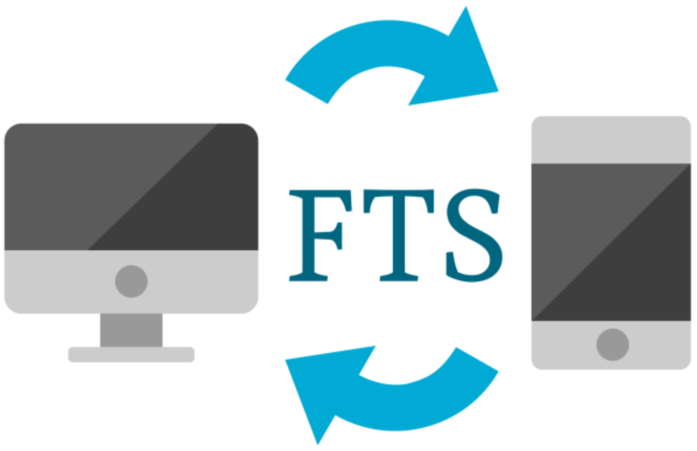


# File Transfer Server

Cross-platform file transfer solution over UDP.

eToile

*(eToile 2025) V: 2.8*

## Index

## Introduction

Thanks for purchasing `FileTransferServer`.

This product is a group of useful scripts that enables your application to send and receive files through a local network (UDP) without a dedicated server. `FileTransferServer` doesn't use **WWW** class neither **FTP**, it implements its own transfer protocol.

FTS is ready to operate under IPV4 and IPV6 networks simultaneously.

Saving and reading is as fast as the device allows. The maximum file size to be transferred is determined by the available disk space in target device, and available memory in both devices: origin and target.

You can access the full source code. The example scene allows testing most of the `FileTransferServer` features.

This documentation describes the class wrapper interfaces, parameters and events. For code examples please refer to the example project, source code it's fully commented.

## Description

`FileTransferServer` is a class that inherits from `MonoBehaviour`, that means that you need to attach the "`FileTransferServer.cs`" script to some `GameObject` in scene, so it can run properly and automatically. This was done in this way to allow multiple instances avoiding complex coding.

`FileTransferServer` uses one instance of the `FTSCore` class, which is the File Transfer Server itself. The `FTSCore` class uses `FileManagement` for reading and saving files on every supported platform, and uses `UDPConnection` to send and receive UDP messages (`UDPConnection` is part of Socket Under Control).

There are no special considerations when exporting for different platforms, neither special considerations when uploading to digital markets. Just switch platform from Build settings dialog on Unity editor and build.

Supported platforms are:

- Windows standalone.
- OSX standalone.
- Linux standalone.
- IOS.
- Android.

The `FileTransferServer` class is a wrapper for `FTSCore`, and its events run into the Unity render thread, so the assigned events (your code) must be kept as small as possible to keep responsiveness. Those events must be assigned in editor, it's done this way for ease of use.

`FTSCore` executes completely in parallel threads, so it allows the main thread to remain responsive and allows to use the maximum system available speed while serving, downloading or saving.

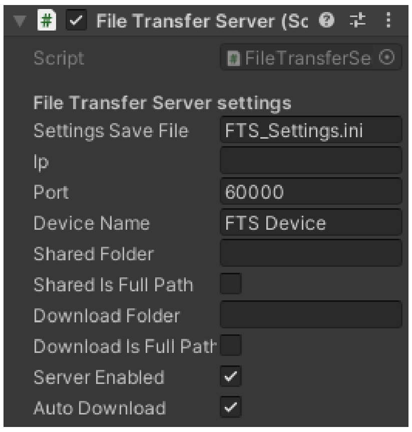
To use this class directly, pleas refer to the advanced documentation.

## Asset Integration

To integrate `FileTransferServer` to your project (once downloaded from the Asset Store) you must attach the "`FileTransferServer.cs`" file to your preferred `GameObject` in your scene.
The `FileTransferServer` class will start properly by itself depending on the options configured in the Unity's "Inspector" window.

## File Transfer Server settings

The `FileTransferServer` component exposes most of its parameters in editor, so you can set the way it should work quick and easy. The parameters in the editor will not take effect if modified in run-time.


