Photon DotNet Client Reference

for Windows, Unity3D, PSM, Xamarin, Windows 8 Store and Phone SDKs

Table of Contents

1 Overview

Photon is a development framework to build real-time multiplayer games and applications for various platforms. It consists of a Server SDK and Client SDKs for several platforms.

This is the documentation and reference for the Photon Client Library.

Additional documentation and help is available online. Visit: doc.exitgames.com/photon-server and forum.exitgames.com

Photon provides a low-latency communication-layer based on UDP (or alternatively TCP). It enables reliable and unreliable transfer of data in "commands". On top of this, an operation- and event-framework is established to ease development of your own games.

Each game is different, so we developed several "server applications" which provide a basic logic and included them in the server SDK as example and code base.

- The "Lite" application offers the basic operations that we felt useful for most room-based multiplayer games. We included it's operations and **events** are part of of the client API.
- The "Loadbalancing" application extends the "Lite" application and allows you to run multiple servers for your game. A master server coordinates the games creation and joining. There is a special API in the SDK that makes use of this application.
- The "MMO" application focuses on seamless worlds and it's client side api is available in source in the MMO application's code (as it shares code with the server side).

Photon Cloud is a hosted service for your Photon games. To use it, register at cloud.exitgames.com and use the Loadbalancing API and demos to start your development.

1.1 Photon Workflow

To get an impression of how to work on the client, we will use the server's Lite logic. This application defines rooms which are created when users try to join them. Each user in a room becomes an actor with her own number.

A simplified workflow looks like this:

- create a **LitePeer** instance
- from now on: regularly call Service to get **events** and send commands (e.g. ten times a second)
- call Connect to connect the server
  - wait until the library calls **OnStatusChanged**
  - the returned status int should equal **StatusCode.Connect**
- call OpJoin to get into a game
  - wait until the library calls OnOperationResponse with opCode: **LiteOpCode.Join**
- send data in the game by calling **OpRaiseEvent**
- receive **events** in **OnEvent**

- The Lite Application defines several useful **events** for common situations: Someone joins or leaves the room.
    - InLite, **events** created by calling OpRaiseEvent will be received by others in the same room in this method.
- when you are done: call **LitePeer.OpLeave** to quit/leave the game
    - wait for "leave" return in OnOperationResponse with opCode: **LiteOpCode.Leave**
- disconnect with Disconnect
    - check "disconnect" return in OnStatusChanged with statusCode: **StatusCode.Disconnect**

Combined with the server's Lite application, this simple workflow would allow you to use rooms and send your game's **events**. The methods used could be broken down into three layers:

- **Low Level:** Service, Connect, Disconnect and the OnStatusChanged are directly referring to the connection to the server. This level works with UDP/TCP packets which transport commands (which in turn carry your operations). It keeps your connection alive and organizes your RPC calls and **events** into packages.
- **Logic Level: Operations**, results and **events** make up the logical level in Photon. Any operation defined on the server (think RPC call) and can have a result. **Events** are incoming from the server and update the client with some data.
- **Application Level:** Made up by a specific application and its features. In this case we use the operations and logic of the Lite application. In this specific case, we have rooms and actors and more. The **LitePeer** is matching the server side implementation and wraps it up for you.

You don't have to manage the low level communication in most cases. However, it makes sense to know that everything that goes from client to server (and the other way round) is put into "commands". Internally, commands are also used to establish and keep the connection between client and server alive (without carrying additional data).

All methods that are operations (RPC calls) are prefixed with "Op" to tell them apart from anything else. Other server side applications (like MMO or your own) will define different operations. These will have different parameters and return values. These operations are not part of the client library but can be implemented by calling OpCustom().

The interface **IPhotonPeerListener** must be implemented for callbacks. They are:

- **OnStatusChanged** is for peer state-changes (connect, disconnect, errors, compare with **StatusCode Enumeration**)
- **OnOperationResponse** is the callback for operations (join, leave, etc.)
- **OnEvent** as callback for **events** coming in
- **DebugReturn** as callback to debug output (less frequently used by release builds)

The following properties in **PhotonPeer** are of special interest:

- **TimePingInterval** sets the time between ping-operations
- **RoundTripTime** of reliable operations to the server and back
- **RoundTripTimeVariance** shows the variability of the roundtrip time
- **ServerTimeInMilliSeconds** is the continuously approximated server's time

1.2 **Operations**

Operation is our term for remote procedure calls (RPC) on Photon. This in turn can be described as methods that are implemented on the server-side and called by clients. As any method, they have parameters and return values. The Photon development framework takes care of getting your RPC calls from clients to server and results back.

