Sockets Under Control

“...They lie eternally into the depths of the operating system, awaiting with low latency all new incoming connections...”

- Lumbreras

(eToile 2021) V: 1.4

Index

Introduction

Welcome to the Sockets Under Control documentation, secretly known since ancient ages as the *Networknomicon* (translation from the Sumerian “Durmab-Kasaru” what means “Establishing connections”).

This arcane knowledge will give you the power to control the three main forces of networking:

1. UDP.
2. TCP.
3. Websockets.

There are basically five entities to tame:

1. UDP client/server.
2. TCP client.
3. TCP server.
4. WebSocket client.
5. WebSocket server.

Many lives has been lost in the search for the perfect abstraction and encapsulation, all the knowledge and experience collected by those brave ancient ones are joined in this *Networknomicon*. All five entities are invoked in similar ways, so once you know one, you almost know all of them.

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.
NOTE: Multi-threading is not supported by actual web browsers, this affects `WSConnection` on WebGL.

All five beasts are able to connect through the Internet, all limitations you may find is due to the Internet nature and structure (The world is still evolving from IPV4 to IPV6, which implements a slightly bigger MTU allowing faster and bigger messages).

Description

This package contains five main core classes:

1. UDPConnection.cs: UDP works as client and server simultaneously.
2. TCPConnection.cs: TCP client, it needs a TCP server to send and receive data.
3. TCPServer.cs: Receives connections from TCP clients.
4. WSConnection.cs: WebSocket client, requires a WebSocket server to connect to.
5. WSServer.cs: Receives connections from WebSocket clients.

The core classes are designed to work on any C# compatible platform, not only in Unity.
NOTE: The `WSConnection` core class uses a plugin to be supported on WebGL. This particular case has partial asynchronous functionality (Web browsers doesn’t allow multi-threading).

You need advanced knowledge on multi-threading to be able to use the core classes directly into Unity, because the whole application must be designed in a thread-safe way.

Every core class has its own Unity wrapper. They fixes most of the multi-thread problems, because Unity is not thread safe, easing significantly the setup and development.

The use of the included Unity wrappers is highly recommended if you suffer from multi-thread programming phobia and you want to preserve your sanity.

The corresponding wrappers for the core classes are:

1. UnityUDPConnection.cs: Wrapper for `UDPConnection`.
2. UnityTCPConnection: Wrapper for `TCPConnection`.
3. UnityTCPServer: Wrapper for `TCPServer`.
4. UnityWSConnection: Wrapper for `WSConnection`.
5. UnityWSServer: Wrapper for `WSServer`.

Every network protocol will be explained separately, and the associated core classes are explained in the advanced documentation.

Several entities (connection/server) can be used in the same application, there is no limit in this aspect. Servers has a configurable maximum available incoming connection to help catching brute-force attacks (there is no fixed limit in simultaneous incoming connections).

Asset Integration

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

Once you get familiarized with the package, you can safely delete the following folders:

- eToile/ExampleShared
- eToile/SocketsUnderControl/Example

Sockets Under Control includes FileManagement, which has some useful features intended to help in the design of your custom communication protocols.

Deleting the examples you'll reduce your project weight. Of course the examples are not being included in the final build even if they are not deleted.

You can import several *eToile* products in the same project, all of them are designed to work together without issues.

Into the eToile/SocketsUnderControl folder you'll also find some other assets:

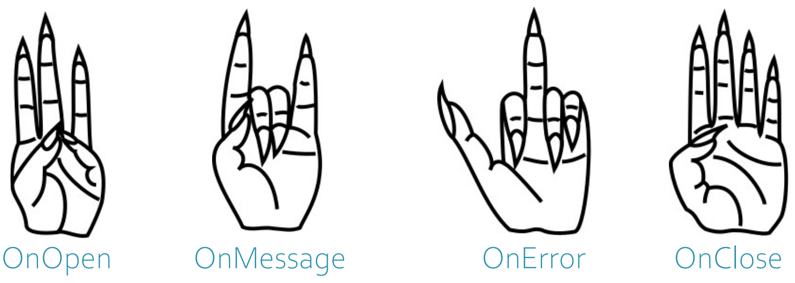
- CoreClasses/WSConnection.jslib: The web browser compatible version of WSConnection.cs.
- CoreClasses/WSClient.cs: This class is a custom client to be used internally by WSServer.
- SUC_WinForms.zip: Windows Forms example project for Visual Studio to test all core classes. This project is also compatible with Linux using Monodevelop.

Sockets Under Control philosophy

All five networking entities are divided in two main groups:

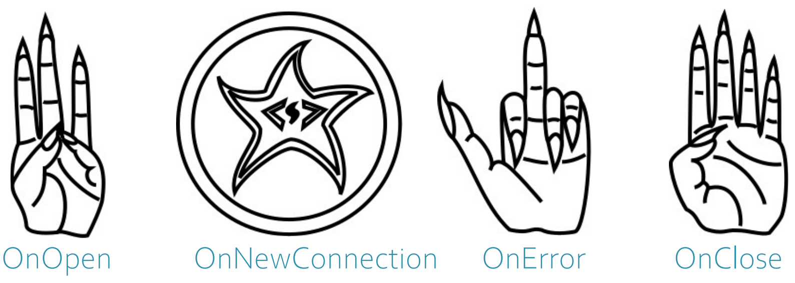
1. Servers: They lie eternally into the depths of the operating system, awaiting with low latency all new incoming connections. They can interchange messages with any connected client, but they are unable to establish a new connection by themselves. The client must always start the connection process.
2. Clients: They connect to a remote server of its same kind, then they are able to send and receive messages (The only exception to this rule is UDP, which doesn't has servers).

All clients has four and only four events:



The events will be extensively described in their respective entity sections. Every entity has its own specific signature for these events, but all of them contains the exact same relevant information.

All servers have also four events only:



Servers can't receive messages directly, so the OnMessage event is replaced with the OnNewConnection event. This particular event returns a local version of the incoming connection. This new connection is the gate that allows a server to interchange messages with a specific client. The returned new connection therefore has the four events as any other connection (in this case the OnOpen event is useless because the connection is already established).
The servers maintains a dynamic list of all active connections in real time. This list can be accessed at any time and from any thread.

All incoming and outgoing messages are binary data only, but there is no reason to be afraid, any data type can be converted to binary and back: This process is called "serialization".
Sending binary data may also reduce the size of data fields (in some cases), increasing the amount of data that can fit in a single message. Thus you can optimize messages if you own the necessary skills (boolean algebra indeed). Sending a `string` like "true,false,true,true" containing 4 flags consumes a total of `20` `byte` while sending this same information as binary `[00001011]` consumes `1` `byte` only (`0x0B`).

This philosophy allows also to send data including special characters without risk. Think about a text file that includes “End Of Line (EOL)” characters: when using any common stream networking system, the message will arrive fragmented, the receiver will divide the incoming data in several messages, one for each encountered EOL character (`"\n"` or `"\r\n"` depending on the OS).
You should have this in mind specially if you are designing your own protocols using TCP.

You can encapsulate several fields of data using the most common data serializers too, as XML, JSON, INI (the last one is included in FileManagement) or your own file structures, then send the binary encoded data safely.

As a normal rule you must consider sending packages of a maximum size of ~64KB, with this you’ll be able to port your protocols from UDP to TCP to WS almost in a transparent way.
In fact TCP and WS allows bigger packages, but they will cause fragmentation. This rule is also affected by MTU. Both evil forces of fragmentation are described later in this chapter.

The first step to optimize communications is to make sure that all the relevant information is included within those ~64KB (for simple local networks), the message will be send immediately, and once received at the other side, the OnMessage event will be fired immediately too.