File Transfer Server (Sc
Script FileTransferSe

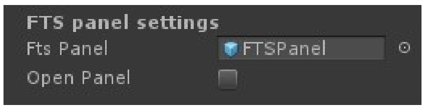
File Transfer Server settings

| | |
|---|---|
| Settings Save File | FTS_Settings.ini |
| Ip | |
| Port | 60000 |
| Device Name | FTS Device |
| Shared Folder | |
| Shared Is Full Path | ☐ |
| Download Folder | |
| Download Is Full Path | ☐ |
| Server Enabled | ✓ |
| Auto Download | ✓ |

Settings Save File: File name to save the FTS settings through the FTSPanel.
Ip: Sets the local IP address to bind to (selected automatically if left empty).
Port: The UDP port to send and receive.
Device Name: Description label for this device (Use this to identify devices).
Shared Folder: The folder that contains the files enabled to be shared (will be created automatically).
Shared Is Full Path: Allows the Shared Folder to be set outside the PersistentDataPath.
Download Folder: The folder where all received files will be placed (will be created automatically).
Download Is Full Path: Allows the Download Folder to be set outside the PersistentDataPath.
Server Enabled: Enables the ability to send files.
Auto Download: Accept forced downloads automatically.

## FTS panel settings

The `FileTransferServer` sets also the parameters of the integrated control panel.


FTS panel settings

| | |
|---|---|
| Fts Panel | FTSPanel |
| Open Panel | ☐ |

Fts Panel: The control panel prefab to be instantiated.
Open Panel: Opens the control panel at start.

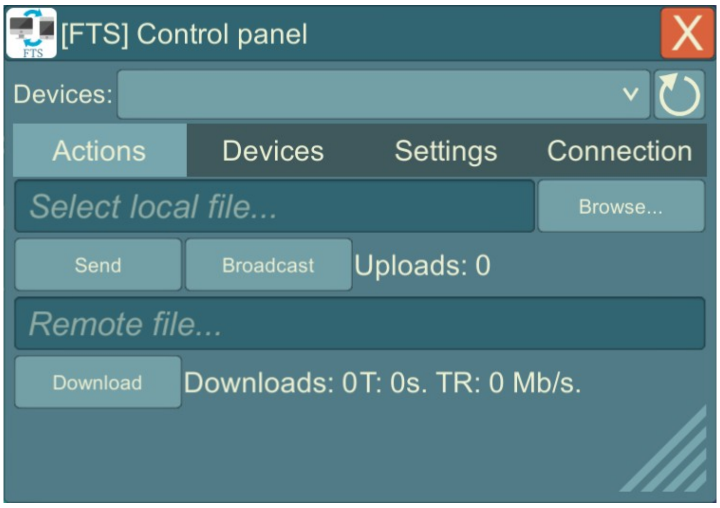
## Control panel

The fastest and easiest way to use FTS in your projects is through the integrated control panel. This control panel includes the most common actions you can do with FTS. It's optimized to allow your application to run smoothly, and you can use it without coding at all.

The control panel can be instantiated easily using the `OpenControlPanel()` method or activating the "`showPanelAtStart`" flag in the FileTransferServer.cs script.

The window can be dragged and resized. There is a fixed section with a drop down labeled as "Devices:" where all discovered devices in the network will be listed to be selected.
The "Refresh" button (placed to the right) sends a polling request to discover all available devices.

The four buttons labeled as "Actions", "Devices", "Settings" and "Connection" contains different controls to dynamically set several parameters in the FileTransferServer which it belongs.



Uploads: 0

Downloads: 0T: 0s. TR: 0 Mb/s.

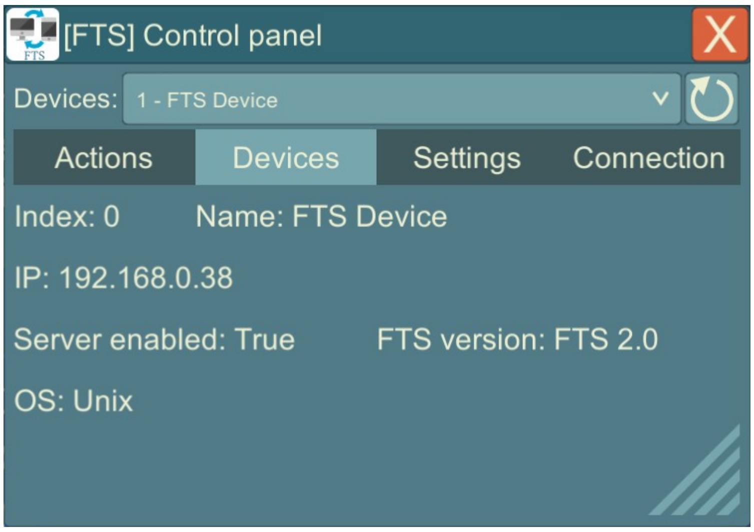
The "Actions" panel allows to send/receive files to/from remote devices.
The "Browse..." button opens the FileManagement's file browser, allowing to select a local shared file. Once the local file is selected, can be send with the "Send" button to the selected device in the "Devices:" drop down. The "Broadcast" button will send the file to all available IPV4 devices in the local network.

