entity is connected, then it will be automatically closed.
The `localIP` parameter binds the entity to a particular local network adapter.
It accepts the following values:

1. Empty: If an empty `string` is provided (`""`), the entity will automatically bind to the fist default IPV4 or IPV6 address (this mode is enough in most cases).
2. IPV4: Binds to provided local IPV4 address, this IP must exist in the system or will fail. This `localIP` type must match the remote server IP type to be able to connect.

3. IPV6: Binds to provided local IPV6 address, this IP must exist in the system or will fail. This `localIP` type must match the remote server IP type to be able to connect.
4. URL: Automatically gets the related IP through the available DNS and binds to.
5. `"ipv4"` literal: Binds to the first available IPV4 address automatically.
6. `"ipv6"` literal: Binds to the first available IPV6 address automatically.

The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.
The `eof` array is the message delimiter. Then the `onMessage` event is fired, this sequence is already removed. Leaving this to `null` will not affect the default (or previously set) `_eof`.

How to call `Setup()`:

```
_connection.Setup("", OnOpen, OnMessage, OnError, OnClose);
```

```
SetEOF()
public void SetEOF(byte[] eof)
public void SetEOF(string eof)
```

This method sets the `_eof` sequence in order to detect message termination (essential for fragmented messages). The `string` overload will convert the `eof` into an UTF8 encoded `byte[]`.
If the `_eof` is set to `null` or empty, the termination detection will be disabled and the `TCPConnection` will fire the `onMessage` event on each received data frame.
This method also allows to switch from "data-frame" mode to "stream" mode dynamically.

How to call `SetEOF()`:

```
_connection.SetEOF("\r\n");
```

```
ClearEOF()
public void ClearEOF()
```

This method clears the `_eof` sequence in order to work in "data-frame" mode.
This method also allows to switch from "stream" mode to "data-frame" mode dynamically.

How to call `ClearEOF()`:

```
_connection.ClearEOF();
```

```
Connect()
public void Connect(int port, string remoteIP, float timeout = 5f, float keepAliveTimeout = 15f,
bool disableWatchdog = false)
```

This method attempts to establish the connection. If the remote server is not available, the entity will retry the connection every `timeout` seconds indefinitely or until `Disconnect()` is called.
Once the connection is established, a "keepAlive" empty message will be sent every `keepAliveTimeout` seconds in order to keep the line open and also to detect connection failures due to infrastructure issues. If no response is received from the server within this `keepAliveTimeout` time, the connection will be assumed and forced as closed (you can disable this feature through the `disableWatchdog` flag).

How to call `Connect()`:

```
_connection.Connect(60000, "192.168.0.10");
```

```
public void Disconnect()
```

Closes the connection with the server. The entity remains ready to `Connect()` again.

```
public void Dispose()
```

This method disconnects the entity and also releases all internal data and resources. The entity remains unusable, a new instance should be created. Use this method when exit the application.

```
public bool DataAvailable()
```

If the `onMessage` event is not set, then the incoming messages will be stored in an internal buffer.

Use this method to know if there are incoming messages available.

```
public byte[] GetMessage()
```
If the `onMessage` event is not set, then the incoming messages will be stored in an internal buffer.
Use this method to get the next available incoming message. If there are not available messages, an empty `byte[]` (`null`) will be returned.

```
public void ClearInputBuffer()
```
If the `onMessage` event is not set, then the incoming messages will be stored in an internal buffer.
Use this method to flush the incoming messages buffer.

```
public void SendData(byte[] data)
public void SendData(string data)
```
Use this method to send data to the server. The `string data` is always encoded to UTF8.

```
public string GetIP()
```
Gets the local IP to which the entity es bind.

```
public int GetPort()
```
Returns the current local listening port.

```
public bool IsConnected()
```
Use this method to check if the entity is currently connected to the server.

```
public string GetRemoteIP()
```
Returns the last valid remote server IP.

```
public string GetRemotePort()
```
Returns the last valid remote server port.

```
public IPEndPoint AddressParser(string ipOrUrl = "ipv4", int port = 0)
```
Converts the provided data into a `System.Net.IPEndPoint` object.
If the `"ipv4"` or `"ipv6"` literals are provided as `ipOrUrl` argument, the `IPEndPoint` will be generated using the default IPV4 or default IPV6 respectively.

```
public string GetDefaultIPAddress(string ipMode = "")
```
This method returns the first available IPV4 or IPV6 address of your device.
The `ipMode` argument also allows the `"ipv4"` and `"ipv6"` literals to force the return.

```
public string[] GetMacAddress()
```
Use this method to list all the MAC addresses of your device.

```
public string ByteArrayToString(byte[] content)
```
Converts a `byte[]` into a `string` using UTF8 encoding.

```
public byte[] StringToByteArray(string content)
```
Converts a `string` into a `byte[]` using UTF8 encoding.

```
public int SecondsToMiliseconds(float seconds)
```
Converts a `float` in seconds into an `int` in milliseconds. Useful when working with timers (`System.Threading.Timer`).

TCPConnection events

All events are invoked as they happen without delays, and all events always runs in a new thread to grant application responsiveness and push at maximum the device performance. Keep this in mind, because it may be an issue when working with some sort of graphic interface as Unity, Windows Forms, Xamarin, etc. As usual, always keep the code of the events as short as possible.

`public volatile EventVoid onOpen`

This event is invoked when the `TCPConnection` succeeds establishing the connection.
The signature is: `void OnOpen(TCPConnection)`
The `TCPConnection` is the reference to the entity which invoked the event.

`public volatile EventMessage onMessage`

This event is invoked when a message has been received.
The signature is: `void OnMessage(byte[], TCPConnection)`
The `byte[]` is the received message.
The `TCPConnection` is the reference to the entity which invoked the event.

`public volatile EventException onError`

This event is invoked when an internal error has occurred. Assume that after an internal error the entity is not operative any more, and needs to be reconnected.
The signature is: `void OnError(int, string, TCPConnection)`
The `int` is the error code.
The `string` is error description.
The `TCPConnection` is the reference to the entity which invoked the event.

`public volatile EventVoid onClose`

This event is invoked when the connection closes unexpectedly (or when timed out).
The signature is: `void OnClose(TCPConnection)`
The `TCPConnection` is the reference to the entity which invoked the event.