NOTE: When working through the Internet (including VPN) you may find a major message size limitation due to the smallest MTU (Maximum Transmission Unit). This limitation affects mostly `UDPConnection`, and unfortunately it can’t be fixed. The smallest MTU size depends on the Internet infrastructure of your country/provider (1,4KB normally for IPV4), and not on your local setup or available buffer (Your network adapter has its own MTU, yes, but it’s rare to be the case). The intermediary routers will try to “fragment” the Datagrams, in TCP and WebSocket they will be restored transparently, but in UDP they’ll always fail to be restored on the other side (in fact, they’ll never arrive). If you find this issue, keep your messages under 1,4KB (way less than 64KB, yes) and you’ll be fine.

NOTE: Fragmentation is performed automatically (by the low level network protocol) when the network detects that you are trying to send a package bigger than it allows, either by MTU, target input buffer, etc. The size of the package is not fixed and may vary from fragment to fragment. Fragmentation is handled seamlessly by `WSConnection`, but TCP don’t. Fortunately `TCPConnection` provides the option of an internal recovery buffer, but a termination character/sequence is required in order to work properly (this affects the way you design your custom protocols, normally the EOL character is used). Just as a reminder, `UDPConnection` can’t handle fragmentation at all.

You can always check the available buffer size from any connection at run-time.
If you have to send data bigger than 64KB the best way to send it is to divide in chunks and send them separately, then joining the received messages in the target device again using your own methods. Doing this you can follow the progress of the transference, otherwise it will just take a single event when it’s finished.

To set any entity to work, follow the next steps:

1. Local address (IP, URL, Port, etc.) to chose the desired local network adapter. Most options here are automatically chosen.

2. Remote address (IP, URL, Port, etc.) to chose the target remote entity/device (servers don't require this data).
3. Set the 4 events before attempting the connection.
4. Connect.

When a new connection is received on any server, you should set its events immediately (just as in step 3) and then start interchanging data with the remote device.

`TCPConnection` and `WSConnection` retries the connection if it's not established at once (This feature is not available on WebGL).

You can Disconnect() and Connect() at any moment too, no need to Dispose() the connection and instance a new one, allowing to easily switch Setup() parameters in between.

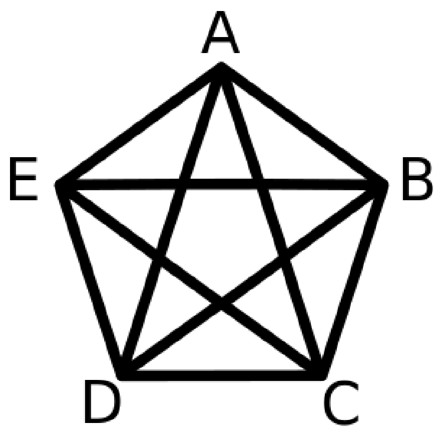
About UDP and its powers

The UDP protocol (User Datagram Protocol) is the most simple and underrated communication protocol.

The main difference between this protocol and the others is that it doesn't cares about establishing a connection, so it can send packages and forget about them, it doesn't matter if they arrive or not.
It is called "unreliable" (and it's associated with a bad thing) but far from being unreliable it's useful for specific purposes. Use it wisely.

The `UDPConnection` measures the maximum available buffer size at Setup() (this is OS and hardware dependent, unfortunately not available on Apple platforms).
This process grants the maximum output buffer available (~64KB or less) to be visible, any UDP message bigger that the output buffer will be automatically dropped without prompt.

Due to its nature a `UDPConnection` entity works as client and server at the same time, so the "server/client" concept is useless in its domains. The messages can be dispatched in a topology called "Mesh" what means: anyone can send/receive messages to/from anyone, without restrictions.



The A to E devices can communicate to each other without intermediary servers. This allows faster communications (compared to TCP and WebSockets) but there is no warranty of message delivery, so caution is advised.

In order to send and receive messages, UDP needs a fixed addressable IP, this is not an issue in local networks but can be a real problem through the Internet (specially when using IPV4, they tend to change very often).

In practice you'll notice that there are no problems in sending messages to a remote "fixed IP server" with UDP, but there is no physical way to address the origin IP through the internet router/modem (it uses a different public IP in most cases, the one of the internet router, which changes dynamically), so the "fixed IP server" can't reply back directly, here the router works as a barrier.

This is the situation when a NAT traversal solution is required, but unfortunately, in practice NAT traversal has not warranty of success. All existing methods depends on the client side NAT implementation (NAT is not standard at all), and also on the network security systems.

Using NAT traversal for UDP is just like trying to hack the client network (using brute force with several methods in parallel, hoping one of them will work).

The "good news" is that there is a "correct" way to get through this NAT, and works in most cases. As It was mentioned before, you'll find no problems in sending a message to the remote internet device/server, then the NAT server (included with all Internet provider's routers nowadays) will save the relation between the local IPV4 and the public IPV4 during the current session (In other words: while your local device is still connected to your router).

So this feature allows the remote internet device/server to send a message to the public IPV4 instead of the local IPV4, and the local device will receive the message thanks to the NAT re-routing.

The bad news is that only the first device which started the communication will receive the incoming messages, so if you plan to have two local devices interchanging data with the same remote internet device/server simultaneously, only the first who sent some data will be able to receive the incoming messages (the other one will remain blinded). It's logical, there is only one public IPV4 to receive all incoming messages and there is no way for the NAT to route both local devices separately.

This problem will not happen when a "fixed IP server" sends a message to another "fixed IP server" and back (this case works with IPV4 and IPV6 addresses) because it behaves as a local network. Fortunately `UDPConnection` allows IPV4 and IPV6 addresses simultaneously, so local IPV6 can be dynamically detected and used if available.

The definitive solution to this problem is IPV6. It allows all possible combinations, but the use of this system is not widely supported (at least it's standard and will work for sure).
When using IPV6 just make sure to send a message from the client to the "fixed IP server" first, then start interchanging messages, and the server will be able to respond directly.
The remote internet device/server can target the local IPV6 directly, allowing multiple connections simultaneously without using any weird NAT hacking.

Finally we reached the Peer to Peer (P2P) magic: it's when a local device sends UDP data to another local device in another remote network through the Internet, without the intervention of any intermediary server.
The first step is that both devices should send any data to the same remote internet device which works as a "Lobby" just to establish the connection. This remote internet device/server will then know the public IPV4 address of both devices (their internet routers specifically), then the IPV4 addresses can be interchanged to each other. Now both devices can send UDP data to the public IPV4 of each other through the Internet, establishing some kind of direct connection without passing through any intermediary server.
Those public addresses will change, so the process of establishing this connection should be made every single time. As you can see, there is no way to avoid the use of a remote server.

This is how WebRTC works (it's UDP, yes), but unfortunately it's not yet widely supported.

Most common uses for UDP is video/audio streaming, network discovery/mapping, low latency game synchronization, NTP, etc. In short: any quick and constant information that can be updated, or restored, or synchronized with the next incoming message in case that the last one was lost or dropped.

As the `UDPConnection` doesn't establishes a connection dialog with the remote devices, there is a way to send a message to all listening devices in the local network, all at once, and this feature is called "Broadcasting".
This feature works because no confirmation is needed, and it's exclusive to UDP powers. Make sure to send small messages when using this feature (8KB maximum) or most devices will drop it without prompt (remember, UDP doesn't care if message arrives or not).
NOTE: Broadcasting works on IPV4 only.

