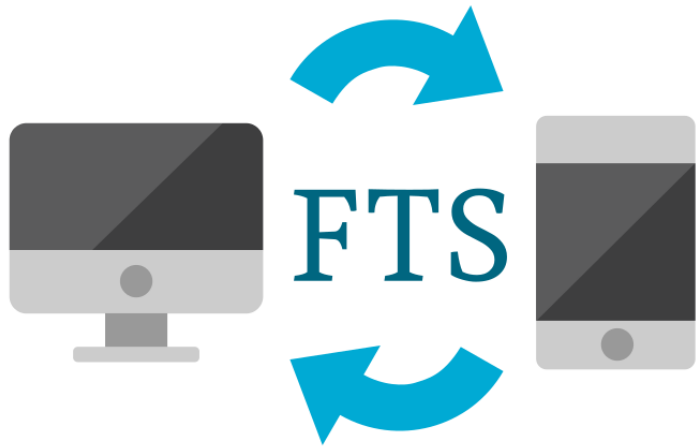
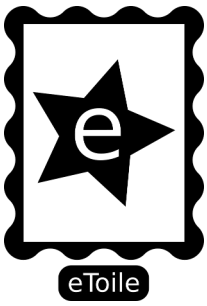


File Transfer Server

Advanced documentation



(eToile 2023) V: 2.6

Index

Introduction

Thanks for purchasing `FileTransferServer`.

This is the advanced documentation of File Transfer Server, that shows how to use the main class `FTSCore` alone. Using this class allows your code to be driven completely by callbacks, profiting the maximum available speed of the device.
You can use this class directly under any C# compatible platform as Unity, Visual Studio, Xamarin and Mono.

Reading and saving is as fast as the device supports. The maximum file size to be transferred is determined by the available disk space in the 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.

`FTSCore` is also ready to operate under IPV4 and IPV6 networks simultaneously.

Description

The `FTSCore` class uses `FileManagement` for reading and saving files on every supported platform, and uses `UDPConnection` to send and receive UDP messages.

The correct version of `FileManagement` is included in the package (`FileManagement` is designed to be used in other platforms as well as in Unity).

`FTSCore` executes completely in parallel threads, so it allows the main thread (UI/game renderer) to remain responsive and allows to use the maximum system available speed. The helper classes can be accessed at any moment from any thread in order to get and show progress, status, etc.

The transfer rate on standalone devices is about 10MB/s and about 3MB/s on mobile devices.

To get a working example of `FTSCore` implementation please take a look at the Windows Forms project found in the "FTS_Core_WinForms.zip" file at the "Assets/eToile/FileTransferServer" folder.

Asset Integration

To integrate this class in your project you must have these files:

- `FTSCore`
- `UDPConnection`
- `FileManagement`

To create the `FTSCore` object using the default parameters (You can use other port than 60000):

```
FTSCore _fts = new FTSCore("60000");
```

In this example only the port is being assigned, every other parameters are explained at the end of this section.

To detect all available devices in a IPV4 network use:

```
_fts.SendPollRequest();
```

Once other devices has responded, you can begin to send and receive files.

NOTE: If you are serving the same file in two different instances of `FTSCore`, the resource will not be memory optimized. It will be loaded twice, one by each `FTSCore` instance.

Sockets allows only one use of ports, so if you have multiple instances of `FTSCore` in your project, make sure you are not repeating the same pair LocalIP + PORT (At least one of them must be different). This is a UDP limitation, but `FTSCore` allows to set the local IP in order to be able to have two instances on two different network adapters, in the same device, running at once.

FTSCore

The constructor allows to set many relevant parameters at the same time that the `FTSCore` object is created and connected:

```
public FTSCore(string port,
               EventError onError = null, EventDevices onDeviceListUpdate = null,
               EventFile onFileDownload = null, EventFile onFileNotFound = null,
               EventFile onForcedDownload = null, EventFile onFileTimeout = null,
               EventUpload onUploadBegin = null, EventUpload onFileUpload = null,
               EventUpload onUploadTimeout = null,
               string localIP = "",
               string defaultSharedFolder = "FTSShared",
               string defaultDownloadFolder = "FTSDownloads",
               bool sharedFullPath = false, bool downloadFullPath = false)
```