There is a dynamic label "Uploads:" that shows the count of simultaneous files being served, ant the percentage of transference.

The "Remote file..." input field, allows to write the name of a remote file, and the "Download" button will attempt to download it from the selected device in the "Devices:" drop down.

The dynamic label "Downloads:" shows the count of simultaneous downloads, the progress percentage, the time and the transfer rate.

There is also a list of events, registering the uploads, downloads and errors.


[FTS] Control panel

Devices: 1 - FTS Device

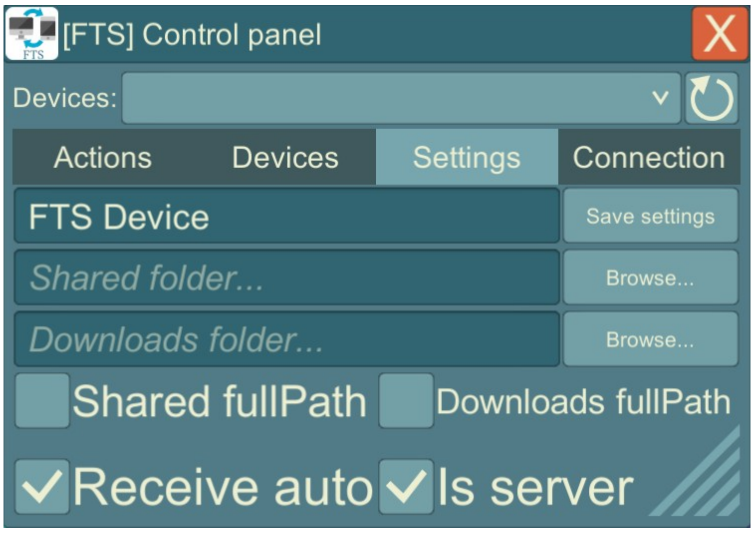
Index: 0 Name: FTS Device

IP: 192.168.0.38

Server enabled: True FTS version: FTS 2.0

OS: Unix

The "Devices" panel shows relevant information relative to the selected device in the dropdown.


[FTS] Control panel

Devices:

FTS Device

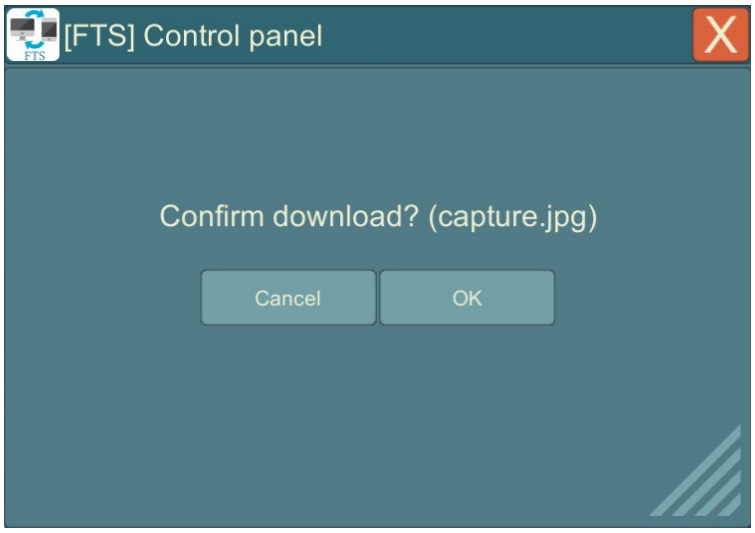
Shared folder...

Downloads folder...