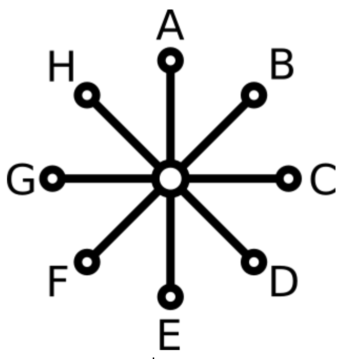
There are some other hidden powers:

- Once a `UDPConnection` is summoned, it can send messages to any port even in broadcast. There is no need to listen to that port, nor any. It also means that sending data in UDP is parallel to listening, so it can send data to any other port simultaneously.
- A broadcasted message (IPV4) is received by any listening `UDPConnection`, so use this feature to discover/map devices in the local network. Broadcast doesn't work through the Internet.
- There can be several devices sending data to the same IP/port, and all messages will be received by the same local `UDPConnection`. This is the only way to emulate a UDP server.
- If any remote device shuts down, the local connection will not close and the system can keep running, even if there are no other devices at all. It also allows other devices to join and leave the communication at any moment.
- The `UDPConnection` will check automatically the maximum size of available TX/RX buffer. This value can vary from device to device, but `UDPConnection` implements a safe way to verify this value at start. Normally this value is ~64KB for practical purposes.
- There are almost no possibilities to receive corrupted data because UDP implements a CRC32 verification signature at low level.
- UDP has no fragmentation, any bigger data will be just dropped. This is not a bad thing, because it grants integrity.

About TCP and its worshipers

The TCP protocol (Transmission Control Protocol) is a complex communication method, but time has proved its robustness through ages. It's the most used communication method in current days, so we can confirm that it rules them all.

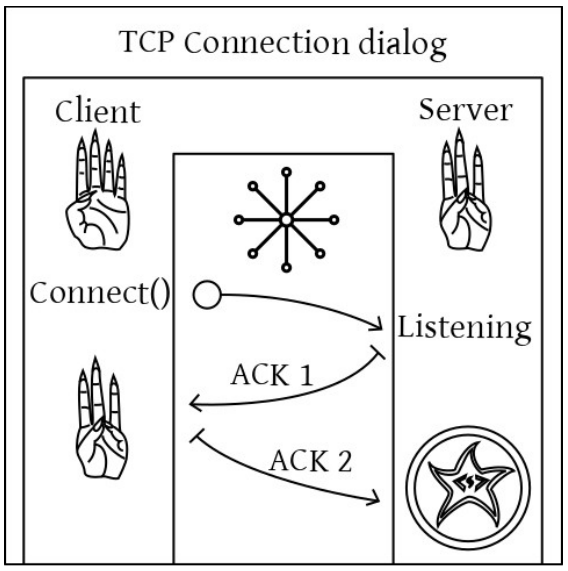
The TCP protocol uses the client/server philosophy. It's mandatory to have a main server in order to allow all clients to establish their connections. Servers can't connect to each other directly, clients can't connect to each other either.
The network infrastructure schema for TCP is called "Star":



The A to H devices connects with the main central TCP server using a single connection each.
The `TCPConnection` entity can use IPV4 or IPV6 depending on the provided remote server address. In this case the protocol is detected and selected automatically.

The `TCPServer` allows to accept incoming IPV4 and IPV6 connection simultaneously.

This is the procedure for a TCP connection to be established:



From the application perspective the dialog is invisible, the only interactions are the events.

The remote `TCPServer` maintains a dynamic `List<TCPConnection>` with all incoming connections, so the server knows the count of connected clients in real-time.

Clients cannot send TCP messages to each other, is the server who must route messages to other clients accordingly (if that's your application purpose) acting as a sort of bridge or distributor.

All messages are delivered with an internal low level dialog, so there is no confirmation event. In practice just assume that messages arrive in the same order they are sent in a synchronous way. This dialog ensures delivery and also allows bidirectional real time communications between local network clients and internet servers.

All common streamed TCP methods awaits until the output buffer is full or the timeout has expired to send the message (the "Nagle's algorithm" optimization is not very optimal anymore) delaying communications making TCP much slower. To prevent this, `TCPConnection` sends messages immediately, being optimal for real time communications. In practice, if any message fails to be delivered, it's retried until it succeeds so the only perceptible effect is the delay.

The streamed input in `TCPConnection` is enabled by default, that's the best approach for most cases and networks, specially when you don't have a complete knowledge of its underlying horrors. The streamed method will grant integrity in delivering messages, so disable it only if you have complete knowledge of your particular use case.

Incoming messages fires the OnMessage event immediately (excepting the case of fragmentation, which will wait until the full message is received).

NOTE: Neither `TCPConnection` nor `TCPServer` uses streams at all, the EOF detection is performed by a for loop on the input buffer, it may sound slower but it's not because it just don't have to wait for the stream to timeout.

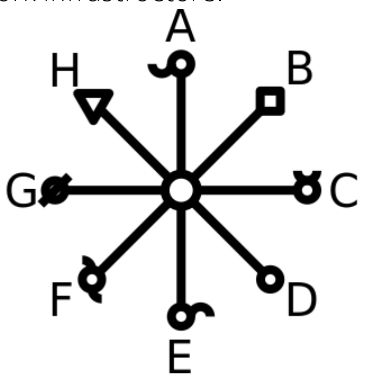
NOTE: Neither `TCPConnection` nor `TCPServer` uses streams at all, so there is no possibility to fire the "too many streams/files open" exception on any OS. This allows a massive number of simultaneous connections for `TCPServer`.

About WebSocket's supremacy

The WebSocket is the next step in the evolution of client/server networking, it claims the throne occupied by TCP nowadays.
It certainly behaves in the same way as TCP does (In fact it's a TCP network adding a new layer). The main aspect you'll find is that it works in web browsers (even in mobile devices) and is way better prepared against fragmentation.

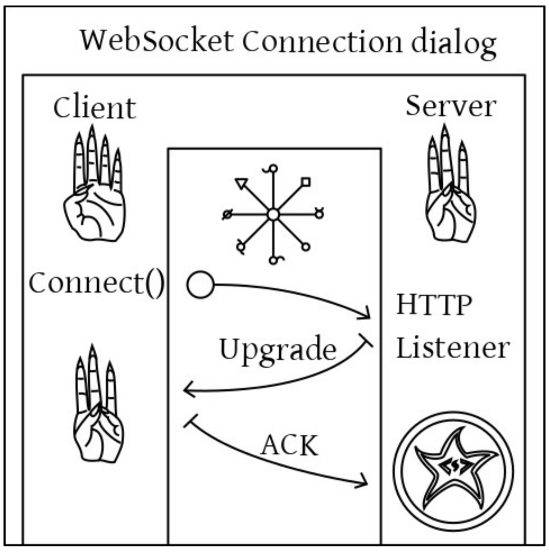
It uses the client/server architecture, so a `WSConnection` requires a `WSServer` to connect to, and a `WSServer` accepts several incoming connections at once. There is no limit for incoming connections.

All WebSockets uses the "Star" network infrastructure:



Both `WSServer` and `WSConnection` supports IPV4 and IPV6 simultaneously.

This is the procedure for a WebSocket connection to be established:



From the application perspective the dialog is invisible, the only interactions are the events.

If you need high compatibility level in your network application think on using WebSockets, it's the most supported among the three networking forces.

The only downside is that it's slower than TCP (about 10% slower).

Keeping connection alive (to sacrifice them later)

In some environments, servers requires a message to be sent every 30 seconds or less, If this requirement is not met, then those connection will be sacrificed.

This is the "keepAlive" message, which Sockets Under Control implements as an empty message that is send every **`keepAliveTimeout`** seconds to the server (or to the network in the case of `UDPConnection`).
The Sockets Under Control servers will automatically reply to this "keepAlive" message with another "keepAlive" message to let know the client that its server is connected and active.
NOTE: The `WSConnection` running on Mono is the only entity that implements this feature natively.