Quick reference

This piece of code is included in the `TCPConnection` header to ease the task of defining the events. Copy to your code, complete with what you need and assign to the delegates:

```
    public void OnTCPOpen(TCPConnection connection)
    { }
    public void OnTCPMessage(byte[] message, TCPConnection connection)
    { }
    public void OnTCPError(int code, string message, TCPConnection connection)
    { }
    public void OnTCPClose(TCPConnection connection)
    { }
```

Events are assigned like this:

```
    _connection.onOpen = OnTCPOpen;  // Use this if you need only one call.
    _connection.onOpen += OnTCPOpen; // Use this if you need several calls.
```

TCPServer

Here is the complete description of the `TCPServer` and how to use it.
You can modify the parameters of this entity dynamically by calling `Setup()`. Once `Setup()` is called, the server will be closed and must be reconnected again by calling `Connect()`.
All interaction with this entity is done through the 4 events.

TCPServer methods

The methods are provided to keep control of the entity.

```
Constructor (new TCPServer())
public TCPServer(int port, string localIP = "", EventVoid evOpen = null, EventConnection
evConnection = null, EventException evError = null, EventVoid evClose = null, int maxConnections
= 100, float keepAliveTimeout = 40f)
```

The constructor allows to create the entity and set its parameters all at once, calling `Setup()` internally. The method will fail if the provided `port`+`localIP` pair is already in use by another entity.
The `port` accepts values between 1 and 65535. Servers needs to get this port open in the local firewall.
The `localIP` parameter binds the entity to a particular local network adapter.
It accepts the following values:

1. Empty: If an empty `string` is provided (`""`), the entity will automatically bind to the fist default IPV4 and IPV6 addresses simultaneously (this mode is enough in most cases).
2. IPV4: Binds to provided local IPV4 address, this IP must exist in the system or will fail. If a IPV4 address is provided, the simultaneity with IPV6 is not possible.
3. IPV6: Binds to provided local IPV6 address, this IP must exist in the system or will fail. If a IPV6 address is provided, the simultaneity with IPV4 is not possible.
4. URL: Automatically gets the related IP through the available DNS and binds to.
5. `"ipv4"` literal: Binds to the first available IPV4 address and discards the simultaneity with IPV6.
6. `"ipv6"` literal: Binds to the first available IPV6 address and discards the simultaneity with IPV4.

The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.
The `maxConnections` parameter sets the maximum number of concurrent connections allowed.
The `keepAliveTimeout` time sets the limit to close inactive connections (disable setting its value to `0f`).

How to create a `TCPServer` with default values (100 connections and 30 secconds for timeout):

```
TCPServer _server = new TCPServer(60000, "", OnOpen, OnNewConnection, OnError, OnClose);
```

```
Setup()
public void Setup(int port, string localIP = "", EventVoid evOpen = null, EventConnection
evConnection = null, EventException evError = null, EventVoid evClose = null, int maxConnections
= 100, float keepAliveTimeout = 40f)
```

This method allows to set or change the internal parameters of the entity. If `Setup()` is called while the entity is connected, then it will be automatically closed. The method will fail if the provided `port`+`localIP` pair is already in use by another entity.
The `port` accepts values between 1 and 65535.
The `localIP` parameter binds the entity to a particular local network adapter.
It accepts the following values:

1. Empty: If an empty `string` is provided (`""`), the entity will automatically bind to the fist default IPV4 and IPV6 addresses simultaneously (this mode is enough in most cases).
2. IPV4: Binds to provided local IPV4 address, this IP must exist in the system or will fail. If a IPV4 address is provided, the simultaneity with IPV6 is not possible.
3. IPV6: Binds to provided local IPV6 address, this IP must exist in the system or will fail. If a IPV6 address is provided, the simultaneity with IPV4 is not possible.

4. URL: Automatically gets the related IP through the available DNS and binds to.
5. `"ipv4"` literal: Binds to the first available IPV4 address and discards the simultaneity with IPV6.
6. `"ipv6"` literal: Binds to the first available IPV6 address and discards the simultaneity with IPV4.

The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.
The `maxConnections` parameter sets the maximum number of concurrent connections allowed.
The `keepAliveTimeout` time sets the limit to close inactive connections (disable setting its value to `0f`).

How to call `Setup()` to modify the maximum value of connections and timeout:

```
_server.Setup(60000, "", OnOpen, OnNewConnection, OnError, OnClose, 200, 60f);
```

```
public void Connect()
```

This method attempts to establish the connection. In `TCPServer` it means that the entity starts listening and receiving incoming connections.

```
public void Disconnect()
```

In `TCPServer` it stops the listening thread and closes all active connections. The entity remains ready to `Connect()` again.

```
public void Dispose()
```

This method disconnects the entity and also releases all internal data and resources. The entity remains unusable, a new instance should be created. Use this method when exit the application.

```
public void CloseConnection(TCPConnection connection)
```

This method allows to close a particular active connection.

```
public void Distribute(byte[] data)
public void Distribute(string data)
```

Use this method to send data to all active connections. The `string data` is always encoded to UTF8.

```
public int GetConnectionsCount()
```

This method returns the total count of active connections.

```
public TCPConnection GetConnection(int index)
```

Gets a particular connection from the internal list of active connections.

```
public string GetIP(bool secondary = false)
```

Gets the local IP to which the entity es bind. There can be two addresses simultaneously (IPV4 and IPV6) the primary is IPV4 and the secondary is IPV6. The `secondary` argument determines which IP should be returned. If there is no secondary IP, then an empty `string` will be returned.

```
public int GetPort()
```

Returns the current local listening port.

```
public bool IsConnected()
```

Use this method to check if the entity is currently listening.

```
public IPEndPoint AddressParser(string ipOrUrl = "ipv4", int port = 0)
```

Converts the provided data into a `System.Net.IPEndPoint` object.
If the `"ipv4"` or `"ipv6"` literals are provided as `ipOrUrl` argument, the `IPEndPoint` will be generated using the default IPV4 or default IPV6 respectively.

```
public string GetDefaultIPAddress(string ipMode = "")
```

This method returns the first available IPV4 or IPV6 address of your device.
The `ipMode` argument also allows the `"ipv4"` and `"ipv6"` literals to force the return.

```
public string[] GetMacAddress()
```

Use this method to list all the MAC addresses of your device.