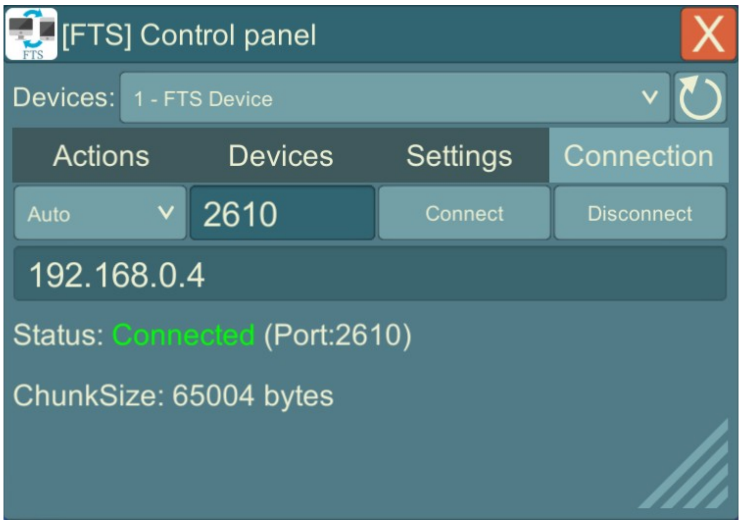
☐ Shared fullPath ☐ Downloads fullPath

✓ Receive auto ✓ Is server

The "Settings" panel allows to set several parameters dynamically.
The "Save settings" button saves the current settings to "FTS_Settings.ini" in the PersistentDataPath. This settings are automatically loaded when the application starts (delete the file to restore default).
The "Device name..." input field sets the name label for the current device.
The "Shared folder..." input field allows to set the current shared folder. The "Browse..." button allows to use the FileManagement's file browser to select this folder visually. The "Shared fullPath" check box indicates that the provided "Shared folder..." is not relative to the PersistentDataPath.
The "Downloads folder..." input field allows to set the current downloads folder, it has its own "Browse..." button too. The "Downloads fullPath" check box indicates that the provided "Downloads folder..." is not relative to the PersistentDataPath.
The "Receive auto" toggle allows to enable/disable the automatic reception of files.
The "Is server" toggle allows to enable/disable the server capabilities dynamically.



When a forced download is received, and the "Receive auto" toggle is disabled, this window will allow the user to accept or reject the incoming download. Otherwise the incoming file will be downloaded immediately (NODE: This message is shown only if the panel is open, it will not open automatically).



2610

192.168.0.4

Status: Connected (Port:2610)

ChunkSize: 65004 bytes

The "Connection" panel allows to connect/disconnect the File Transfer Server.
The drop down allows 4 options:

1. Auto: Connects the FTS using both IPV4 and IPV6 automatically.
2. Auto (IPv4): Connects using IPV4 only (selects automatically).
3. Auto (IPv6): Connects using IPV6 only (selects automatically).
4. Custom: Allows to provide a custom IP or URL in the "Custom local IP…" input field.

The dynamic label "Status" shows if the FTS is connected or disconnected.
The "ChunkSize:" label shows the available chunk size in bytes for this device.

```
This control panel is a custom prefab that can be modified or customized to match your
application needs or visuals.
```

## FileTransferServer Events

This is the complete definition of `FileTransferServer` status events.

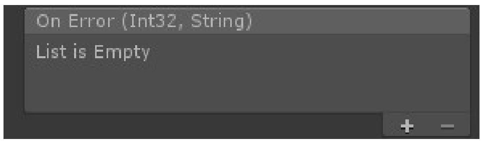
On Error (Int32, String)

```
public volatile ExceptionEvent onError;
```

This event is invoked when any error has occurred. This event must be assigned in editor and the signature to be able to assign a method is:

```
public void UnityErrorEvent(int code, string description);
```

The arguments are the error code and its description.



Your custom event method (to be assigned in editor) must look like this:

```
public void MyErrorEvent(int code, string description) {
    // Your code here.
};
```

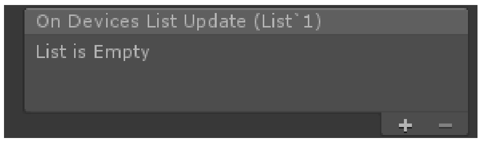
On Devices List Update (List`1)

```
public volatile FTSDeviceEvent onDevicesListUpdate;
```

This event is fired when changes to the automatically detected devices list has been done. That means devices were added or removed.
This event must be assigned in editor and the signature to be able to assign a method is:

```
public void UnityDeviceEvent(List<FTSCore.RemoteDevice>);
```

The argument is the updated list of detected devices.