Both servers replies to the "keepAlive" message always, but the timeout control can be disabled by setting the **`keepAliveTimeout`** to **`0f`**. Doing this, the server will never detect if a client is inactive. The activity in server side is measured by any received message, so if you are using some non SUC client, you can send anything and it will be detected too.

All clients send a "keepAlive" message every **`keepAliveTimeout`** seconds, and this feature can be disabled setting its value to **`0f`** (it also disables the watchdog described below).
This timeout also serves as an interval to probe the server activity. If the client detect that there is no server activity, then the connection will be automatically closed. This feature can also be disabled independently by setting the **`disableWatchdog`** flag (keeps sending "keepAlive", but doesn't disconnects if not replied).

As default values, the server uses a timeout of 40 seconds while the client uses a timeout of 15 seconds (what is almost the half). The PING-PONG dialog must fail 3 times to validate the inactivity (in some .net WS clients a PING message may be get missed randomly).

This feature is very useful in case of network instability (It doesn't means slow networks or applications closing unexpectedly, it means hardware issues) allowing to detect that something is wrong between the client and the server and nobody knows what happened.
In this situation the only thing that can be done is close the connection and try to connect again (of course for games it's not often needed, but in industrial environments it allows the application to fix itself automatically without the need of user intervenction).

NTP servers lurking around

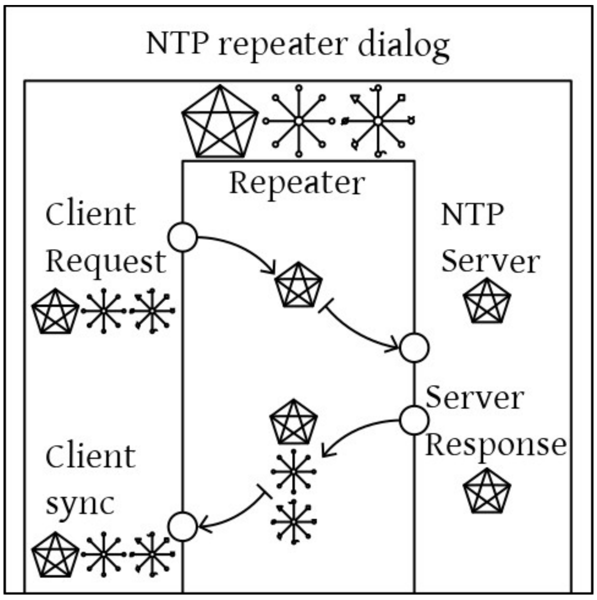
The NTP (Network Time Protocol) servers are there, everywhere, even if you can't see them.
These servers are a simple way to get the current world time (UTC or GMT0), and it's used to synchronize the local clock. This protocol is the same used by the OS to correct the system time.

The most important feature of this servers is to prevent time manipulation. The best NTP servers are attached to an atomic clock, that's why it has a high precision (The `"time.windows.com"` NTP server is used by default for main synchronization, but you can use any other if it's slow in your country).

Unfortunately NTP servers are UDP only entities, so there is no way to get time synchronization through TCP or WebSockets in this plane of the multiverse. Sockets Under Control implements a solution to help with this problem: The repeaters and emulators.

Sockets Under Control provides a static class called `NTP_RealTime` to manage this protocol and latency mitigation automatically. This class also provides methods to request the synchronization through the main three forces of networking (Check the complete description on the advanced documentation).

The repeaters are just simple implementations of UDP, TCP and WebSocket servers what repeats the incoming requests to the main UDP request protocol, then repeats the NTP server response to the device who has requested in first place:



The `NTP_RealTime` class implements the three repeaters separately, and they can be enabled or disabled independently too (please don't confuse NTP repeaters with NTP servers).
The repeaters has the option to work as NTP server emulators too, returning the local synchronized time if available (they return the system time otherwise, not the same accuracy but it works).

In your applications just use `NTP_RealTime._now` instead of `System.DateTime.Now` to get the time.
There are two events available: `onSync` and `onError` (Check the advanced documentation).

NOTE: `NTP_RealTime` does not updates the system clock, it maintains its own time internally. This feature prevent the user from cheating by system time manipulation.

Compatibility table

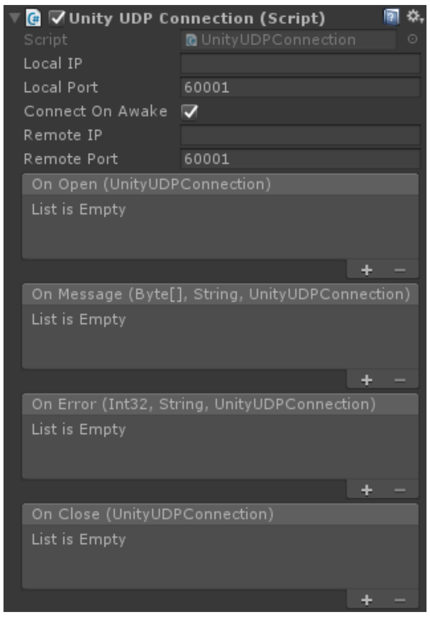
This table presents all supported platforms for each entity:

Unity	UDP	TCP	TCP-S	WS	WS-S
Windows	☑	☑	☑	☑	☑
Mac	☑	☑	☑	☑	☑
Linux	☑	☑	☑	☑	☑
iOS	☑	☑	☑	☑	☑
Android	☑	☑	☑	☑	☑
WebGL	NA	NA	NA	☑	NA
Bridge.net	NA	NA	NA	☑	NA

The core classes can also be compiled in Visual Studio, Xamarin and MonoDevelop.

UnityUDPConnection

Here is the complete description of the `UnityUDPConnection` wrapper and how to use it in Unity.



Unity UDP Connection (Script)

Script	UnityUDPConnection
Local IP	
Local Port	60001
Connect On Awake	✓
Remote IP	
Remote Port	60001

On Open (UnityUDPConnection)
List is Empty

On Message (Byte[], String, UnityUDPConnection)
List is Empty

On Error (Int32, String, UnityUDPConnection)
List is Empty

On Close (UnityUDPConnection)
List is Empty

The `UnityUDPConnection` is designed to be attached to a `GameObject` and all settings can be done in the editor too. The wrapper is enough for most application purposes.

You can modify the parameters of this entity dynamically, but they will take effect only once `Setup()` is called.

UnityUDPConnection parameters

Here is the complete description of all public parameters of this entity.

`public string _localIP`

This parameters allows to bind the `UnityUDPConnection` to a particular local IP pertaining to a given local Ethernet adapter (specially useful in case there are several) and can be set in many ways:

1. Empty: If the parameter is left empty, 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.

`public int _localPort`

This is the port to bind to, is the port used to send data by default. Accepts values between 1 and 65535.

`public bool _connectOnAwake`

If this flag is set, the `UnityUDPConnection` will `Setup()` and `Connect()` automatically at `Awake()`.

`public string _remoteIP`

This is the remote address to listen to (IPV4, IPV6 or URL), in UDP the remote IP can be set to filter all incoming messages in order to accept those coming from a specific remote device.
Left this parameter empty in order to receive messages from all IP addresses.

`public int _remotePort`

This is the port that the `UnityUDPConnection` will be listening to. This port can be different from the `_localPort`.

`public float _keepAliveTimeout`

This is the time in seconds to send a "keepAlive" message to the network.

UnityUDPConnection events

All events are stacked in the same order they occur, and then they are invoked on each `FixedUpdate()` loop (the whole FIFO stack is invoked progressively). This way it's able to execute Unity code in good synchronization avoiding thread issues.
As usual, always keep the code of the events as short as possible to grant responsiveness and fluidity.