Server-side, operations are part of an application running on top of Photon. The default application provided by Exit Games is called "Lite Application" or simply Lite. The **LitePeer** class extends the **PhotonPeer** by methods for each of the Lite Operations.

Examples for Lite Operations are "join" and "raise event". On the client side, they can be found in the **LitePeer** class as methods: OpJoin and OpRaiseEvent. They can be used right away with the default implementation of Photon and the Lite Application.

Custom Operations

Photon is extendable with features that are specific to your game. You could persist world states or double check information from the clients. Any operation that is not in Lite or the MMO application logic is called Custom Operation. Creating those is primarily a server-side task, of course, but the clients have to use new functions / operations of the server.

So Operations are methods that can be called from the client side. They can have any number of parameters and any name. To preserve bandwidth, we assign byte-codes for every operation and each parameter. The definition is done server side. Each Operation has its own, unique number to identify it, known as the operation code (opCode). An operation class defines the expected parameters and assigns a parameter code for each. With this definition, the client side only has to fill in the values and let the server know the opCode of the Operation.

Photon uses Dictionaries to aggregate parameters for operation requests, responses and **events**. Use OpCustom to call any operation, providing the parameters in a Dictionary.

Client side, opCode and parameter-codes are currently of type byte (to minimize overhead). They need to match the definition of the server side to successfully call your operation.

1.3 Events

Unlike operations, events are "messages" that are rarely triggered by the client that receives them. Events come from outside: the server or other clients.

They are created as side effect of operations (e.g. when you join a room) or raised as main purpose of the operation Raise Event. Most events carry some form of data but in rare cases the type of event itself is the message.

Events are (once more) Dictionaries with arbitrary content. In the "top-level" of an event, bytes are used as keys for values. The values can be of any serializable type. The Lite Application, e.g., uses a Hashtable for custom event content in its operation RaiseEvent.

1.4 Fragmentation and Channels

Fragmentation

Bigger data chunks of data won't fit into a single package, so they are fragmented and reassembled automatically. Depending on the data size, a single operation or event can be made up of multiple packages.

Be aware that this might stall other commands. Call Service() or SendOutgoingCommands() more often than absolutely necessary. You should check that **PhotonPeer.QueuedOutgoingCommands** is becoming zero regularly to make sure everything gets out. You can also check the debug output for "UDP package is full", which can happen from time to time but should not happen permanently.

Maximum Transfer Unit

The maximum size for any UDP package can be configured by setting **PhotonPeer.MaximumTransferUnit Property**. By default, this is 1200 bytes. Some routers will fragment even this UDP package size. If you don't need bigger sizes, go for 512 bytes per package, which is more overhead per command but potentially safer.

This setting is ignored by TCP connections, which negotiate their MTU internally.

Sequencing

The sequencing of the protocol makes sure that any receiving client will Dispatch your actions in the order you sent them. Unreliable data is considered replaceable and can be lost. Reliable **events** and operations will be repeated several times if needed but they will all be Dispatched in order without gaps. Unreliable actions are also related to the last reliable action and not Dispatched before that reliable data was Dispatched first. This can be useful, if the **events** are related to each other.

Example: Your FPS sends out unreliable movement updates and reliable chat messages. A lost package with movement updates would be left out as the next movement update is coming fast. An the receiving end, this would maybe show as a small jump. If a package with a chat message is lost, this is repeated and would introduce lag, even to all movement updates after the message was created. In this case, the data is unrelated and should be put into different channels.

Channels

The DotNet clients and server are now supporting "channels". This allows you to separate information into multiple channels, each being sequenced independently. This means, that **Events** of one channel will not be stalled because **events** of another channel are not available.

By default an **PhotonPeer** has two channels and channel zero is the default to send operations. The operations join and leave are always sent in channel zero (for simplicity). There is a "background" channel 255 used internally for connect and disconnect messages. This is ignored for the channel count.

Channels are prioritized: the lowest channel number is put into a UDP package first. Data in a higher channel might be sent later when a UDP package is already full.