`port`: The port assigned to this instance.
`onError`: Event fired when an error occurred.
`onDeviceListUpdate`: Event fired when the list of detected devices was modified.
`onFileDownload`: Event fired when a file was downloaded.
`onFileNotFound`: Event fired when the requested file not exists in server side.
`onForcedDownload`: Event fired when receiving a file that wasn't requested.
`onFileTimeout`: When a download has lost connection.
`onUploadBegin`: When a file started being served.
`onFileUpload`: When a file was served.
`onUploadTimeout`: When an upload has lost connection.
`localIP`: The desired local IP to bind to (Provide IPV4 or IPV6 as is. An empty string (`""`) will set two IP addresses automatically, being IPV4 the main IP and IPV6 the secondary, this is the default and recommended mode. You can provide also the literal `"ipv4"` in order to use an automatic IPV4 address only, or `"ipv6"` in order to use an automatic IPV6 address only).
`defaultSharedFolder`: The desired shared folder (`"FTSShared"` by default, it's created if not exists).
`defaultDownloadFolder`: The desired download folder (`"FTSDownloads"` by default, it's created if not exists).
`sharedFullPath`: Should be `true` it providing an absolute path in `defaultSharedFolder`.
`downloadFullPath`: Should be `true` it providing an absolute path in `defaultDownloadFolder`.

FTSCore parameters

This is the complete definition of `FTSCore` parameters. Most of them can be set dynamically.

Device Name

```
public string _deviceName;
```

Set the name for the device. This is a human friendly name to recognize devices easily on remote FTS devices.

Shared Folder

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

Set the folder intended to contain all resources allowed to be shared.

Shared Folder Full Path

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

Sets the interpretation of the `_sharedFolder` provided path as relative to persistentDataPath (`false`) or as an absolute path (`true`).

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.

Auto Download

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

This parameter allows to receive any "forced download" automatically. The "forced download" will be saved with it's original name and path relative to the "Downloads" folder.

Resources Timeout

```
public int _resourcesTimeout = 30000;
```

This parameter sets the time that resources are maintained in memory before its disposal. The value is in milliseconds, so 30000 is equal to 30 seconds.

Downloads Folder

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

Set the folder where all downloads will be saved once received. The downloaded files will be saved using their source names, but discarding the source path.

Downloads Folder Full Path

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

Sets the interpretation of `_downloadFolder` provided path as relative to persistentDataPath (`false`) or as an absolute path (`true`).

Chunk Size

```
public int _chunkSize;
```

This is calculated automatically using the available UDP buffer at start, but can be modified if needed (not recommended unless it helps to fix some issue).
Files are divided in parts called chunks to be transferred. Those chunks must smaller than the available buffer size on server and buffer. By default FTS will use the maximum available size, never assign a chunk size bigger than the automatically calculated or FTS will be not able to transfer.

Devices list

```
public List<RemoteDevice> _devices;
```

This is the dynamic list of available FTS devices.

Requests list

```
public List<FileRequest> _requests;
```

This is the list of all current file requests. Do not modify this list manually, it's controlled automatically and must be used for information only.

Uploads list

```
public List<FileUpload> _uploads;
```

This is the list of all current file uploads. Do not modify this list manually, it's controlled automatically and must be used for information only.

FileStatus

```
public enum FileStatus
```

This `enum` helps to define and establish the different stages during a download or upload.

```
   0 = inactive,                    // The default status (when just created).
   1 = requested,                   // The first request was sent.
   2 = started,                     // This file has received at least one chunk.
   3 = saving,                      // The file was downloaded and is being saved to disk.
   4 = finished,                    // The file was saved.
   5 = failed                       // The requested file not found or timed out.
```

FTSCore Events

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

On Error

```
public EventError _onError;
```

This event is invoked when any error has occurred (system, code or transference errors). It can be assigned through the class constructor or adding the custom callback method to the public delegate.
The signature of the `EventError` delegate is:

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

The arguments are the error code and its description.
This callback is assigned like this (or in the constructor):

```
_fts._onError += UnityErrorEvent;
```

You can assign several methods to the same delegate.

