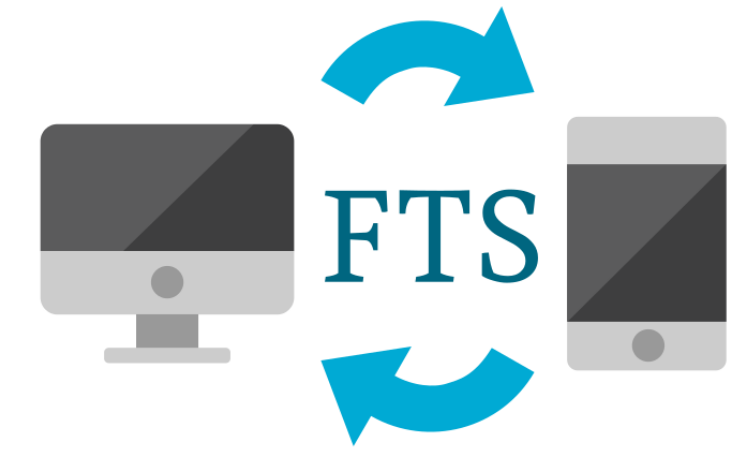
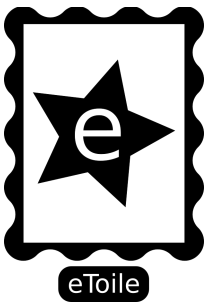


File Transfer Server

File Transfer Server

Cross-platform file transfer solution over UDP.



(eToile 2016) V: 1.2

Index

Introduction

Thank you for purchasing FileTransferServer.
This product is a useful script that enables your application to send and receive large files through a network (UDP) without a dedicated server. FileTransferServer doesn't use WWW class neither FTP protocol.

Saving and reading is as fast as the device admits. There is no limit on file size, just the size on destination's drive.
You can also access the full source code.

The example scene allows testing most of the FileTransferServer features.

This documentation describes the class interfaces and parameters, for code examples please refer to the 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.
This was done in this way to allow multiple instances of the UDP connection without complex coding.

You can access the FileTransferServer interfaces from C# or UnityScript (please note that UnityScript is deprecated).

FileTransferServer uses the FileManagement class for reading and saving files on every supported platform (This class is also included).

There are no any special considerations when exporting to other platforms, neither any special considerations when uploading to digital markets.
Just switch platform from "Build settings" dialog on Unity editor.

Supported platforms are:

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

- Android.

Every action in the FileTransferServer class is event driven. Those events must be added/assigned in editor to be invoked properly.

When FileTransferServer starts execution, automatically establishes an UDP port listener in a parallel thread to avoid blocking the main thread of the application.

TIP: You may take a look at the source code to use the SendString() and SendData() functions (non public) in order to send custom messages between your applications.

Class Integration

To integrate this class to your project you must attach the "FileTransferServer.cs" file to any GameObject in your scene. The FileTransferServer class will start properly by itself depending on the options configured in Unity editor.

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, 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

This is the complete definition of FileTransferServer configuration parameters.

Port

```
[SerializeField] int port = 2600;
```

Sets the UDP port to listen to. The default port is 2600, but it 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 scene.

Enable Server

```
[SerializeField] bool enableServer = 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 another devices.

Max Chunk Size

```
[SerializeField] int maxChunkSize = 65536;
```

This parameter sets the maximum size in bytes of a piece of file to be sent through the FileTransferServer protocol. This value is automatically clamped between 300 and 65236.
By default FileTransferServer will attempt to use the maximum transmit buffer available, depending on your hardware limitations. The UDP connection doesn't admits messages greater than 65536 bytes by definition.
Additionally to the file chunk, a protocol layer is added automatically by FileTransferServer, so the final message size is slightly bigger. If your messages are greater than maxChunkSize, an error message will be printed in the console.
Bigger chunk sizes increases the file transfer rate in high speed networks.

Status Timeout

```
[SerializeField] float statusTimeout = 3f;
```

This parameter sets the time in seconds to assume the remote server is not responding. Due to UDP nature, there are two possible errors in this case:

- Server is offline.
- Package was lost, need to retry.

When timeout is reached, the onStatusTimeOut event is invoked.
This request has not automatic retry, you can retry by code using the onStatusTimeOut event.

Rx File Timeout

```
[SerializeField] float rxFileTimeout = 0.5f;
```

This parameter sets the time in seconds to assume the server is not responding with its own IP. Due to UDP nature, there are two possible errors in this case:

- Server is offline.
- Package was lost, retry automatically.

Rx File Retry Max Cnt