Your custom event method (to be assigned in editor) must look like this:

```
public void MyDeviceEvent(List<FTSCore.RemoteDevice> devices) {
    // Your code here.
};
```

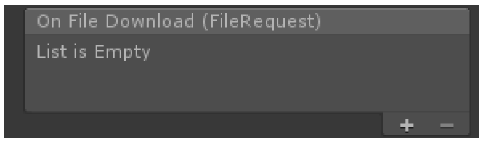
On File Download (FileRequest)

```
public volatile FTSFileEvent onFileDownload;
```

This event is invoked when a complete file was successfully downloaded and saved to disk.
This event must be assigned in editor and the signature to be able to assign a method is:

```
public void UnityDownloadEvent(FTSCore.FileRequest);
```

The argument is the object that represents the requested/downloaded file.



Your custom event method (to be assigned in editor) must look like this:

```
public void MyDownloadEvent(FTSCore.FileRequest request) {
    // Your code here.
};
```

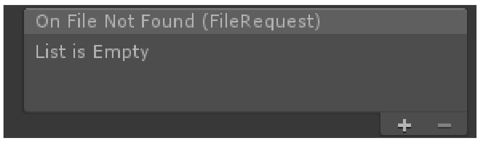
On File Not Found (FileRequest)

```
public volatile FTSFileEvent onFileNotFound;
```

This event is invoked when the requested server responds with a "File not found" message. The remote device responds also with "File not found" when its server capabilities are disabled.

```
public void UnityNotFoundEvent(FTSCore.FileRequest);
```

The argument is the object that represents the requested/downloaded file.



On File Not Found (FileRequest)

List is Empty

Your custom event method (to be assigned in editor) must look like this:

```
public void MyFileNotFoundEvent(FTSCore.FileRequest request) {
    // Your code here.
};
```

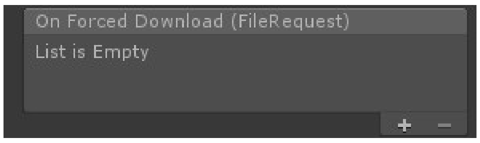
### On Forced Download (FileRequest)

```
public volatile FTSFileEvent onForcedDownload;
```

This event is invoked when a download requested by a remote device has started.

```
public void UnityForcedEvent(FTSCore.FileRequest);
```

The argument is the object that represents the requested/downloaded file.



On Forced Download (FileRequest)

List is Empty

Your custom event method (to be assigned in editor) must look like this:

```
public void MyForcedDownloadEvent(FTSCore.FileRequest request) {
    // Your code here.
};
```

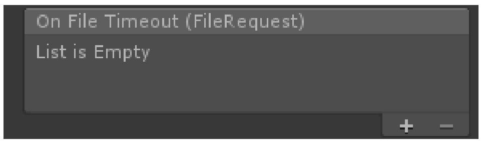
### On File Timeout (FileRequest)

```
public volatile FTSFileEvent onFileTimeout;
```

This event is invoked when the remote device is not responding after 10 retries. The timeout event can be fired during a file request or once the download has already started.

```
public void UnityTimeoutEvent(FTSCore.FileRequest);
```

The argument is the object that represents the requested/downloaded file.



On File Timeout (FileRequest)

List is Empty

Your custom event method (to be assigned in editor) must look like this:

```
public void MyTimeoutEvent(FTSCore.FileRequest request) {
    // Your code here.
};
```

### On Upload Begin (FileUpload)

```
public volatile FTSUploadEvent onUploadBegin;
```

This event is invoked when some device requested a file and the transference has started.

```
public void UnityBeginEvent(FTSCore.FileUpload);
```

The argument is the object that represents the uploaded file.

Your custom event method (to be assigned in editor) must look like this:

```
public void MyBeginEvent(FTSCore.FileUpload upload) {
    // Your code here.
};
```

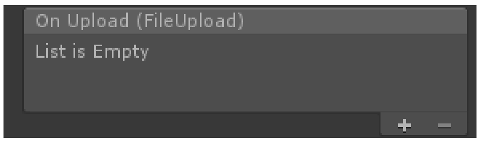
### On Upload (FileUpload)

```
public volatile FTSUploadEvent onUpload;
```

This event is invoked when a file has been transferred successfully.

```
public void UnityUploadEvent(FTSCore.FileUpload);
```

The argument is the object that represents the uploaded file.