`public BaseEvent _onOpen`

This event is invoked when the `UnityUDPConnection` 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(UnityUDPConnection)`
The `UnityUDPConnection` is the reference to the entity which invoked the event.

`public MessageEvent _onMessage`

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

`public ErrorEvent _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, UnityUDPConnection)`
The `int` is the error code.
The `string` is error description.
The `UnityUDPConnection` is the reference to the entity which invoked the event.

`public BaseEvent _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(UnityUDPConnection)`
The `UnityUDPConnection` is the reference to the entity which invoked the event.

Quick reference

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

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

UnityUDPConnection methods

The methods are provided to keep control of the entity.

`public void Setup()`

This method prepares the entity to be ready to connect, applying all the provided parameters. Always call `Setup()` before attempting to `Connect()` the first time. The `UnityUDPConnection` allows to send data once `Setup()` is called, but it will not receive until `Connect()` succeeds.

`public void Connect()`

This method attempts to establish the connection. In `UnityUDPConnection` it means that the entity starts listening and receiving messages. Always call `Setup()` before attempting to `Connect()`.
The connection will fail if the provided `_localIP`+`_localPort` pair is already in use by another entity.

`public void Disconnect()`

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

`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(string ip, byte[] data, int port = 0)
public void SendData(string ip, string data, int port = 0)
```
Use this method to send data. This two overloads allows to send data to a specific remote IP+Port. If the port is not provided the `_remotePort` 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 `_remoteIP+_remotePort` device automatically. Will do nothing if no `_remoteIP` was provided at `Setup()`. 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 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 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.

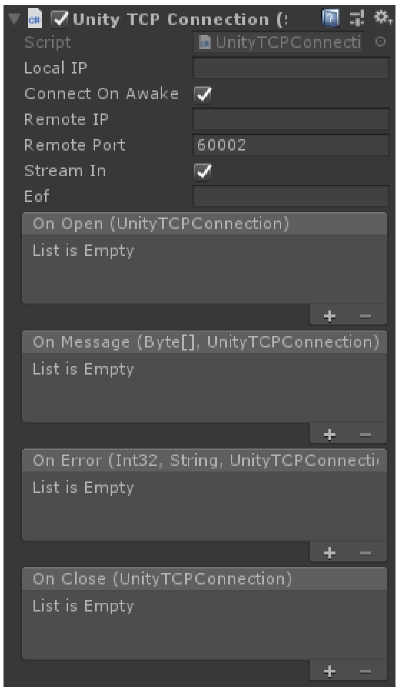
```
public string ByteArrayToString(byte[] content)
```
Converts a `byte[]` into a `string` using UTF8 encoding.

```
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`).

UnityTCPConnection

Here is the complete description of the `UnityTCPConnection` wrapper and how to use it in Unity.



```
Unity TCP Connection (:
Script              UnityTCPConnecti
Local IP
Connect On Awake    ✓
Remote IP
Remote Port         60002
Stream In           ✓
Eof
On Open (UnityTCPConnection)
List is Empty
                                  +  −
On Message (Byte[], UnityTCPConnection)
List is Empty
                                  +  −
On Error (Int32, String, UnityTCPConnecti
List is Empty
                                  +  −
On Close (UnityTCPConnection)
List is Empty
                                  +  −
```

The `UnityTCPConnection` is designed to be attached to a `GameObject` and all settings can be done in editor too. The wrapper is more than enough for most application purposes.

You can modify the parameters of this entity dynamically, but they will take effect only once **`Setup()`** is called.

UnityTCPConnection parameters

Here is the complete description of all public parameters of this entity.

`public string _localIP`

This parameters allows to bind the `UnityTCPConnection` to a particular local IP pertaining to a given local Ethernet adapter (specially useful in case there are several) and can be set in many ways:

1. Empty: If the parameter is left empty, the entity will automatically bind to the fist default IPV4 or IPV6 address (will automatically chose the same **`_remoteIP`** type).
2. IPV4: Binds to provided local IPV4 address, this IP must exist in the system or will fail. The **`_localIP`** type must match the **`_remoteIP`** type.
3. IPV6: Binds to provided local IPV6 address, this IP must exist in the system or will fail. The **`_localIP`** type must match the **`_remoteIP`** type.
4. URL: Automatically gets the related IP through the available DNS and binds to.
5. `"ipv4"` literal: Binds to the first available IPV4 address.
6. `"ipv6"` literal: Binds to the first available IPV6 address.

`public bool _connectOnAwake`

If this flag is set, the `UnityTCPConnection` will **`Setup()`** and **`Connect()`** automatically at `Awake()`.

`public string _remoteIP`

This is the remote address to connect to (IPV4, IPV6 or URL). This parameter must be provided in order to establish the connection, or the `UnityTCPConnection` will be not able to find the server.

`public int _remotePort`

This is the port that the `UnityTCPConnection` will connect to. This port must be open in server side.

`public bool _streamIn`

This parameter enables the fragmentation restoration when receiving a message. This feature needs an `_eof` character in order to detect the end of the incoming message.

`public char _eof`

This character is needed to detect the end of an incoming message when the `_streamIn` flag is enabled. The default value for this character is `'\n'` which is the "New line" command (It works for Windows and Unix platforms). If this character is not set or the wrong one is set, the communications will be corrupted.

`public float _timeout`

This is the time in seconds to retry the connection when the server is not responding/accepting.

`public float _keepAliveTimeout`

This is the time in seconds to send a "keepAlive" message to the server. This is also the time to probe the server activity.

`public bool _disableWatchdog`

This flag disables the forced onClose of the connection when no activity is detected with the server.

UnityTCPConnection events

All events are stacked in the same order they occur, and then they are invoked on each `FixedUpdate()` loop (the whole FIFO stack is invoked progressively). This way it's able to execute Unity code in good synchronization avoiding thread issues.
As usual, always keep the code of the events as short as possible to grant responsiveness and fluidity.

`public BaseEvent _onOpen`

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

`public MessageEvent _onMessage`

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

`public ErrorEvent _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, UnityTCPConnection)`
The `int` is the error code.
The `string` is error description.
The `UnityTCPConnection` is the reference to the entity which invoked the event.

`public BaseEvent _onClose`

This event is invoked when the connection closes unexpectedly.
The signature is: `void OnClose(UnityTCPConnection)`

The UnityTCPConnection is the reference to the entity which invoked the event.

Quick reference

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

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

UnityTCPConnection methods

The methods are provided to keep control of the entity.

`public void Setup()`

This method prepares the entity to be ready to connect, applying all the provided parameters. Always call `Setup()` before attempting to `Connect()` the first time.

`public void Connect()`

This method attempts to establish the connection. Always call `Setup()` before attempting to `Connect()`. If the remote server is not available, the entity will retry the connection every few seconds indefinitely or until `Disconnect()` is called.
The connection will fail if the provided `_localIP`+`_localPort` pair is already in use by another entity.

`public void Disconnect()`

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

`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 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)`

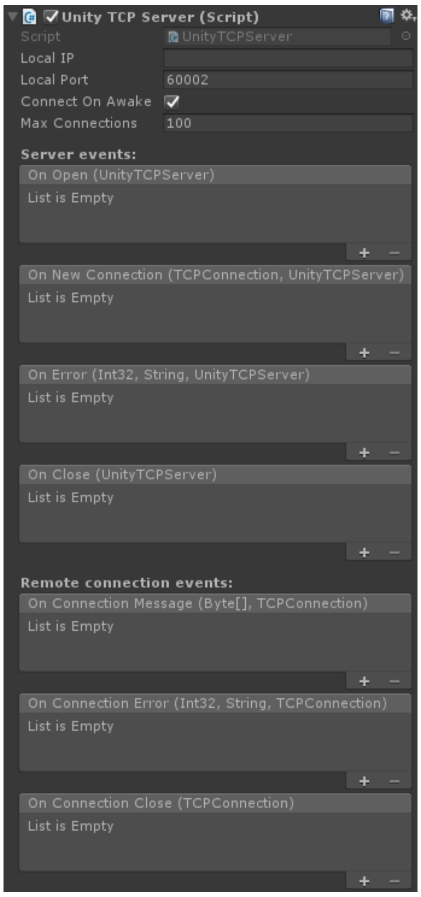
Converts a `string` into a `byte[]` using UTF8 encoding.