TCPServer events

All events are invoked as they happen without delays, and all events always runs in a new thread to grant application responsiveness and push at maximum the device performance. Keep this in mind, because it may be an issue when working with some sort of graphic interface as Unity, Windows Forms, Xamarin, etc. As usual, always keep the code of the events as short as possible.

```
public volatile EventVoid onOpen
```

This event is invoked when the `TCPServer` starts listening. If any problem occurs and the listening can't be started, then a `onError` event is invoked instead.
The signature is: `void OnOpen(TCPServer)`
The `TCPServer` is the reference to the entity which invoked the event.

```
public volatile EventConnection onNewConnection
```

This event is invoked when a message has been received.
The signature is: `void OnMessage(TCPConnection, TCPServer)`
The `TCPConnection` is the new incoming connection. Set the events of this connection here.
The `TCPServer` is the reference to the entity which invoked the event.

```
public volatile EventException onError
```

This event is invoked when an internal error has occurred. Assume that after an internal error the entity is not operative any more, and needs to be reconnected (or restarted in this case).
The signature is: `void OnError(int, string, TCPServer)`
The `int` is the error code.
The `string` is error description.
The `TCPServer` is the reference to the entity which invoked the event.

```
public volatile EventVoid onClose
```

This event is invoked when the server stops unexpectedly.
The signature is: `void OnClose(TCPServer)`
The `TCPServer` is the reference to the entity which invoked the event.

Quick reference

This piece of code is included in the `TCPServer` header to ease the task of defining the events. Copy to your code, complete with what you need and assign to the delegates:

```
    // Server:
    void OnTCPSOpen(TCPServer server)
    { }
    void OnTCPSNewConnection(TCPConnection connection, TCPServer server)
    { }
    void OnTCPSError(int code, string message, TCPServer server)
    { }
    void OnTCPSClose(TCPServer server)
    { }
    // Client (incoming connection):
    void OnTCPMessage(byte[] message, TCPConnection connection)
    { }
    void OnTCPError(int code, string message, TCPConnection connection)
    { }
    void OnTCPClose(TCPConnection connection)
    { }
```

Events are assigned like this:

```
    _server.onOpen = OnTCPSOpen;     // Use this if you need only one call.
    _server.onOpen += OnTCPSOpen;    // Use this if you need several calls.
```

WSConnection

Here is the complete description of the `WSConnection` and how to use it.
You can modify the parameters of this entity dynamically by calling `Setup()`. Once `Setup()` is called, the server will be closed and must be reconnected again by calling `Connect()`.
All interaction with this entity is done through the 4 events.

If you are building for Bridge.net you should enable the next directive:
```
#define BRIDGE_NET
```

WSConnection methods

The methods are provided to keep control of the entity.

```
Constructor (new WSConnection())
public WSConnection(EventVoid evOpen = null, EventMessage evMessage = null, EventException
evError = null, EventVoid evClose = null)
```

The constructor allows to create the entity and set its parameters all at once, calling `Setup()` internally.
The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.
There is a constructor overload, but it's for internal use by `WSServer`, so there is no point in giving an extended explanation (It's used to set the new incoming connections).

How to create a `WSConnection`:

```
WSConnection _connection = new WSConnection(OnOpen, OnMessage, OnError, OnClose);
```

```
Setup()
public void Setup(EventVoid evOpen = null, EventMessage evMessage = null, EventException evError
= null, EventVoid evClose = null)
```

This method allows to set or change the internal parameters of the entity. If `Setup()` is called while the entity is connected, then it will be automatically closed.
The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.

How to call `Setup()`:

```
_connection.Setup(OnOpen, OnMessage, OnError, OnClose);
```

```
Connect()
public void Connect(string serverURL, float timeout = 5f, float keepAliveTimeout = 15f, bool
disableWatchdog = false)
```

This method attempts to establish the connection. If the remote server is not available, the entity will retry the connection every `timeout` seconds indefinitely or until `Disconnect()` is called.
The `serverURL` is the remote address to connect to, it must be provided in the following formats:

1. For IPV4: `"http://44.33.22.11:60000"`, `"https://44.33.22.11:60000"`, `"ws://44.33.22.11:60000"` or `"wss://44.33.22.11:60000"`.
2. For IPV6: `"http://[8888:7777::2222:1111]:60000"`, `"https://[8888:7777::2222:1111]:60000"`, `"ws://[8888:7777::2222:1111]:60000"`, or `"wss://[8888:7777::2222:1111]:60000"`.
3. For URL: `"http://myserver.com:60000"`, `"https://myserver.com:60000"`, `"ws://myserver.com:60000"`, or `"wss://myserver.com:60000"`.

This parameter must be provided in order to establish the connection, or the `WSConnection` will be not able to find the server. This URL must match also the service ID defined in server side. The ending backslash is optional.
The `timeout` parameter sets the time to wait until to fire the onClose event and then retry the connection automatically (Not yet available on WebGL builds).

Once the connection is established, a "keepAlive" empty message will be sent every **keepAliveTimeout** seconds (Not yet available on WebGL builds) in order to keep the line open and also to detect connection failures due to infrastructure issues. If no response is received from the server within this **keepAliveTimeout** time, the connection will be assumed and forced as closed (you can disable this feature through the **disableWatchdog** flag).