```
[SerializeField] int rxFileRetryMaxCnt = 5;
```

This parameter determines how many times FileTransferServer should retry a file request automatically.

When retry count is reached, the onFileTimeout event is invoked.

Rx List Timeout

```
[SerializeField] float rxListTimeout = 1f;
```

This parameter sets the time in seconds to assume the server is not responding with the update list. Due to UDP nature, there are two possible errors in this case:

- Server is offline.
- Package was lost, need to retry.

When timeout is reached, the onUpdateTimeout event is invoked.
This request has not automatic retry, you can retry by code using the onUpdateTimeout event.

FileTransferServer Events

This is the complete definition of FileTransferServer status events.

On Status Response

```
public UnityEvent onStatusReponse;
```

This event is invoked when a device responds with its own IP. It doesn't makes difference between enabled and disabled servers, it always calls because is intended as an internal "Sever List" change.

On Status Timeout

```
public UnityEvent onStatusTimeout;
```

This event is invoked when no device responds to the status call. There is no automatic retry on this state.

On File Download

```
public UnityEvent onFileDownload;
```

This event is invoked when a complete file was downloaded and saved to disk.
This protocol relies on CRC16 UDP check-sum for file integrity, so it assumes that every received message (file part) is correct.

On File Not Found

```
public UnityEvent onFileNotFound;
```

This event is invoked when the requested server responds with a "File not found" message.
This command is also used internally to skip the download in progress and head to the next one.

On File Timeout

```
public UnityEvent onFileTimeout;
```

This event is invoked when server is not responding to the file request for rxFileRetryMaxCnt times.
The timeout event doesn't stops the download, it continues retrying. To abort the download in progress, call the AbortDownloadInProgress() function, so the FileTransferServer will continue with next download.

On List Download

```
public UnityEvent onListDownload;
```

Internally there is only one download list, and every file request is added to this list. This event is invoked when this list is empty.

It doesn't ensures that every file was received properly, some file may not exists on server side.

On List Not Found

```
public UnityEvent onListNotFound;
```

This event is invoked when the server responds with a "list not found" message. The list is a file in server side.

On Update Timeout

```
public UnityEvent onUpdateTimeout;
```

This event is invoked when the server is not responding to the "update list" request.
There is no automatic retry on this state.

FileTransferServer Client Interfaces

The FileTransferServer interfaces are divided in three main groups:

- Client interfaces: Requests to a remote server.
- Server interfaces: Local server related.
- Progress interfaces: Download in progress status.

CheckServerStatus()

```
public void CheckServerStatus(string serverIP)
```

Sends a status request message to the serverIP address.

You can send a broadcast request to detect every supported device connected to the network terminating the serverIP address with 255.

If your local IP is 192.168.0.10 then use 192.168.0.255.

This example requests server status in broadcast:

```
string[] ipParts = Network.player.ipAddress.Split('.');
string serverIP = ipParts[0] + "."+ ipParts[1] + "."+ ipParts[2] + ".255";
fts.CheckServerStatus(serverIP);
```

GetServerList()

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

Retrieves the internal server list. Valid servers are added and deleted automatically on every "ServerStatus" response message.

If server responds and is enabled, then is added. If server responds and is disabled, then is deleted.

This example requests the server list:

```
List<string> validServers = fts.GetServerList();
```

ResetServerList()

```
public void ResetServerList()
```

Deletes the list of valid servers.

Use this to purge the server list, then send a new server status request.

This example deletes the server list:

```
fts.ResetServerList ();
```

RequestFile()

```
public void RequestFile(string serverIP, string file, string savePath = "",
bool fullPath = false)
public void RequestFile(int serverIndex, string file, string savePath = "",
bool fullPath = false)
```

Requests a single file. The file is added to the queued 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 0).

The savePath argument sets the destination folder of the downloaded file.

The fullPath argument allows to use the savePath as an absolute path.

This example requests a file to the second server in the list:

```
fts.RequestFile(1, "capture.jpg");
```

RequestUpdateList()

```
public void RequestUpdateList(string serverIP, string file, string savePath = "",
bool fullPath = false)
public void RequestUpdateList(int serverIndex, string file, string savePath = "",
bool fullPath = false)
```

Requests a list to start a complete update.

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 0).

The savePath argument sets the destination folder of the downloaded files, all of them.

The fullPath argument allows to use the savePath as an absolute path.

This example requests list update to the second server:

```
fts.RequestUpdateList (1, "list.txt");
```

AbortDownloadInProgress()

```
public void AbortDownloadInProgress()
```

Aborts the download in progress and deletes all temporary files.

Use this function if you decide to abort instead of retrying the download of a file (if server is disconnected).

This example aborts the download in progress:

```
fts.AbortDownloadInProgress();
```

AbortDownloadList()