`public int SecondsToMiliseconds(float seconds)`

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

UnityTCPServer

Here is the complete description of the `UnityTCPServer` wrapper and how to use it in Unity.



Unity TCP Server (Script)

Script: UnityTCPServer

Local IP:

Local Port: 60002

Connect On Awake: ✓

Max Connections: 100

Server events:

On Open (UnityTCPServer)
List is Empty

On New Connection (TCPConnection, UnityTCPServer)
List is Empty

On Error (Int32, String, UnityTCPServer)
List is Empty

On Close (UnityTCPServer)
List is Empty

Remote connection events:

On Connection Message (Byte[], TCPConnection)
List is Empty

On Connection Error (Int32, String, TCPConnection)
List is Empty

On Connection Close (TCPConnection)
List is Empty

The `UnityTCPServer` is designed to be attached to a `GameObject` and all settings can be done in editor too. The wrapper is more than enough for most application purposes.

You can modify the parameters of this entity dynamically, but they will take effect only once **`Setup()`** is called.

UnityTCPServer parameters

Here is the complete description of all public parameters of this entity.

`public string _localIP`

This parameters allows to bind the `UnityTCPServer` to a particular local IP pertaining to a given local Ethernet adapter (specially useful in case there are several) and can be set in many ways:

1. Empty: If the parameter is left empty, 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.

`public int _localPort`

This is the port to bind to. This port must be open in the local firewall in order to receive incoming connections. Accepts values between 1 and 65535.

`public bool _connectOnAwake`

If this flag is set, the `UnityTCPServer` will `Setup()` and `Connect()` automatically at `Awake()`.

`public int _maxConnections`

This parameter sets the maximum allowed simultaneous incoming connections. All additional connections will be automatically rejected.

`public float _keepAliveTimeout`

This is the time in seconds to probe connection activity. Use two times the timeout set on clients.

UnityTCPServer events

All events are stacked in the same order they occur, and then they are invoked on each `FixedUpdate()` loop (the whole FIFO stack is invoked progressively). This way it's able to execute Unity code in good synchronization avoiding thread issues.
As usual, always keep the code of the events as short as possible to grant responsiveness and fluidity.

`public BaseEvent _onOpen`

This event is invoked when the `UnityTCPServer` 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(UnityTCPServer)`
The `UnityTCPServer` is the reference to the entity which invoked the event.

`public ConnectionEvent _onNewConnection`

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

`public ErrorEvent _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, UnityTCPServer)`
The `int` is the error code.
The `string` is error description.
The `UnityTCPServer` is the reference to the entity which invoked the event.

`public BaseEvent _onClose`

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

`public ConnectionMessageEvent _onConnectionMessage`

This event is invoked when a connection receives a new message. This event is assigned automatically to all new incoming connections.
The signature is: `void OnConnectionMessage(byte[], TCPConnection)`
The `byte[]` is the received message.
The `TCPConnection` is the reference to the entity which invoked the event.

`public ConnectionErrorEvent _onConnectionError`

This event is invoked when an internal error has occurred. Assume that after an internal error the entity is not operative any more. The connection is automatically closed and disposed.
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 ConnectionBaseEvent _onConnectionClose`

This event is invoked when the connection closes unexpectedly.
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 `UnityTCPServer` header to ease the task of defining the events.
Copy to your code, complete with what you need and assign in editor:

```
    // Server:
    public void OnTCPSOpen(UnityTCPServer server)
    { }
    public void OnTCPSNewConnection(TCPConnection connection, UnityTCPServer server)
    { }
    public void OnTCPSError(int code, string message, UnityTCPServer server)
    { }
    public void OnTCPSClose(UnityTCPServer server)
    { }

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

UnityTCPServer methods

The methods are provided to keep control of the entity.

`public void Setup()`

This method prepares the entity to be ready to connect, applying all the provided parameters.
Always call `Setup()` before attempting to `Connect()` the first time.

`public void Connect()`

This method attempts to establish the connection. In `UnityTCPServer` it means that the entity starts listening and receiving incoming connections. Always call `Setup()` before attempting to `Connect()`.
The connection will fail if the provided `_localIP+_localPort` pair is already in use by another entity.

`public void Disconnect()`

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

`public void CloseAllConnections()`

This method forces to close all active connections.

`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.

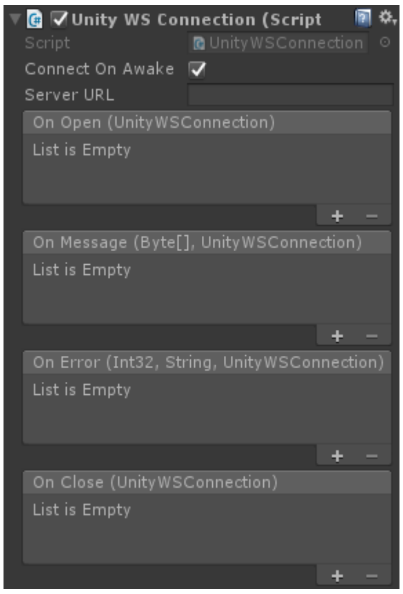
```
public bool IsConnected()
```
Use this method to check if the entity is currently listening.

```
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.

UnityWSConnection

Here is the complete description of the `UnityWSConnection` wrapper and how to use it in Unity.



Unity WS Connection (Script

Script: UnityWSConnection

Connect On Awake: ✓

Server URL:

On Open (UnityWSConnection)
List is Empty

On Message (Byte[], UnityWSConnection)
List is Empty

On Error (Int32, String, UnityWSConnection)
List is Empty

On Close (UnityWSConnection)
List is Empty

The `UnityWSConnection` is designed to be attached to a `GameObject` and all settings can be done in editor too. The wrapper is more than enough for most application purposes.

You can modify the parameters of this entity dynamically, but they will take effect only once `Setup()` is called.

UnityWSConnection parameters

Here is the complete description of all public parameters of this entity.

`public bool _connectOnAwake`

If this flag is set, the `UnityWSConnection` will `Setup()` and `Connect()` automatically at `Awake()`.

`public string _serverURL`

This 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 `UnityWSConnection` will be not able to find the server.
The WebSocket URL can include a "service" ID as this example: `"ws://myserver.com:60000/ws"`
Here the service ID is `"ws"`. This URL must match the prefix set in the server side.

`public float _timeout`

This is the time in seconds to retry the connection when the server is not responding/accepting.

`public float _keepAliveTimeout`

This is the time in seconds to send a "keepAlive" message to the server. This is also the time to probe the server activity.

`public bool _disableWatchdog`

This flag disables the forced onClose of the connection when no activity is detected with the server.

UnityWSConnection events

All events are stacked in the same order they occur, and then they are invoked on each `FixedUpdate()` loop (the whole FIFO stack is invoked progressively). This way it's able to execute Unity code in good synchronization avoiding thread issues.
As usual, always keep the code of the events as short as possible to grant responsiveness and fluidity.

`public BaseEvent _onOpen`

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

`public MessageEvent _onMessage`

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

`public ErrorEvent _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, UnityWSConnection)`
The `int` is the error code.
The `string` is error description.
The `UnityWSConnection` is the reference to the entity which invoked the event.