Your custom event method (to be assigned in editor) must look like this:

```
public void MyUploadEvent(FTSCore.FileUpload upload) {
    // Your code here.
};
```

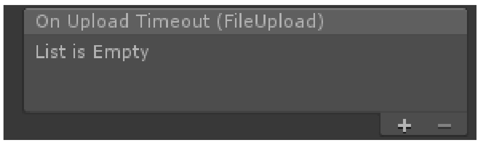
### On Upload Timeout (FileUpload)

```
public volatile FTSUploadEvent onUploadTimeout;
```

This event is invoked when a file has been transferred successfully.

```
public void UnityTimeoutEvent(FTSCore.FileUpload);
```

The argument is the object that represents the uploaded file.



Your custom event method (to be assigned in editor) must look like this:

```
public void MyTimeoutEvent(FTSCore.FileUpload upload) {
    // Your code here.
};
```

## Accessing FileTransferServer (Scripting)

To connect with the `FileTransferServer` class from any other script you should know the name of the `GameObject` where it is attached to, and access the `FileTransferServer` component like this:

```
This example connects with a GameObject called “FTS”:
FileTransferServer fts;
void Start () {
       fts = GameObject.Find("FTS").GetComponent<FileTransferServer>();
       // Now the "fts" variable is connected with the FileTransferServer instance in "FTS".
}
```

Or if the “FileTransferServer.cs” file is attached to itself, use:

```
This example connects with itself:
```

```
FileTransferServer fts;
void Start () {
	fts = gameObject.GetComponent<FileTransferServer>();
	// Now the "fts" variable is connected with the local FileTransferServer instance.
}
```

To access the `FileTransferServer` interfaces you just have to invoke the proper functions in your local connection of the instance:

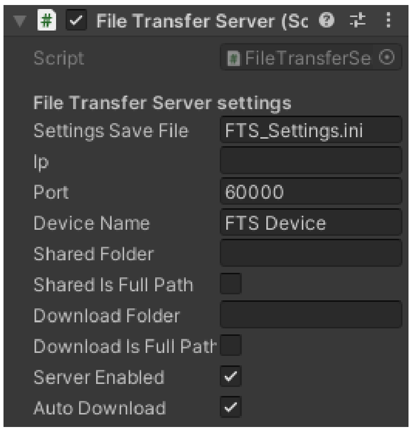
```
This example calls for a single file:
void RequestFile () {
	// The server IP can also be requested automatically with a single call.
	fts.RequestFile("192.168.0.10", "capture.jpg");
}
```

You can add as many `FileTransferServer` instances as you need to your project. Resources are liberated automatically when closing the application or destroying the `GameObject` or switching the scene.

Sockets admits the use of ports only once, so if you have multiple instances of `FileTransferServer` in your project use always different port numbers.

## FileTransferServer parameters

The parameters in the editor will not take effect if modified in run-time.



File Transfer Server (Sc

Script: FileTransferSe

File Transfer Server settings

| | |
|---|---|
| Settings Save File | FTS_Settings.ini |
| Ip | |
| Port | 60000 |
| Device Name | FTS Device |
| Shared Folder | |
| Shared Is Full Path | ☐ |
| Download Folder | |
| Download Is Full Path | ☐ |
| Server Enabled | ✓ |
| Auto Download | ✓ |

### Settings Save File

```
public string _settingsSaveFile = "FTS_Settings.ini";
```

Sets the name of the file to save the settings from the FTS control panel.

### IP

```
public string _ip = "";
```

There are four options to set the IP for the UDP connection:

1. `""` (Empty): Connection will get both available IPV4 and IPV6 addresses automatically.
2. `"ipv4"`: Connection will get the available IPV4 address (only) automatically.
3. `"ipv6"`: Connection will get the available IPV6 address (only) automatically.
4. Custom IP or URL: Connection will bind to the requested address (IPV4, IPV6 or URL).

### Port

```
public string _port = "60000";
```

Sets the UDP port to send and receive. The port can be modified in editor. You can define many port numbers in different `FileTransferServer` instances to listen many ports at same time.
Do not repeat port numbers on different `FileTransferServer` instances in the same device.

Device Name

```
public string _deviceName;
```

Set the name label for the device. This is a human friendly name to recognize devices easily.

Shared Folder

```
public string _sharedFolder = "FTSShared";
```

This parameter defines the folder that contains the files that are allowed to be shared with other devices. It'll be created at start if not exists.