How to call **Connect()** with default values:

```
_connection.Connect("ws://myserver.com:60000/ws");
```

```
public void Disconnect()
```

Closes the connection with the server. The entity remains ready to **Connect()** again.

```
public void Dispose()
```

This method disconnects the entity and also releases all internal data and resources. The entity remains unusable, a new instance should be created. Use this method when exit the application.

```
public bool DataAvailable()
```

If the **onMessage** event is not set, then the incoming messages will be stored in an internal buffer. Use this method to know if there are incoming messages available.

```
public byte[] GetMessage()
```

If the **onMessage** event is not set, then the incoming messages will be stored in an internal buffer. Use this method to get the next available incoming message. If there are not available messages, an empty `byte[]` (`null`) will be returned.

```
public void ClearInputBuffer()
```

If the **onMessage** event is not set, then the incoming messages will be stored in an internal buffer. Use this method to flush the incoming messages buffer.

```
public void SendData(byte[] data)
public void SendData(string data)
```

Use this method to send data to the server. The `string` **data** is always encoded to UTF8.

```
public string GetURL()
```

Returns the provided server URL including the port.

```
public bool IsConnected()
```

Use this method to check if the entity is currently connected to the server.

```
public string GetDefaultIPAddress(string ipMode = "")
```

This method returns the first available IPV4 or IPV6 address of your device.
The **ipMode** argument also allows the `"ipv4"` and `"ipv6"` literals to force the return.

```
public string[] GetMacAddress()
```

Use this method to list all the MAC addresses of your device.

```
public string ByteArrayToString(byte[] content)
```

Converts a `byte[]` into a `string` using UTF8 encoding.

```
public byte[] StringToByteArray(string content)
```

Converts a `string` into a `byte[]` using UTF8 encoding.

```
public int SecondsToMiliseconds(float seconds)
```

Converts a `float` in seconds into an `int` in milliseconds. Useful when working with timers (**System.Threading.Timer**).

WSConnection events

All events are invoked as they happen without delays, and all events always runs in a new thread to grant application responsiveness and push at maximum the device performance. Keep this in mind, because it may be an issue when working with some sort of graphic interface as Unity, Windows Forms, Xamarin, etc. As usual, always keep the code of the events as short as possible.

```
public volatile EventVoid onOpen
```

This event is invoked when the `WSConnection` succeeds establishing the connection.
The signature is: `void OnOpen(WSConnection)`
The `WSConnection` is the reference to the entity which invoked the event.

```
public volatile EventMessage onMessage
```

This event is invoked when a message has been received.
The signature is: `void OnMessage(byte[], WSConnection)`
The `byte[]` is the received message.
The `WSConnection` is the reference to the entity which invoked the event.

```
public volatile EventException onError
```

This event is invoked when an internal error has occurred. Assume that after an internal error the entity is not operative any more, and needs to be reconnected.
The signature is: `void OnError(int, string, WSConnection)`
The `int` is the error code.
The `string` is error description.
The `WSConnection` is the reference to the entity which invoked the event.

```
public volatile EventVoid onClose
```

This event is invoked when the connection closes unexpectedly (or when timed out).
The signature is: `void OnClose(WSConnection)`
The `WSConnection` is the reference to the entity which invoked the event.

Quick reference

This piece of code is included in the `WSConnection` header to ease the task of defining the events. Copy to your code, complete with what you need and assign to the delegates:

```
    public void OnWSOpen(WSConnection connection)
    { }
    public void OnWSMessage(byte[] message, WSConnection connection)
    { }
    public void OnWSError(int code, string message, WSConnection connection)
    { }
    public void OnWSClose(WSConnection connection)
    { }
```

Events are assigned like this:

```
    _connection.onOpen = OnWSOpen;          // Use this if you need only one call.
    _connection.onOpen += OnWSOpen; // Use this if you need several calls.
```

WSServer

Here is the complete description of the `WSServer` and how to use it.
You can modify the parameters of this entity dynamically calling `Setup()`. Once `Setup()` is called, the server will be closed and must be reconnected again calling `Connect()`.
All interaction with this entity is done through the 4 events.

If you are building for `.net` platforms, then you don't need anything else than the WSServer.cs file.
But if you are building for MonoDevelop you'll have to add the custom version of WebSocket-Sharp, and remove the comment-mark in the next line (It will work just as it does in Unity):

```
#define UNITY_2017_1_OR_NEWER
```

WSServer methods

The methods are provided to keep control of the entity.

```
Constructor (new WSServer())
public WSServer(int port, string localIP = "", string service = "", EventVoid evOpen = null,
EventConnection evConnection = null, EventException evError = null, EventVoid evClose =
null, int maxConnections = 100, float keepAliveTimeout = 40f)

public WSServer(string localAddress, EventVoid evOpen = null, EventConnection evConnection =
null, EventException evError = null, EventVoid evClose = null, int maxConnections = 100,
float keepAliveTimeout = 40f)
```

The constructor allows to create the entity and set its parameters all at once, calling `Setup()` internally.
The first overload will fail if the provided `port`+`localIP` pair is already in use by another entity.
The `port` accepts values between 1 and 65535. Servers needs to get this port open in the local firewall.
The `localIP` parameter binds the entity to a particular local network adapter.
It accepts the following values:

1. Empty: If an empty `string` is provided (`""`), the entity will automatically bind to the fist default IPV4 and IPV6 addresses simultaneously (this mode is enough in most cases).
2. IPV4: Binds to provided local IPV4 address, this IP must exist in the system or will fail. If a IPV4 address is provided, the simultaneity with IPV6 is not possible.
3. IPV6: Binds to provided local IPV6 address, this IP must exist in the system or will fail. If a IPV6 address is provided, the simultaneity with IPV4 is not possible.
4. URL: Automatically gets the related IP through the available DNS and binds to.
5. `"ipv4"` literal: Binds to the first available IPV4 address and discards the simultaneity with IPV6.
6. `"ipv6"` literal: Binds to the first available IPV6 address and discards the simultaneity with IPV4.

Use the `service` parameter as an identifier for the URL. It doesn't has any effect.
The second overload will fail if the provided `localAddress` is already in use by another entity.
The `localAddress` is the public address to receive incoming connections, it must be provided in the following formats:

1. For IPV4: `"http://44.33.22.11:60000"` or `"ws://44.33.22.11:60000"`.
2. For IPV6: `"http://[8888:7777::2222:1111]:60000"` or `"ws://[8888:7777::2222:1111]:60000"`.

NOTE: At the moment it doesn't supports "WSS" (but it'll be fine if you don't use the 80 or 8080 ports).
NOTE: Always provide an IP address to this overload, at the moment it doesn't supports URLs.
The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.
The `maxConnections` parameter sets the maximum number of concurrent connections allowed.
The `keepAliveTimeout` time sets the limit to close inactive connections (disable setting its value to `0f`).