`public BaseEvent _onClose`

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

Quick reference

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

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

UnityWSConnection methods

The methods are provided to keep control of the entity.

`public void Setup()`

This method prepares the entity to be ready to connect, applying all the provided parameters. Always call `Setup()` before attempting to `Connect()` the first time.

`public void Connect()`

This method attempts to establish the connection. Always call `Setup()` before attempting to `Connect()`. If the remote server is not available, the entity will retry the connection every few seconds indefinitely or until `Disconnect()` is called.

`public void Disconnect()`

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

`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()`

Gets the connected remote URL in its full format.

`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)`

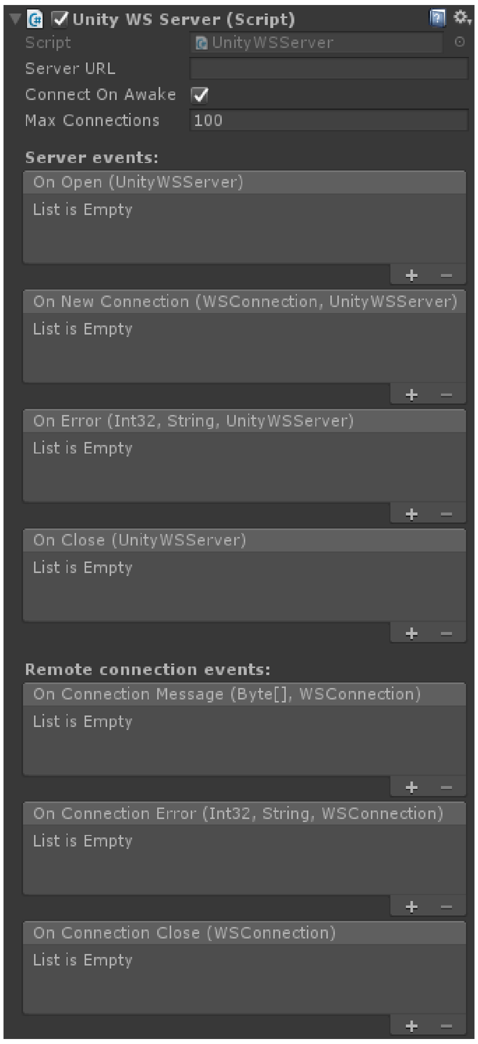
Converts a `string` into a `byte[]` using UTF8 encoding.

`public int SecondsToMiliseconds(float seconds)`

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

UnityWSServer

Here is the complete description of the `UnityWSServer` wrapper and how to use it in Unity.



Unity WS Server (Script)

Script: UnityWSServer

Server URL:

Connect On Awake: ✓

Max Connections: 100

Server events:

On Open (UnityWSServer)
List is Empty

On New Connection (WSConnection, UnityWSServer)
List is Empty

On Error (Int32, String, UnityWSServer)
List is Empty

On Close (UnityWSServer)
List is Empty

Remote connection events:

On Connection Message (Byte[], WSConnection)
List is Empty

On Connection Error (Int32, String, WSConnection)
List is Empty

On Connection Close (WSConnection)
List is Empty

The `UnityWSServer` is designed to be attached to a `GameObject` and all settings can be done in editor too. The wrapper is more than enough for most application purposes.

You can modify the parameters of this entity dynamically, but they will take effect only once **`Setup()`** is called.

UnityWSServer parameters

Here is the complete description of all public parameters of this entity.

`public string _serverURL`

This is the public address to accept new incoming connections (it's a prefix, only one is allowed in this case), 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, there is no way to get it automatically. The provided port must also be open in the local firewall in order to receive incoming connections.
The public URL can include a server ID as in this example: `"ws://myserver.com:60000/ws"`
Here the service ID is `"ws"`. The incoming connections must match the URL including the service ID, or will be not able to connect.

`public bool _connectOnAwake`

If this flag is set, the `UnityWSServer` will `Setup()` and `Connect()` automatically at `Awake()`.

`public int _maxConnections`

This parameter sets the maximum allowed simultaneous incoming connections. All additional connections will be automatically rejected.

`public float _keepAliveTimeout`

This is the time in seconds to probe connection activity. Use two times the timeout set on clients.

UnityWSServer events

All events are stacked in the same order they occur, and then they are invoked on each `FixedUpdate()` loop (the whole FIFO stack is invoked progressively). This way it's able to execute Unity code in good synchronization avoiding thread issues.
As usual, always keep the code of the events as short as possible to grant responsiveness and fluidity.

`public BaseEvent _onOpen`

This event is invoked when the `UnityWSServer` 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(UnityWSServer)`
The `UnityWSServer` is the reference to the entity which invoked the event.

`public ConnectionEvent _onNewConnection`

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

`public ErrorEvent _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, UnityWSServer)`
The `int` is the error code.
The `string` is error description.
The `UnityWSServer` is the reference to the entity which invoked the event.

`public BaseEvent _onClose`

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

`public ConnectionMessageEvent _onConnectionMessage`

This event is invoked when a connection receives a new message. This event is assigned automatically to all new incoming connections.
The signature is: `void OnConnectionMessage(byte[], WSConnection)`
The `byte[]` is the received message.
The `WSConnection` is the reference to the entity which invoked the event.

`public ConnectionErrorEvent _onConnectionError`

This event is invoked when an internal error has occurred. Assume that after an internal error the entity is not operative any more. The connection is automatically closed and disposed.
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 ConnectionBaseEvent _onConnectionClose`

This event is invoked when the connection closes unexpectedly.
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 `UnityWSServer` header to ease the task of defining the events. Copy to your code, complete with what you need and assign in editor:

```
    public void OnWSSOpen(UnityWSServer server)
    { }
    public void OnWSSNewConnection(WSConnection connection, UnityWSServer server)
    { }
    void OnWSSError(int code, string message, UnityWSServer server)
    { }
    public void OnWSSClose(UnityWSServer server)
    { }

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

UnityWSServer methods

The methods are provided to keep control of the entity.

`public void Setup()`

This method prepares the entity to be ready to connect, applying all the provided parameters. Always call **`Setup()`** before attempting to **`Connect()`** the first time.

`public void Connect()`

This method attempts to establish the connection. In `UnityWSServer` it means that the entity starts listening and receiving incoming connections. Always call **`Setup()`** before attempting to **`Connect()`**. The connection will fail if the provided **`_serverURL`** is already in use by another entity.

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

```
public void CloseAllConnections()
```
This method forces to close all active connections.

```
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 int PrefixesCount()
```
Gets the local URL to which the entity es bind. The wrapper allows one single prefix only. This a limitation due to WebSocketSharp, the `.net` version of `UnityWSServer` allows several at once.

```
public string[] Prefixes()
```
Gets the complete list of current available prefixes. The wrapper allows one single prefix only. This a limitation due to WebSocketSharp, the `.net` version of `UnityWSServer` allows several at once.

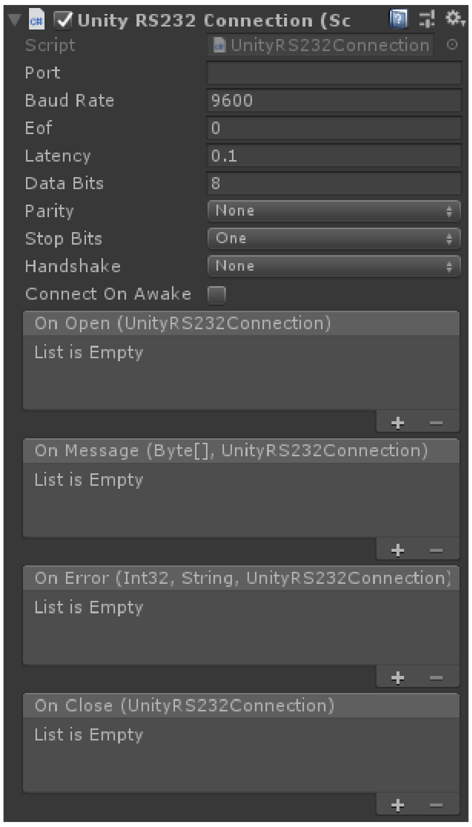
```
public bool IsConnected()
```
Use this method to check if the entity is currently listening.