On Devices List Update

```
public EventDevices _onDeviceListUpdate;
```

This event is fired when the automated devices list was modified, devices were added or removed.
The signature of the `EventDevices` delegate is:

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

The argument is the updated list of detected devices.
This callback is assigned like this (or in the constructor):

```
_fts._onDeviceListUpdate += UnityDeviceEvent;
```

You can assign several methods to the same delegate.

On File Download

```
public EventFile _onFileDownload;
```

This event is invoked when a complete file was successfully downloaded and saved to disk.
The signature of the `EventFile` delegate is:

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

The argument is the object that represents the requested/downloaded file.
This callback is assigned like this (or in the constructor):

```
_fts._onFileDownload += UnityDownloadEvent;
```

You can assign several methods to the same delegate.

On File Not Found

```
public EventFile _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.
The signature of the `EventFile` delegate is:

```
public void UnityNotFoundEvent(FTSCore.FileRequest);
```
The argument is the object that represents the requested/downloaded file.
This callback is assigned like this (or in the constructor):
```
_fts._onFileNotFound += UnityNotFoundEvent;
```
You can assign several methods to the same delegate.

On Forced Download
```
public EventFile _onForcedDownload;
```
This event is invoked when a download requested by a remote device has been received.
The signature of the `EventFile` delegate is:
```
public void UnityForcedEvent(FTSCore.FileRequest);
```
The argument is the object that represents the requested-to-be-downloaded file.
In case that automatic downloads are disabled, this request is inactive and must be started manually or disposed.
This callback is assigned like this (or in the constructor):
```
_fts._onForcedDownload += UnityForcedEvent;
```
You can assign several methods to the same delegate.

On File Timeout
```
public EventFile _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 even once the download has already started.
The signature of the `EventFile` delegate is:
```
public void UnityTimeoutEvent(FTSCore.FileRequest);
```
The argument is the object that represents the requested file.
This callback is assigned like this (or in the constructor):
```
_fts._onFileTimeout += UnityTimeoutEvent;
```
You can assign several methods to the same delegate.

On Upload Begin
```
public EventUpload _onUploadBegin;
```
This event is invoked when some device requested a file and the transference has started.
The signature of the `EventUpload` delegate is:
```
public void UnityBeginEvent(FTSCore.FileUpload);
```
The argument is the object that represents the file being uploaded.
This callback is assigned like this (or in the constructor):
```
_fts._onUploadBegin += UnityBeginEvent;
```
You can assign several methods to the same delegate.

On File Upload
```
public EventUpload _onFileUpload;
```
This event is invoked when a file has been sent successfully.
The signature of the `EventUpload` delegate is:
```
public void UnityUploadEvent(FTSCore.FileUpload);
```
The argument is the object that represents the uploaded file.
This callback is assigned like this (or in the constructor):
```
_fts._onFileUpload += UnityUploadEvent;
```
You can assign several methods to the same delegate.

On Upload Timeout (FileUpload)
```
public EventUpload _onUploadTimeout;
```
This event is invoked when a file has been transferred successfully.
The signature of the `EventUpload` delegate is:
```
public void UnityTimeoutEvent(FTSCore.FileUpload);
```

The argument is the object that represents the uploaded file.
This callback is assigned like this (or in the constructor):

```
_fts._onUploadTimeout += UnityTimeoutEvent;
```

You can assign several methods to the same delegate.

FTSCore Methods

The `FTSCore` methods are divided in 5 groups:

1. Connection: To connect and disconnect the server.
2. Devices: To control the list of detected devices.
3. Cache: Control over the resources (files) to be served.
4. Download: Request file downloads (and get status/progress).
5. Upload: Control on file uploads (and get status/progress).

Connection: Dispose()

```
public void Dispose()
```

Closes the server connection completely. You have to create a new `FTSCore` object after calling this method.

Connection: GetIP()

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

Gets the IP address being used by the UDP connection. This connection may contain two IP addresses at once, being the main one IPV4 and the secondary IPV6.
If one of this connections is not available, it will return the localhost address or an empty string.

Connection: GetPort()

```
public string GetPort()
```

Gets the port used by the UDP connection to send and receive files.

Connection: IsConnected()

```
public bool IsConnected()
```

Gets the connection status. This parameter is affected just by the UDP connection.