How to create a `WSServer` with the default amount of connections:

```
WSServer _server = new WSServer("http://44.33.22.11:60000/chat", OnOpen, OnNewConnection,
OnError, OnClose);
```

```
Setup()
public void Setup(string localAddress, EventVoid evOpen = null, EventConnection evConnection
= null, EventException evError = null, EventVoid evClose = null, int maxConnections = 100,
float keepAliveTimeout = 40f)

public void Setup(int port, string localIP = "", string service = "", EventVoid evOpen =
null, EventConnection evConnection = null, EventException evError = null, EventVoid evClose
= null, int maxConnections = 100, float keepAliveTimeout = 40f)
```

This method allows to set or change the internal parameters of the entity. If `Setup()` is called while the entity is connected, then it will be automatically closed.
The first overload will fail if the provided `port`+`localIP` pair is already in use by another entity.
The `port` accepts values between 1 and 65535. Servers needs to get this port open in the local firewall.
The `localIP` parameter binds the entity to a particular local network adapter.
It accepts the following values:

1. Empty: If an empty `string` is provided (`""`), the entity will automatically bind to the fist default IPV4 and IPV6 addresses simultaneously (this mode is enough in most cases).
2. IPV4: Binds to provided local IPV4 address, this IP must exist in the system or will fail. If a IPV4 address is provided, the simultaneity with IPV6 is not possible.
3. IPV6: Binds to provided local IPV6 address, this IP must exist in the system or will fail. If a IPV6 address is provided, the simultaneity with IPV4 is not possible.
4. URL: Automatically gets the related IP through the available DNS and binds to.
5. `"ipv4"` literal: Binds to the first available IPV4 address and discards the simultaneity with IPV6.
6. `"ipv6"` literal: Binds to the first available IPV6 address and discards the simultaneity with IPV4.

Use the `service` parameter as an identifier for the URL. It doesn't has any effect.
The second overload will fail if the provided `localAddress` is already in use by another entity.
The `localAddress` is the public address to receive incoming connections, it must be provided in the following formats:

1. For IPV4: `"http://44.33.22.11:60000"` or `"ws://44.33.22.11:60000"`.
2. For IPV6: `"http://[8888:7777::2222:1111]:60000"` or `"ws://[8888:7777::2222:1111]:60000"`.

NOTE: At the moment it doesn't supports "WSS" (but it'll be fine if you don't use the 80 or 8080 ports).
NOTE: Always provide an IP address to this overload, at the moment it doesn't supports URLs.
The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.
The `maxConnections` parameter sets the maximum number of concurrent connections allowed.
The `keepAliveTimeout` time sets the limit to close inactive connections (disable setting its value to `0f`).

How to call `Setup()` to modify the maximum value of connections to 200:

```
_server.Setup("http://44.33.22.11:60000/chat", OnOpen, OnNewConnection, OnError, OnClose, 200);
```

```
public void Connect()
```

This method attempts to establish the connection. In `WSServer` it means that the entity starts listening and receiving incoming connections.

```
public void Disconnect()
```

In `WSServer` it stops the listening thread and closes all active connections. The entity remains ready to `Connect()` again.

```
public void Dispose()
```

This method disconnects the entity and also releases all internal data and resources. The entity remains unusable, a new instance should be created. Use this method when exit the application.

```
public void CloseConnection(WSConnection connection)
```
This method allows to close a particular active connection.

```
public void Distribute(byte[] data)
public void Distribute(string data)
```
Use this method to send data to all active connections. The `string data` is always encoded to UTF8.

```
public int GetConnectionsCount()
```
This method returns the total count of active connections.

```
public WSConnection GetConnection(int index)
```
Gets a particular connection from the internal list of active connections.

```
public string GetIP(bool secondary = false)
```
Gets the local IP to which the entity is bind. There can be two addresses simultaneously (IPV4 and IPV6) the primary is IPV4 and the secondary is IPV6. The `secondary` argument determines which IP should be returned. If there is no secondary IP, then an empty `string` will be returned.

```
public int GetPort()
```
Returns the current local listening port.

```
public string GetURL(bool secondary = false)
```
Gets the local URL to which the entity is bind (including port and service). There can be two addresses simultaneously (IPV4 and IPV6) the primary is IPV4 and the secondary is IPV6. The `secondary` argument determines which URL should be returned. If there is no secondary URL, then an empty `string` will be returned.

```
public bool IsConnected()
```
Use this method to check if the entity is currently listening.

```
public string GetDefaultIPAddress(string ipMode = "")
```
This method returns the first available IPV4 or IPV6 address of your device.
The `ipMode` argument also allows the `"ipv4"` and `"ipv6"` literals to force the return.

```
public string[] GetMacAddress()
```
Use this method to list all the MAC addresses of your device.

WSServer events

All events are invoked as they happen without delays, and all events always runs in a new thread to grant application responsiveness and push at maximum the device performance. Keep this in mind, because it may be an issue when working with some sort of graphic interface as Unity, Windows Forms, Xamarin, etc. As usual, always keep the code of the events as short as possible.

```
public volatile EventVoid onOpen
```
This event is invoked when the `WSServer` starts listening. If any problem occurs and the listening can't be started, then a `_onError` event is invoked instead.
The signature is: `void OnOpen(WSServer)`
The `WSServer` is the reference to the entity which invoked the event.

```
public volatile EventConnection onNewConnection
```
This event is invoked when a message has been received.
The signature is: `void OnMessage(WSConnection, WSServer)`
The `WSConnection` is the new incoming connection.
The `WSServer` is the reference to the entity which invoked the event.

```
public volatile EventException onError
```
This event is invoked when an internal error has occurred. Assume that after an internal error the entity is not operative any more, and needs to be reconnected (or restarted in this case).
The signature is: `void OnError(int, string, WSServer)`
The `int` is the error code.
The `string` is error description.
The `WSServer` is the reference to the entity which invoked the event.

```
public volatile EventVoid onClose
```
This event is invoked when the server stops unexpectedly.
The signature is: `void OnClose(WSServer)`
The `WSServer` is the reference to the entity which invoked the event.