```
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.

UnityRS232Connection

Here is the complete description of the `UnityRS232Connection` wrapper and how to use it in Unity.


Unity RS232 Connection (Sc

Script	UnityRS232Connection
Port	
Baud Rate	9600
Eof	0
Latency	0.1
Data Bits	8
Parity	None
Stop Bits	One
Handshake	None
Connect On Awake	☐

On Open (UnityRS232Connection)
List is Empty

On Message (Byte[], UnityRS232Connection)
List is Empty

On Error (Int32, String, UnityRS232Connection)
List is Empty

On Close (UnityRS232Connection)
List is Empty

The `UnityRS232Connection` is designed to be attached to a `GameObject` and all settings can be done in editor too.

The RS232 is not a network protocol, so it's not standard at all excepting for its voltage levels and timings. Every RS232 implementation is different, every product/vendor implements a dialog in its own way. Use the core class `RS232Connection` to make your own custom version, it already implements multi-threading and wrapper for Unity. This is better described in the advanced documentation (you'll need strong binary skills).

You can modify the parameters of this entity dynamically, but they will take effect only once **`Setup()`** is called.

UnityRS232Connection parameters

Here is the complete description of all public parameters of this entity.

`public string _port`

This is the identifier name of the serial port. Those names depends on the OS and can't be set randomly, it must be one of those available and can't be repeated (Get them with `GetAvailablePorts()`). This parameter must be provided, there is no way to get it automatically.

`public int _baudRate`

This is the bauds per second value set to the serial port, any value can be set but it should match the same baud rate of the target device, otherwise they'll be not able to communicate (corrupted data is usually a sign of inaccurate baud rate).

`public byte _eof`

This byte (or character) is used to determine the end of a message to be able to fire the `_onMessage` event. This is the default behavior of this class (a very simple one), not standard at all, but very common. If this behavior is not what you need, you should customize the core class `RS232Connection` with your own implementation (the end of a message may be a byte count, a single code, a sequence, a custom CRC, etc.).

`public float _latency`

This parameter sets the latency, the default value of 0,1 is enough for most cases.
The latency is a timeout to discard incomplete incoming data, if incoming byte is interrupted (target device somehow disconnected) after the latency time it's discarded. A smaller value may result in discarding valid bytes if they are send with too much space in between.

`public int _dataBits`

This parameter sets the size of the received byte. Nowadays all systems uses 8 bits data.

`public Parity _parity`

The parity is an extra bit calculated depending on the byte data itself. This bit is never received by the application, it's relevant for the UART only.

`public StopBits _stopBits`

The stop bit is an extra bit which determines the end of the byte data. One bit is the most common setting.

`public Handshake _handshake`

This parameter sets the method to control the data flow. Older (slower) devices needed extra time to process the incoming data before keep receiving, so the handshake signals the data flow to stop or resume. Nowadays it isn't needed anymore (in most cases).

`public bool _connectOnAwake`

If this flag is set, the `UnityRS232Connection` will `Setup()` and `Connect()` automatically at `Awake()`.

UnityRS232Connection events

All events are stacked in the same order they occur, and then they are invoked on each `FixedUpdate()` loop (the whole FIFO stack is invoked progressively). This way it's able to execute Unity code in good synchronization avoiding thread issues.
As usual, always keep the code of the events as short as possible to grant responsiveness and fluidity.

`public BaseEvent _onOpen`

This event is invoked when the `UnityRS232Connection` succeeds establishing the connection.
The signature is: `void OnOpen(UnityRS232Connection)`
The `UnityRS232Connection` is the reference to the entity which invoked the event.

`public MessageEvent _onMessage`

This event is invoked when a message has been received.
The signature is: `void OnMessage(byte[], UnityRS232Connection)`
The `byte[]` is the received message.
The `UnityRS232Connection` is the reference to the entity which invoked the event.

`public ErrorEvent _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, UnityRS232Connection)`
The `int` is the error code.
The `string` is error description.
The `UnityRS232Connection` is the reference to the entity which invoked the event.

`public BaseEvent _onClose`

This event is invoked when the connection closes unexpectedly.
The signature is: `void OnClose(UnityRS232Connection)`
The `UnityRS232Connection` is the reference to the entity which invoked the event.

Quick reference

This piece of code is included in the `UnityRS232Connection` header to ease the task of defining the events. Copy to your code, complete with what you need and assign in editor:

```
    public void OnCOMOpen(UnityRS232Connection connection)
    { }
    public void OnCOMMessage(byte[] message, UnityRS232Connection connection)
    { }
    public void OnCOMError(int code, string message, UnityRS232Connection connection)
    { }
    public void OnCOMClose(UnityRS232Connection connection)
    { }
```

UnityRS232Connection methods

The methods are provided to keep control of the entity.

`public void Setup()`

This method prepares the entity to be ready to connect, applying all the provided parameters. Always call `Setup()` before attempting to `Connect()` the first time.

`public void Connect()`

This method attempts to set the serial port, if available. Always call `Setup()` before attempting to `Connect()`.
The connection will fail if the provided `_port` is not available.

`public void Disconnect()`

In `UnityRS232Connection` it stops the listening thread but not releasing the port. The entity remains ready to `Connect()` again.

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

```
public static string[] GetAvailablePorts()
```
Returns the complete list of all available serial ports in the device. Use this list to verify the name of the port before attempting to `Setup()` or `Connect()`.

```
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.

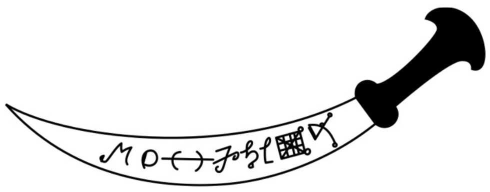
```
public int SecondsToMiliseconds(float seconds)
```
Converts a `float` in seconds into an `int` in milliseconds. Useful when working with timers (`System.Threading.Timer`).

Core entities

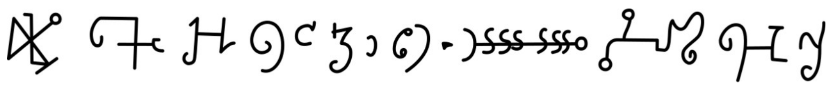
The complete description of core entities `UDPConnection`, `TCPConnection`, `TCPServer`, `WSConnection`, `WSServer`, `NTP_RealTime` and `RS232Connection` is in the advanced documentation.

Known Issues

- WebSocket entities requires API compatibility level `".net 4.x"`. (Set in Project Settings>Player>Other Settings).
- It's forbidden to summon several `UDPConnection` and any server using the exact same IP and Port. One of them must be different.
- In `UDPConnection` the size of the TX/RX buffer will always be reduced when binding to a fixed local IP, thus setting IP automatically is preferable.
- Servers usually needs its owned Ports to be open in the immediate firewall.
- 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.
- The Barzai scimitar is useless for summoning entities. But if you insist here is how it should be forged:



Engrave this characters on one side:



And this characters on the other side:

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, and your name will be engraved on the list of the brave ancient ones.

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

Thanks.