Devices: SendPollRequest()

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

Sends a status request message to the provided address. If the `ip` parameter is left blank, and IPV4 network is available, the request will be broadcasted. The `ip` parameter admits also URLs.
Use this request to make sure the remote device is active and enabled, this step is recommended when your architecture is not detected automatically (as happens with IPV6 or with fixed IP devices).
Every response to this request will be added to the internal devices list, and the `onDevicesListUpdate` event will be fired.

```
This example discovers all FTS available devices in the local network:
_fts.SendPollRequest();
```

Devices: AddNewDevice()

```
public void AddNewDevice(string ip, string name, string enabled, string os = "", string
ftsVer = "")
```

Allows to add a known remote device manually.
The `ip` argument is the remote IP (IPV4, IPV6 or URL).
The `name` argument is the human friendly name of the remote device.
The `enabled` argument should have the literals `"TRUE"` or `"FALSE"`. This is the server capabilities of the remote device.
The `os` and `ftsVer` are optional, and can be left blank if not needed.

```
This example gets the detected devices list:
```

```
List<FTSCore.RemoteDevice> list = fts.GetDevicesList();
```

Devices: 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();
```

Cache: LoadResource()

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

Manually loads a resource in memory (from file or from a `byte[]`). The **timeout** argument is in seconds, and will dispose the resource liberating memory once timed out. If the **timeout** argument is not provided, the resource will never be eliminated from the memory. The optional parameter **fullPath** allows to load a file with a path other than the **_sharedFolder**.

```
This example loads a resource manually, and leave it there permanently:
_fts.LoadResource("MyMovie.AVI");
```

Cache: GetResourcesCount()

```
public int GetResourcesCount()
```

Gets the count of resources loaded currently in memory. This is intended for status information.

```
This example gets the count of resources in memory:
int res = _fts.GetResourcesCount();
```

Cache: GetResourcesSize()

```
public long GetResourcesSize()
```

Gets the total size in memory used by the loaded resources. This is intended for status information.

```
This example gets the count of resources in memory:
long res = _fts.GetResourcesSize();
```

Download: RequestFile()

```
public FileRequest RequestFile(int device, string name, string saveName = "", bool
autoDownload = true)
public FileRequest RequestFile(string serverIP, string name, string saveName = "",
bool autoDownload = true)
```

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 a new name for the downloaded file. This argument allows to set also a new path for the downloaded file, always relative to the **_downloadFolder** folder.
The method returns a `FileRequest` object representing the download, to track status and information.
The **autoDownload** parameter allows this particular request to be created but not started when `false`.

```
This example requests a file to the second device in the list, and saves without asking:
FTSCore.FileRequest request = _fts.RequestFile(1, "capture.jpg");
```

Download: RequestCount()

```
public int RequestCount()
```

Returns the amount of file request in progress. This is intended for status information.

```
This example gets the count of file requests in progress:
int requests = _fts.RequestCount();
```

Upload: SendFile()

```
public FileUpload SendFile(string deviceIP, string name, bool fullPath = false)
public FileUpload SendFile(int device, string name, bool fullPath = false)
```

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.
The **fullPath** parameter allows to send a file from outside the **_sharedFolder** (Any `FileUpload` using this feature will fail with "File not found" if broadcasted).

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

Upload: BroadcastFile()

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

Sends in broadcast a "download request" to all devices in the local network. This works under IPV4 networks only. Every remote device will start downloading the requested file independently.
The **name** file path must be always relative to the **_sharedFolder**.

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

Cache: UploadCount()

```
public int UploadCount()
```

Returns the amount of file uploads in progress. This is intended for status information.

```
This example gets the count of uploads in progress:
int uploads = _fts.UploadCount();
```

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 four main internal classes to allow the track of 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 ready to be uploaded, or accumulates the received chunks to be served.

RemoteDevice

The `RemoteDevice` class is part of the `FTSCore` class, and represents a valid remote device detected automatically through the **SendPollRequest()** method.
This class stores relevant information about the detected devices. Detected devices are added to an internal list automatically in two possible cases:

- When a device replies to a poll request.
- When the poll request itself is received.

This means that when a poll request is sent, both devices (origin and target) know the existence of each other.
Sending the poll request in broadcast allows to map the whole local network with compatible devices.
This object is returned by the **onDevicesListUpdate** event and the **GetDevicesList()** method.

ip

```
volatile public string ip = "";
```

This parameter saves the remote device's IP address. This IP can be IPV4 or IPV6.