Quick reference

This piece of code is included in the `WSServer` header to ease the task of defining the events. Copy to your code, complete with what you need and assign to the delegates:

```
    // Server:
    void OnWSSOpen(WSServer server)
    { }
    void OnWSSNewConnection(WSConnection connection, WSServer server)
    { }
    void OnWSSError(int code, string message, WSServer server)
    { }
    void OnWSSClose(WSServer server)
    { }
    // Client (incoming connection):
    void OnWSMessage(byte[] message, WSConnection connection)
    { }
    void OnWSError(int code, string message, WSConnection connection)
    { }
    void OnWSClose(WSConnection connection)
    { }
```

Events are assigned like this:

```
    _server.onOpen = OnWSSOpen;      // Use this if you need only one call.
    _server.onOpen += OnWSSOpen;     // Use this if you need several calls.
```

RS232Connection

Here is the complete description of the `RS232Connection` class and how to use it.
You can modify the parameters of this entity dynamically by calling `Setup()`. Once `Setup()` is called, the connection will be closed and must be reconnected again by calling `Connect()`.
All interaction with this entity is done through the 4 events.

The RS232 connection is not a standard protocol, but an electric standard, so you should adapt this class to your particular needs. It implements the most common package control, but unfortunately it's far from being usable in all cases as it is.
NOTE: To implement your own version first make a copy of `RS232Connection` and then rename accordingly (file and class must match), then search the `void ReceiveData(object state)` method and set there your custom message detection (the code is commented, don't be afraid).

RS232Connection methods

The methods are provided to keep control of the entity.

```
Constructor (new RS232Connection())
public RS232Connection(string port, int baudrate, byte eof, EventVoid evOpen = null, EventMessage
evMessage = null, EventException evError = null, EventVoid evClose = null, float latency = 0.1f,
Parity parity = Parity.None, int dataBits = 8, StopBits stopBits = StopBits.One, Handshake
handshake = Handshake.None)
```

The constructor allows to create the entity and set its parameters all at once, calling `Setup()` internally.
The `port` must contain an available name of the port. You can get the ports with `GetAvailablePorts()`.
The `baudrate` is the "speed" of the port, it allows any value from 1 to the maximum that alows the UART being used (this is a hardware feature, not standard at all, different UARTS from different vendors has their own features). The most common values in the industry are: 1200, 2400, 4800, 9600, 19200 and 115200 bauds. The `baudrate` must imperatively match in both sides.
The `eof` byte/character is used to detect the end of a message, and should always be provided, otherwise the incoming data will be accumulated internally and the `onMessage` event will never be fired.
The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.

The next parameters are kept with it's default values in most cases:
The `latency` is the time used for the UART to discard an incomplete incoming byte, it's also used internally to set the time period to check available incoming data.
The `parity` is an extra bit calculated from an incoming byte to detect errors (For UART's internal use).
The `dataBits` is the size of the incoming byte. Nowadays all systems uses 8 bits.
The `stopBits` is an extra bit intended to determine the end of a byte (For UART's internal use).
The `handshake` is used to stop the incoming data until new confirmation. Nowadays all UARTs are able to handle data very quickly, so it's rarely used.

How to create a `RS232Connection`:

```
RS232Connection _connection = new RS232Connection("COM3", 9600, (byte)'*', OnCOMOpen,
OnCOMMessage, OnCOMError, OnCOMClose);
```

```
Setup()
public void Setup(string port, int baudrate, byte eof, EventVoid evOpen = null, EventMessage
evMessage = null, EventException evError = null, EventVoid evClose = null, float latency = 0.1f,
```

```
Parity parity = Parity.None, int dataBits = 8, StopBits stopBits = StopBits.One, Handshake handshake = Handshake.None)
```

This method allows to set or change the internal parameters of the entity. If `Setup()` is called while the entity is connected, then it will be automatically closed.
The `port` must contain an available name of the port. You can get the ports with `GetAvailablePorts()`.
The `baudrate` is the "speed" of the port, it allows any value from 1 to the maximum that alows the UART being used (this is a hardware feature, not standard at all, different UARTS from different vendors has their own features). The most common values in the industry are: 1200, 2400, 4800, 9600, 19200 and 115200 bauds. The `baudrate` must imperatively match in both sides.
The `eof` byte/character is used to detect the end of a message, and should always be provided, otherwise the incoming data will be accumulated internally and the `onMessage` event will never be fired.
The `evOpen`, `evMessage`, `evError` and `evClose` are the events, they are explained in the next section.

The next parameters are kept with it's default values in most cases:
The `latency` is the time used for the UART to discard an incomplete incoming byte, it's also used internally to set the time period to check available incoming data.
The `parity` is an extra bit calculated from an incoming byte to detect errors (For UART's internal use).
The `dataBits` is the size of the incoming byte. Nowadays all systems uses 8 bits.
The `stopBits` is an extra bit intended to determine the end of a byte (For UART's internal use).
The `handshake` is used to stop the incoming data until new confirmation. Nowadays all UARTs are able to handle data very quickly, so it's rarely used.

How to call `Setup()`:

```
_connection.Setup("COM3", 9600, (byte)'*', OnCOMOpen, OnCOMMessage, OnCOMError, OnCOMClose);
```

```
Connect()
public void Connect()
```

This method attempts to open the port, if it succeeds there is no connection establishment at all, it just means that you are able to send and receive data.

How to call `Connect()` to listen all incoming messages to port 60000 from any IP:

```
_connection.Connect();
```

```
public void Disconnect()
```

In `RS232Connection` it stops the listening thread instead of disconnecting. The entity remains ready to `Connect()` again.

```
public void Dispose()
```

This method disconnects the entity and also releases all internal data and resources. The entity remains unusable, a new instance should be created. Use this method when exit the application.

```
public bool DataAvailable()
```

If the `_onMessage` event is not set, then the incoming messages will be stored in an internal buffer. Use this method to know if there are incoming messages available.

```
public byte[] GetMessage()
```

If the `onMessage` event is not set, then the incoming messages will be stored in an internal buffer. Use this method to get the next available incoming message. If there are not available messages, an empty `byte[]` (`null`) will be returned.

```
public void ClearInputBuffer()
```

If the `onMessage` event is not set, then the incoming messages will be stored in an internal buffer. Use this method to flush the incoming messages buffer.

```
public void SendData(byte[] data)
public void SendData(string data)
```
Use this methods to send data. This two overloads allows to send data if the port is already open. The `string data` is always encoded to UTF8.

```
public bool IsConnected()
```
Use this method to check if the entity is currently listening.

```
public static string[] GetAvailablePorts()
```
This method returns the list of all available serial ports on the device. Please note that port names are different across all operating systems, so keep that in mind when exporting to several platforms.
The constructor and `Setup()` verifies internally the availability of the provided port using this method.
NOTE: This method is static, so you should call the base class to get access to it.