Shared Full Path

```
public bool _sharedFullPath = false;
```

This parameter defines if the shared folder is relative to `PersistentData`. This way you can share a random folder, even in removable drives.

Download Folder

```
public string _downloadFolder = "FTSDownloads";
```

This parameter defines the folder where all received files will be saved. It'll be created at start if not exists.

Download Full Path

```
public bool _downloadFullPath = false;
```

This parameter defines if the download folder is relative to `PersistentData`. This way you can define a random folder, even in removable drives.

Chunk Size

```
public int _chunkSize;
```

This parameter is read only and is calculated automatically using the available UDP buffer at start.
Files are divided in parts called chunks to be transferred. Those chunks must fit the available buffer size on server and client.
By default FTS will use the maximum available size, never assign a chunk size bigger than the calculated automatically or FTS will not be able to transfer. When the available chunk size is different between devices, the smaller one will be used to transfer the file.
When transferring files through internet the available buffer size is not the main limitation but the smallest MTU (Maximum Transmission Unity).
Smallest MTU is usually 1400, use this `_chunkSize` if you are being able to detect devices but files are always timed-out. The MTU depends on the internet infrastructure in your region.

Server Enabled

```
public bool _serverEnabled = true;
```

This parameter controls the capability of serving files to clients. Can be enabled or disabled in run-time. This parameter doesn't disables communications, so it still being able to receive files from other devices.

Enable automatic downloads

```
public bool _autoDownload = true;
```

This parameter controls the option of downloading automatically any forced download received.
If this option is disabled, the forced downloads will need user approval (in the Control Panel) or needs to be started programmatically.

## FileTransferServer Methods

The `FileTransferServer` methods are reduced to the most common tasks and diagnostics.

### Connect()

```
public void Connect()
```

Establishes the UDP connection and applies the last provided values. This connection is done automatically in `Awake()`.

### Disconnect()

```
public void Disconnect()
```

Closes the server connection, but doesn't disposes the object.
Use this method to close the server, change some settings if needed, and reconnect calling the `Connect()` method.

### GetIP()

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

Returns the current IP addresses. The `secondary` argument allows to get both addresses in the case where IPV4 and IPV6 are used simultaneously.

### IsConnected()

```
public bool IsConnected()
```

Returns the connection status.

### SendPollRequest()

```
public void SendPollRequest(string ip = "")
```

Sends a status request message to the provided address. If the `ip` parameter is left blank, then the request will be broadcasted if IPV4 network is available. The `ip` parameter admits also URLs. The use of this method is recommended to make sure the remote device is active and enabled before attempting any transference (IPV6 networks doesn't allows automatic detection).
Every response to this request will be added to the internal devices list, and the `onDevicesListUpdate` event will be fired.

```
This example will detect every available FTS device in the local network:
fts.SendPollRequest();
```

### ResetDeviceList()

```
public void ResetDeviceList()
```

Deletes the internal list of detected devices. The use of this method fires the `onDevicesListUpdate` event.

```
This example deletes the devices list:
fts.ResetDeviceList();
```

### GetDevicesList()

```
public List<FTSCore.RemoteDevice> GetDevicesList()
```

Returns the list of detected devices represented by the `RemoteDevice` object. Use this method to retrieve the list of remote valid FTS devices. This list is sorted by order of appearance.

```
This example gets the detected devices list:
List<FTSCore.RemoteDevice> list = fts.GetDevicesList();
```

### GetDevice()

```
public FTSCore.RemoteDevice GetDevice(int index)
```

Returns the particular device found in the devices list with the provided index (if available).

```
This example gets a particular device from the list:
FTSCore.RemoteDevice device = fts.GetDevice(1);
```

### GetDeviceNamesList()

```
public List<string> GetDeviceNamesList()
```

Returns the list of friendly names of all detected devices. This list is sorted by order of appearance, and corresponds with `GetDevicesList()`.

```
This example gets the detected device's names:
List<string> list = fts.GetDeviceNamesList();
```

## GetDeviceIPList()

```
public List<string> GetDeviceIPList()
```

Returns the list of IP addresses of all detected devices. This list is sorted by order of appearance, and corresponds with the `GetDevicesList()` list.

```
This example gets the detected device's IP addresses:
List<string> list = fts.GetDeviceIPList();
```