```
public void AbortDownloadList()
```

Aborts the download in progress, deletes all temporary files and clears the download list preventing from further downloads.

This example aborts the download in progress:

```
fts.AbortDownloadList ();
```

FileTransferServer Server Interfaces

The FileTransferServer interfaces are divided in three main groups:

- Client interfaces: Requests to a remote server.
- Server interfaces: Local server related.
- Progress interfaces: Download in progress status.

SetLocalServerStatus()

```
public void SetLocalServerStatus(bool enabled)
```

Enables or disables the local server capabilities.
The UDP connection still enabled and client capabilities still also operatives.

This example disables local server:

```
fts.SetLocalServerStatus(false);
```

SetMaxChunkSize()

```
public void SetMaxChunkSize(int chunk)
```

Modifies the maximum chunk size in bytes.
The maxChunkSize will be assigned immediately, being clamped between 300 and 65236. If the available SendBufferSize (hardware limitation) is smaller than the requested chunk size, it will be automatically lowered to SendBufferSize minus 300.
You can modify the chunk size dynamically, but don't do that during a transmission or the served file will be corrupted.

This example sets the chunk size for iOS:

```
fts.SetMaxChunkSize(9000);
```

GetMaxChunkSize()

```
public int GetMaxChunkSize()
```

Returns the assigned maxChunkSize for verification purposes.

This example gets the available maximum chunk size:

```
int chunkSize = fts.GetMaxChunkSize ();
```

SendFile()

```
public void SendFile(string ip, string name)
```

Sends a single file to a client. This functions fires the download process in the client side.
This message can be sent in broadcast.

This example send a file in broadcast:

```
fts.SendFile ("192.168.0.255","text.txt");
```

SendUpdate()

```
public void SendUpdate(string ip, string list)
```

Sends the update list to a client. This functions fires the download process in the client side.
This message can be sent in broadcast.

This example send a list in broadcast:

```
fts.SendUpdate ("192.168.0.255","list.txt");
```

FileTransferServer Progress Interfaces

The FileTransferServer interfaces are divided in three main groups:

- Client interfaces: Requests to a remote server.
- Server interfaces: Local server related.
- Progress interfaces: Download in progress status.

GetCurrentFile()

```
public string GetCurrentFile()
```

This method returns the name of the file being downloaded. Please note that the returned name contains the server path (if any). Use FileManagement.GetFileName() to filter the path name. The current file name is available during the onFileDownload event too.

This example gets the downloaded file:

```
string currentFile = fts.GetCurrentFile ();
```

GetCurrentPartialStatus()

```
public string GetCurrentPartialStatus()
```

Gets a progress string formatted as "part/total" for the download in progress.

This example gets the partial status:

```
string status = fts.GetCurrentPartialStatus ();
```

GetCurrentPartialProgress()

```
public float GetCurrentPartialProgress()
```

Gets a progress float from 0 to 1 for the download in progress.

This example gets the partial status:

```
float percent = 100f * fts.GetCurrentPartialProgress();
```

Process of restoring a file

Every part of the file being downloaded is saved into the tempFolder (default name = "FTS_TempDownload") with the same original file name plus an order number.

Once every part of the requested file is downloaded, the existing file is overwritten in destination with the new one that has been downloaded.

There are two ways of restoring the downloaded file:

- In disk (default): This method is optimal. It reads a partial file, adds its content to the main final file and then deletes the temporary partial file. Saves disk and memory.
- In memory (#define RESTORE_IN_RAM): This method is not optimal, but

is compatible with all (future) platforms. Restores the complete file in memory while deletes every temporary partial file in the process. Once the complete restored file was loaded into memory, it's written to disk. Disk use is optimized, but may run out of memory if files are too large.

If a download was interrupted or aborted because application was forced to close, the next time the application starts deletes every partial temporary file that may exist into the tempFolder.

The downloaded file is not saved in the same directory hierarchy that was originally in server side. The file is saved in the provided directory when the download/update is requested.

Every downloaded file is written to the persistent data path if no alternative path was provided.

If you need local folders, you can create them easily with FileManagement.

Known Issues

- Don't run two instances of the FTS in the same device with the same IP and the same PORT. At least one of them must be different, that is a UDP limitation.

- You can connect through internet, is mandatory to use a fixed IP (without any Proxy) or a VPN and forward the desired port in the destination router.

Contact

If you need some new interfaces or if you find some errors in this documentation or the application, don't hesitate on send me an email: jmonsuarez@gmail.com

Please, once you have tested this product, take a minute of your time to write a good review in the Unity Asset Store, so you will help to improve this product.

https://www.assetstore.unity3d.com/#!/content/73518

Thanks a lot.