How to call `GetAvailablePorts()` to get the list of available ports:
```
string[] ports = RS232Connection.GetAvailablePorts();
```

```
public string ByteArrayToString(byte[] content)
```
Converts a `byte[]` into a `string` using UTF8 encoding.

```
public byte[] StringToByteArray(string content)
```
Converts a `string` into a `byte[]` using UTF8 encoding.

```
public int SecondsToMiliseconds(float seconds)
```
Converts a `float` in seconds into an `int` in milliseconds. Useful when working with timers (`System.Threading.Timer`).

RS232Connection events

All events are invoked as they happen without delays, and all events always runs in a new thread to grant application responsiveness and push at maximum the device performance. Keep this in mind, because it may be an issue when working with some sort of graphic interface as Unity, Windows Forms, Xamarin, etc. As usual, always keep the code of the events as short as possible.

```
public volatile EventVoid onOpen
```
This event is invoked when the `RS232Connection` is open starts listening. If any problem occurs and the listening can't be started, then a `onError` event is invoked instead.
The signature is: `void OnOpen(RS232Connection)`
The `RS232Connection` is the reference to the entity which invoked the event.

```
public volatile EventMessage onMessage
```
This event is invoked when a complete message has been received.
The signature is: `void OnMessage(byte[], RS232Connection)`
The `byte[]` is the received message.
The `RS232Connection` is the reference to the entity which invoked the event.

```
public volatile EventException onError
```
This event is invoked when an internal error has occurred. Error may be fired for internal misbehavior (self descriptive in most cases) or for the code assigned to the `onMessage` event (otherwise you'll be not able to debug your multi-threaded code), which means your own custom code.
The signature is: `void OnError(int, string, RS232Connection)`
The `int` is the error code.
The `string` is error description.
The `RS232Connection` is the reference to the entity which invoked the event.

```
public volatile EventVoid onClose
```

This event is invoked when the connection closes unexpectedly. In most cases it will never be fired unless a severe hardware problem is found.
The signature is: `void OnClose(RS232Connection)`
The `RS232Connection` is the reference to the entity which invoked the event.

Quick reference

This piece of code is included in the `UDPConnection` header to ease the task of defining the events. Copy to your code, complete with what you need and assign to the delegates:

```
    void OnCOMOpen(RS232Connection connection)
    { }
    void OnCOMMessage(byte[] message, RS232Connection connection)
    { }
    void OnCOMError(int code, string message, RS232Connection connection)
    { }
    void OnCOMClose(RS232Connection connection)
    { }
```

Events are assigned like this:

```
    _connection.onOpen = OnCOMOpen; // Use this if you need only one call.
    _connection.onOpen += OnCOMOpen;// Use this if you need several calls.
```

NTP_RealTime

Here is the complete description of the `NTP_RealTime` and how to use it. This is a static class, so there is only one instance by application.
This class is designed to be thread-safe, so its status can be checked at any moment without restrictions.
There are two events provided to control the status of this class, which are described below.

NTP_RealTime parameters

The parameters can be checked at any moment, this is the main way of interacting with this class.

```
public static DateTime _now
```

This parameter is read-only and returns the internal clock converted to local time, depending on the OS settings. Make sure to check **`_requestState`** to know if the internal clock is synchronized or after the first **`onSync`** event invocation.

```
static public RequestState _requestState
```

This parameter returns the status of the internal clock synchronization. It's value is determined using the `RequestState` `enum` and its values are:

- **`Default`**: It just started, needs synchronization.
- **`Waiting`**: Request was sent, but no response for now.
- **`Ready`**: NTP response received and synchronized.
- **`Error`**: The error description is in **`_errorMsg`**.

```
static public string _errorMsg
```

This parameter contains the description of the error when the **`_requestState`** is **`Error`**.
When an error occurs, the **`onError`** event is also fired.

```
static public UDPConnection _udpRepeater
```

This is the UDP repeater used to receive requests and repeat the synchronization process to the main NTP remote server. When the NTP response arrives, the local clock is also updated.
This entity is left `public` in order to check the status of the connection, not to be manipulated.

```
static public TCPServer _tcpRepeater
```

This is the TCP repeater used to receive requests and repeat the synchronization process to the main NTP remote server. When the NTP response arrives, the local clock is also updated.
This entity is left `public` in order to check the status of the server, not to be manipulated.

```
static public WSServer _wsRepeater
```

This is the WebSockets repeater used to receive requests and repeat the synchronization process to the main NTP remote server. When the NTP response arrives, the local clock is also updated.
This entity is left `public` in order to check the status of the server, not to be manipulated.

NTP_RealTime methods

The methods are provided to keep control of the entity.

```
static public DateTime GetUTCTime()
```

This method returns the internal UTC clock. Check the **`_requestState`** to know if this clock is synchronized or wait for the **`onSync`** event. If the internal clock is not synchronized (at least once), then the system time is returned (the system time is not always accurate, but it keeps your application running).

```
static public void SendUDPRequest(string ntpServer = "", int port = 0)
```
This method sends an NTP request using UDP. If the `ntpServer` and `port` parameters are provided, then `NTP_RealTime` class will save them as the new default NTP server for future use.
This method can also be used to send a request to a repeater/emulator.

How to request synchronization to the default server:
```
NTP_RealTime.SendUDPRequest();
```

```
static public void SendTCPRequest(string ntpServer, int port)
```
This method sends an NTP request using TCP. The NTP request is exactly the same as the one sent through UDP, also is the response.

How to request synchronization to a TCP repeater/emulator:
```
NTP_RealTime.SendTCPRequest("192.168.0.10", 60010);
```

```
static public void SendWSRequest(string ntpServer)
```
This method sends an NTP request using WebSocket. The NTP request is exactly the same as the one sent through UDP, also is the response.