## RequestFile()

```
public FTSCore.FileRequest RequestFile(string deviceIP, string file, string saveName = "")
public FTSCore.FileRequest RequestFile(int deviceIndex, string file, string saveName = "")
```

Requests a file. The file is added to the requests list to be downloaded from the selected server.
There are two versions of this interface, the first one admits the IP address directly, the other one selects a server from the internal valid server list (starting with zero).
The `saveName` argument sets the destination folder and name of the downloaded file, always relative to the `_downloadFolder` folder. This method allows to set a new name for the downloaded file.
The method returns a `FileRequest` object representing the download, to track status and information.

```
This example requests a file to the second device in the list:
FTSCore.FileRequest request = fts.RequestFile(1, "capture.jpg");
```

## SendFile()

```
public FTSCore.FileUpload SendFile(string deviceIP, string name)
public FTSCore.FileUpload SendFile(int deviceIndex, string name)
```

Starts a transference from server side. The remote device will start the download using the same method used for the request, but firing the `onForcedDownload` event. This method returns a `FileUpload` object representing the uploaded file, and can be used to track status and get relevant information.

```
This example sends a file to the second device in the list:
FTSCore.FileUpload upload = fts.SendFile(1, "capture.jpg");
```

## BroadcastFile()

```
public void BroadcastFile(string name)
```

This method sends a forced download to all available devices in the network (IPv4 only).

```
This example sends a file in broadcast:
fts.BroadcastFile ("capture.jpg");
```

## LoadResource()

```
public void LoadResource(string name, float timeout = System.Threading.Timeout.Infinite,
bool fullPath = false)
public void LoadResource(string name, byte[] content, float timeout =
System.Threading.Timeout.Infinite)
```

This method forces to load a resource in memory, either from disk or from a custom `byte[]`.
The `timeout` parameter allows the resource to never be released by default, so it's intended to remain in memory during the whole life of the application.

```
This example loads a resource in memory for further use (never being disposed):
fts.LoadResource("capture.jpg");
```

## OpenControlPanel()

```
public void OpenControlPanel()
```

Opens the FTS control panel instantiating the `_openPanel` prefab.

```
This example opens the control panel:
fts.OpenControlPanel();
```

### LoadSettings()

```
public void LoadSettings(string fileName)
```

Loads the settings from the provided `fileName`. This method fills the parameters but doesn't resets the connection, this must be done calling `Disconnect()` and `Connect()`programmatically.

```
This example resets the connection loading new settings from file:
fts.Disconnect();
fts.LoadSettings("FTS_Settings.ini");
fts.Connect();
```

### SaveSettings()

```
public void SaveSettings(string fileName)
```

Saves the current settings to the provided `fileName`.

```
This example saves the current settings:
fts.SaveSettings ("FTS_MySettings.ini");
```

### GetRelativeToSharedFolder()

```
public string GetRelativeToSharedFolder(string absolutePath)
```

Providing an `absolutePath` will calculate the relative path into the shared folder. Will return an empty string if the provided path is not contained into the shared folder.

### IsIntoSharedFolder()

```
public bool IsIntoSharedFolder(string absolutePath))
```

Checks if the provided `absolutePath` is contained into the shared folder.

## Internal classes

There three main internal classes to allow the track any transference and remote device, those classes are:

- `RemoteDevice`: Represents a remote device and saves relevant information.
- `FileRequest`: Represents a file being downloaded and saves status and relevant information.
- `FileUpload`: Represents a file being uploaded and saves status and relevant information.
- `FileResource`: Represents a file cached in memory (this object is used by `FileUpload` and `FileRequest`).

## Known Issues

- Don't run two instances of FTS with the same IP and the same PORT in the same device. At least one of both parameters must be different (that's a UDP limitation).
- You can connect through internet (from/to public server) but the transfer rates can be really slow depending on the MTU in your region (Smallest is 1.4K normally vs ~64K in local networks). You have to set the `_chunkSize` accordingly.
- Android file system is case sensitive (others are not). Files may not be found if names don't match exactly.
- This applies to Unity Editor only (exported applications are fine): In order to be able to allow communications in UDP using IPV6, you must start by sending a message to a valid remote IPV6 address, then will work normally. You can achieve this sending the PollRequest message.
- Mobile platforms can't handle files larger than 2GB.
- Files larger than 2GB in StreamingAssets aren't supported on any platform.

## Contact

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

Once you have used this product, 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/slug/158337

Thanks.