```
This example gets the IP address of a remote device:
string remoteIP = device.ip;
```

name

```
volatile public string name = "";
```

This parameter saves the remote device's friendly name.

```
This example gets the name of a remote device:
string name = device.name;
```

os

```
volatile public string os = "";
```

This parameter saves the remote device's operative system description and version.

```
This example gets the OS of a remote device:
string os = device.os;
```

ftsVersion

```
volatile public string ftsVersion = "";
```

This parameter saves the remote device's FTS version protocol, in order to detect incompatible devices in the network.

```
This example gets the FTS version of a remote device:
string ftsVersion = device.ftsVersion;
```

isServer

```
volatile public string isServer = "";
```

This parameter saves the remote device's server capabilities status. If this parameter is disables, then files can't be downloaded/requested, the `RequestFile()` method will do nothing and return null.

```
This example tests if the remote device is active as a server or not:
if(device.isServer) print("Is a valid server.");
```

FileRequest

The `FileRequest` class is part of the `FTSCore` class, and represents a file to be downloaded from a remote device properly enabled as a server.
This class stores relevant information about the remote device, source file, download status, progress and statistics. This object is returned by the `onFileDownload`, `onFileNotFound`, `onForcedDownload` and `onFileTimeout` events and the `RequestFile()` methods.

This object is thread safe, and can be accessed at any moment and from any thread. Here will be described only the most relevant parameters and interfaces.

Please note that parameters are for internal use only, so it's not recommended to modify them programmatically or its behavior will be unexpected.

This object is disposed automatically once finished or failed.

To be able to access this object, just save its reference into a variable. Then check the status, progress during the download. Check also download statistics once the transference has finished.

FileRequest parameters

_remoteIP

```
public volatile string _remoteIP = "";
```

This parameter saves the IP of the remote server from where the file is being downloaded.

```
This example gets the remote IP for this download:
string remoteIP = request._remoteIP;
```

_sourceName

```
public volatile string _sourceName = "";
```

This parameter saves the remote file path and name.

```
This example gets the file name for this download:
string name = request._sourceName;
```

_saveName

```
public volatile string _saveName = "";
```

This parameter saves the local name which will be assigned to the file once downloaded. If no parameter was provided, then the file will be saved with exactly the same name.

```
This example gets the "rename" file name for this download:
string newName = request._saveName;
```

_chunkSize

```
public volatile int _chunkSize;
```

This parameter saves the smaller supported chunk size between server and client. This parameter is updated automatically.

```
This example gets the automatically chosen chunk size for this download:
int chunkSize = request._chunkSize;
```

_elapsedTime

```
public volatile float _elapsedTime = -1f;
```

This parameter saves the total elapsed time from the reception of the first to the last chunk. The time is expressed in seconds.

NOTE: Read this value once the download has finished, or may contain garbage.

```
This example gets the total time taken to finish the download:
float time = request._elapsedTime;
```

_transferRate

```
public volatile float _transferRate = 0f;
```

This parameter saves the calculation of the transfer rate achieved to finish the download. This value is expressed on Mega bytes per second.

NOTE: Read this value once the download has finished, or may contain garbage.

```
This example gets the transfer rate:
float transferRate = request._transferRate;
```

_size

```
public volatile int _size = 0;
```

This parameter saves the total size of the file being downloaded in bytes.

```
This example gets the transfer rate:
int fileSize = request._size;
```

_retries

```
public volatile int _retries = 0;
```

This parameter saves the count of chunks not received at first try, and then had to be requested twice. This value gives an idea about the amount of dropped messages in the local network.

```
This example gets the transfer rate:
float transferRate = request._transferRate;
```

FileRequest methods

Start()

```
public void Start()
```

This method resumes the download when it's inactive. Use this method to resume a forced download once the user has granted approval.

This example starts/resumes the download:
```
request.Start();
```

GetStatus()

```
public FileStatus GetStatus()
```

This method returns the internal download status using the `enum FTSCore.FileStatus` enumerator. The status can have the values:

- `inactive`: The request was just created.
- `requested`: The request is being requested to the remote device.
- `started`: The download has started and the first chunk was already received.
- `saving`: The file was already downloaded and is being restored and saved to disk.
- `finished`: The file was saved successfully.
- `failed`: There was a problem (File not found or timeout).