How to request synchronization to a WebSocket repeater/emulator:
```
NTP_RealTime.SendWSRequest("ws://192.168.0.10:60012/ntp/");
```

```
static public void StartUDPRepeater(int port, string localIP = "", bool ntpEmulation =
       false)
```
This method starts the local UDP repeater.
The `port` should be provided, and must be available to receive incoming requests (not already in use and open in the firewall). The `localIP` can be empty (`""`) for automatic configuration, or contain the literals `"ipv4"` or `"ipv6"`, or contain an available IPV4 or IPV6 address.
The `ntpEmulation` parameter enables this repeater as an NTP server responding from the internal synchronized clock (or the system clock, if it's not synchronized).
If the emulation is disabled, all incoming NTP requests will be repeated using the default NTP server (UDP) saved internally. The default `"time.windows.com"` server will be used if none was provided through `SendUDPRequest()` (Chose the one that works better in your region/country).

How to start the local UDP repeater using available IPV4 and IPV6 automatically and emulation:
```
NTP_RealTime.StartUDPRepeater(60010, "", true);
```

```
static public void StopUDPRepeater()
```
This method stops the UDP repeater and releases all its resources.

```
static public bool IsUDPRepeaterRunning()
```
This method is useful to easily query if the UDP repeater/emulator is active.

```
static public void StartTCPRepeater(int port, string localIP = "", bool ntpEmulation =
       false)
```
This method starts the local TCP repeater/emulator.
The `port` should be provided, and must be available to receive incoming requests (not already in use and open in the firewall). The `localIP` can be empty (`""`) for automatic configuration, or contain the literals `"ipv4"` or `"ipv6"`, or contain an available IPV4 or IPV6 address.
The `ntpEmulation` parameter enables this repeater as an NTP server responding from the internal synchronized clock (or the system clock, if it's not synchronized).
If the emulation is disabled, all incoming NTP requests will be repeated using the default NTP server (UDP) saved internally. The default `"time.windows.com"` server will be used if none was provided through `SendUDPRequest()` (Chose the one that works better in your region/country).

How to start the local TCP repeater using available IPV4 and IPV6 automatically and emulation:

```
NTP_RealTime.StartTCPRepeater(60011, "", true);
```

```
static public void StopTCPRepeater()
```

This method stops the TCP repeater and releases all its resources.

```
static public bool IsTCPRepeaterRunning()
```

This method is useful to easily query if the TCP repeater/emulator is active.

```
static public void StartWSRepeater(string address, bool ntpEmulation = false)
```

This method starts the local WebSocket repeater/emulator.
The `address` should be provided, and must be available to receive incoming requests (open in the firewall and not already in use).
The `ntpEmulation` parameter enables this repeater as an NTP server responding from the internal synchronized clock (or the system clock, if it's not synchronized).
If the emulation is disabled, all incoming NTP requests will be repeated using the default NTP server (UDP) saved internally. The default `"time.windows.com"` server will be used if none was provided through `SendUDPRequest()` (Chose the one that works better in your region/country).

How to start the local WebSocket repeater with emulation:

```
NTP_RealTime.StartWSRepeater("http://192.168.0.10:60012/ntp/", true);
```

```
static public void StopWSRepeater()
```

This method stops the WebSockets repeater and releases all its resources.

```
static public bool IsWSRepeaterRunning()
```

This method is useful to easily query if the WebSocket repeater/emulator is active.

```
static public void Dispose()
```

This method stops all current requests and repeaters releasing all resources. Use this method when exiting the application or when loading a new scene in Unity.

```
static public IPEndPoint AddressParser(string ipOrUrl = "ipv4", int port = 0)
```

Converts the provided data into a `System.Net.IPEndPoint` object.
If the `"ipv4"` or `"ipv6"` literals are provided as `ipOrUrl` argument, the `IPEndPoint` will be generated using the default IPV4 or default IPV6 respectively.

```
static public string GetDefaultIPAddress(string ipMode = "")
```

This method returns the first available IPV4 or IPV6 address of your device.
The `ipMode` argument also allows the `"ipv4"` and `"ipv6"` literals to force the return.

```
static public string[] GetMacAddress()
```

Use this method to list all the MAC addresses of your device.

NTP_RealTime events

All events are invoked as they happen without delays, and all events always runs in a new thread to grant application responsiveness and push at maximum the device performance. Keep this in mind, because it may be an issue when working with some sort of graphic interface as Unity, Windows Forms, Xamarin, etc. As usual, always keep the code of the events as short as possible.

```
static public volatile Event onSync
```

This event is invoked when the `NTP_RealTime` receives a valid synchronization message by any channel. Once a valid synchronization message is received, the `NTP_RealTime` uses its internal clock

until a new valid message is received, and the internal clock is updated. In default state it returns the system time (not recommended when synchronizing your internal network).
The signature is: `void OnSync()`

```
static public volatile EventException onError
```

This event is invoked when an internal error has occurred. If any errors occurred, the internal synchronized clock is not modified, and keeps the last valid synchronization value.
This event can also be fired if there is an error in the custom code assigned to the `onSync` event.
The signature is: `void OnError(int, string)`
The `int` is the error code.
The `string` is the error description.

Quick reference

This piece of code is included in the `NTP_RealTime` header to ease the task of defining the events. Copy to your code, complete with what you need and assign to the delegates:

```
    // NTP:
    void OnNTPSync()
    { }
    void OnNTPError(int code, string message)
    { }
```

Events are assigned like this:

```
    NTP_RealTime.onSync = OnNTPSync; // Use this if you need only one call.
    NTP_RealTime.onError += OnNTPError;     // Use this if you need several calls.
```

Known Issues

- WebSocket entities requires `.net` 4.6 or later.
- Not all platforms supports all connections and servers, please check the compatibility table.
- There is no way to send UDP messages to a TCP or WebSocket entity, messages must be interchanged between entities of the same kind.
- When the network becomes unreachable due to an infrastructure failure (connector unplugged, router turned off, etc.) there is no onClose event, so the "keepAlive" timer and the "Watchdog" will detect inactivity and force this event in both sides of the line.

Contact

If you need some help or if you find some errors in this documentation or the asset, or you just want to give some feedback, don't hesitate to contact me to: jmonsuarez@gmail.com

Once you have used this asset, please take a few minutes to write a good review in the Unity Asset Store so you'll be helping to improve this product.

https://assetstore.unity.com/packages/tools/network/sockets-under-control-159512

Thanks.