```
This example gets the file status:
FTSCore.FileStatus status = request.GetStatus();
```

IsThis()

```
public bool IsThis(string remoteIP, string name)
public bool IsThis(FileRequest request)
```

This method checks if the provided information or `FileRequest` matches `this` request. This can be asked at any moment.

```
This example checks if the previously saved request matches the argument of the event:
public void ShowDownload(FTSCore.FileRequest request)
{
    if(_fileRequest != null && _fileRequest.IsThis(request))
        print("Match !!");
}
```

Dispose()

```
public void Dispose()
```

This method disposes the request completely. Use this method to reject a forced download.

This example rejects the download:
```
request.Dispose();
```

GetProgress()

```
public float GetProgress()
```

This method returns the current progress of the download. The returned `float` value grows from zero to one (0 to 1). This value can be asked at any moment.

This example fills a bar with the current progress:
```
Image bar;
bar = transform.GetComponent<Image>();
bar.fillAmount = upload.GetProgress();
```

GetProgressInt()

```
public int GetProgressInt()
```

This method returns the current progress of the download. The returned `int` value grows from zero to one hundred (0 to 100). This value can be asked at any moment.

```
This example gets the current progress and converts into percentage (0% to 100%):
string progress = request.GetProgressInt() + "%";
```

FileUpload

The `FileUpload` class is part of the `FTSCore` class, and represents a file to be uploaded to a remote device.

This class stores relevant information about the remote device, source file, download status and progress. This object is returned by the `onUploadBegin`, `onUpload`, and `onUploadTimeout` events and the `SendFile()` methods.

This object is thread safe, and can be requested at any moment and from any thread. Here will be described only the most relevant parameters and interfaces.

Please note that parameters are for internal use only, so it's not recommended to modify them programmatically or its behavior will be unexpected.

This object is disposed automatically once finished or failed.

To be able to access this object, just save its reference into a variable. Then check the status, progress during the download.

FileUpload parameters

_chunkSize

```
volatile public int _chunkSize;
```

This parameter saves the smaller supported chunk size between server and client. This parameter is updated automatically and should not be modified or the transference risks to fail.

```
This example gets the automatically chosen chunk size for this upload:
int chunkSize = request._chunkSize;
```

FileUpload methods

GetStatus()

```
public FileStatus GetStatus()
```

This method returns the internal upload status using the `enum FTSCore.FileStatus` enumerator.
The status can have the values:

- `inactive`: The upload was just created.
- `requested`: The upload is being pulled to the remote device.
- `started`: The download has started and the first chunk was already sent.
- `finished`: The file was transferred successfully.
- `failed`: There was a problem (File not found or timeout).

```
This example gets the file status:
FTSCore.FileStatus status = upload.GetStatus();
```

IsThis()

```
public bool IsThis(string remoteIP, string name)
public bool IsThis(FileUpload upload)
```

This method checks if the provided information or `FileUpload` matches `this` upload. This can be asked at any moment.

```
This example checks if the previously saved upload matches the argument of the event:
public void ShowUpload(FTSCore.FileUpload upload)
{
    if(_fileUpload != null && _fileUpload.IsThis(upload))
        print("Match !!");
}
```

GetProgress()

```
public float GetProgress()
```

This method returns the current progress of the upload. The returned `float` value grows from zero to one (0 to 1). This value can be asked at any moment.

This example fills a bar with the current progress:

```
Image bar;
bar = transform.GetComponent<Image>();
bar.fillAmount = upload.GetProgress();
```

GetProgressInt()

```
public int GetProgressInt()
```

This method returns the current progress of the upload. The returned `int` value grows from zero to one hundred (0 to 100). This value can be asked at any moment.

```
This example gets the current progress:
string progress = upload.GetProgress() + "%";
```

File resources

The resources are files loaded in memory when requested to be served. Resources are loaded automatically when a transference was requested by a remote server, or if a file must be send to a remote device.

This resources stay in memory by 30 seconds by default. This value can be established accessing the `FTSCore` class directly.

Once elapsed this time, the resource will be disposed to free memory. If any transference is taking place, this timer is restored until the transference has finished (so, unnecessary resources are automatically disposed).

It's possible to load a resource into memory and leave it there without being automatically disposed, through the `LoadResource()` method. This method also allows to load a resource through a `byte[]` instead of a real file (2GB max due to arrays cant be larger).

NOTE: Large files may reduce the system memory, so make sure you have space enough before serving or the transference will fail.

FileResource parameters

_name

```
volatile public string _name;
```

This parameter saves the name used to load the file (relative or full path).

```
This example gets the name of the resource:
Print("File name: " + resource._name);
```

_fullPath

```
volatile public bool _fullPath;
```

When its value is `true` then the _name is fullPath, otherwise it's relative to shared folder.

```
This example gets the name of the resource:
Print("Is full path?: " + resource._fullPath);
```

FileResource methods

Reload()

```
public bool Reload()
```

This method reloads the file content in memory from the original file in disk, it also resets the life time timer. This method is useful if you know that the file in disk was modified and you need to update the cached file.
The file is reloaded using the same parameters used when the object was created.

```
This example updates the cached file dynamically:
if(resource.Reload())
    Print("Cached file reloaded successfully");
else
    Print("File reload has failed (file doesn't exist anymore)");
```

IsThis()

```
public bool IsThis(string name)
```

This method checks if the provided information matches `this` resource. This can be asked at any moment.

```
This example checks if the resource name matches:
if(resource.IsThis("capture.jpg"))
    print("Match !!");
```

GetFileSize()

```
public long GetFileSize()
```

This method returns the size of the cached file in bytes.

```
This example prints the size of a resource:
print("Size in bytes: " + resource.GetFileSize());
```

CheckSumA()

```
public uint CheckSumA()
```

This method performs a "8bit-addition" checksum. Useful to compare files.

```
This example prints the checksum of a resource:
print("Addition checksum: " + resource.CheckSumA());
```

CheckSumX()

```
public byte CheckSumX()
```

This method performs a "8bit-XOR" checksum. Useful to compare files.

```
This example prints the checksum of a resource:
print("Addition checksum: " + resource.CheckSumX());
```

How are files transferred

Any file is loaded in memory as a resource before the transference begins, the file access is faster from memory than from disk. Files are loaded only once in memory per class instance, so the memory is optimized as much as possible. If several devices are downloading the same resource, it will be loaded in memory only once and will be served to all those devices.

If you are serving the same file from two different instances of `FTSCore` in the same device, each of them will load the same file in memory for internal use, resulting in an involuntary duplicated file.

There are two ways to transfer a file, requesting it to any remote server or sending the request itself to a remote device.
Once the download starts in the client side (by himself, or requested by a server) it requests every chunk separately, then stores the received chunk and requests the next one. If the requested chunk is not received within a timeout, it's requested again, if this request fails repeatedly the request will be marked as failed and then disposed.

This is done in this way in order to optimize the transference depending on the processing power of the devices, so the transference will be as fast as the slowest device allows. The size of the chunks is established as the smallest available between both devices.

For files under 2GB every received chunk is saved in memory, and once completed, the whole content is saved to disk. Big files takes time to be saved, so the process is completely done in parallel threads. On the other hand, files larger than 2GB are saved to disk chunk by chunk, so if the transference is failed, the resulting file will be corrupted (the corrupted file should be deleted manually, or retried).

Mobile platforms can't handle files larger than 2GB. Standalone platforms can't load files larger than 2GB only if they are stored in StreamingAssets.

When a "Forced download" is received, but the "autoDownload" flag is disabled, it's necessary to start the download manually. This behavior allows the application to request user approval before allowing the file to be downloaded.

Known Issues

- Don't run two instances of FTS in the same device with the same IP and the same PORT. At least one of them must be different (that's a basic 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 may 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 goes 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 it'll work normally. You can achieve this sending the PollRequest message.
- Providing a local IP address to bind to, will (in most cases) reduce the available chunkSize significantly, so try to stay with the automatic IP modes when possible.
- Mobile platforms can't handle files larger than 2GB.
- Files larger than 2GB in StreamingAssets are not supported for all 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.