Example: The chat messages can now be sent in channel one, while movement is sent in channel zero. They are not related and if a chat message is delayed, it will no longer affect movement in channel zero. Also, channel zero has higher priority and is more likely to be sent (in case packages get filled up).

1.5 **Using TCP**

A **PhotonPeer** could be instanced with TCP as underlying protocol if necessary. This is not best practice but some platforms don't support UDP sockets. This is why Silverlight (e.g.) uses TCP in all cases.

The Photon Client API is the same for both protocols but there are some differences in what goes on under the hood.

Everything sent over TCP is always reliable, even if you call your operations as unreliable!

If you use only TCP clients

Simply send any operation unreliable. It saves some work (and bandwidth) in the underlying protocols.

If you have TCP and UDP clients

Anything you send between the TCP clients will always be transferred reliable. But as you communicate with some clients that use UDP these will get your **events** reliable or unreliable.

Example:

A Silverlight client might send unreliable movement updates in channel # 1. This will be sent via TCP, which makes it reliable. Photon however also has connections with UDP clients (like a 3D downloadable game client). It will use your reliable / unreliable settings to forward your movement updates accordingly.

1.6 **Network Simulation**

During development, most tests will be done in a local network. Once released, the clients will communicate through the internet, which has a higher delay per message and in some cases even drops messages entirely.

To prepare a game for real-life conditions, the Photon client libraries let you simulate some effects of internet-communication: lag, jitter and packet loss.

- **Lag / Latency**: a more or less constant delay of messages between client and server. Either direction can be affected in a different way but usually the values are close to another. Affects the roundtrip time.
- **Jitter**: Is randomizes the Lag in the simulation. This affects the variance of the roundtrip time. Udp packages can get out of order this way, which also is simulated. The new lag will be: Lag + [-JitterValue..+JitterValue]. This keeps the mean Lag at the setting and some packages are actually faster than the Lag value implies.
- **Packet Loss**: UPD packages can become lost. In the Photon protocol, commands that are flagged as reliable will be repeated while other commands (operations) might get lost this way.

The lag simulation is running in its own Thread which tries to meet delays defined in the settings. In most cases, they can be met but actual delays will have a variance of up to +/- 20ms.

Using Network Simulation

By default, Network Simulation is turned off. It can be turned on by setting **PhotonPeer.IsSimulationEnabled** and the settings are aggregated into a **NetworkSimulationSet**, also accessible by the peer class (e.g. the **LitePeer**).

Code Sample:

//Activate / Deactivate:

this.peer.IsSimulationEnabled = true;

//Raise Incoming Lag:

this.peer.NetworkSimulationSettings.IncomingLag = 300; //default is 100ms

//add 10% of outgoing loss:

this.peer.NetworkSimulationSettings.OutgoingLossPercentage = 10; //default is 1

//this property counts the actual simulated loss:

this.peer.NetworkSimulationSettings.LostPackagesOut;

1.7 **Serializable Datatypes**

Starting with Photon Server SDK 1.8.0 and DotNet SDK 5.6.0, the set of serializable datatypes is changed. The ArrayList was removed but aside from Arrays and Hashtables every serializable type can also be sent as array (e.g. String[], Float[]). Only one-dimensional arrays are supported currently.

Photon 1.8.0 and higher

String / string

Boolean / bool

Byte / byte (unsigned! can be cast to SByte / sbyte to be eqivalent to Java's byte)

Int16 / short (signed)

Int32 / int (signed)

Int64 / long

Single / float

Double / double

Array (of the types above, with a max number of Short.MaxValue enties).

Hashtable (not available as array)

The following data types can be used within **events** as keys and values on Neutron:

Photon 1.6.0 and lower

String

Boolean

Byte / byte (unsigned! can be cast to SByte / sbyte to be equivalent to Java's byte)

Int16 (equals Short in Java)

Int32 (equals Integer in Java)

Int64 (equals Long in Java)

ArrayList (equals Vector in Java)

Hashtable

sbyte[]

int[]

String[]

1.8 The Photon Server

The Photon Server is the central hub for communication for all your clients. It is a service that can be run on any Windows machine, handling the UDP and TCP connections of clients and hosting a DotNet runtime layer with your own business logic, called application.

The Photon Server SDK includes several applications in source and pre-built. You can run them out of the box or develop your own server logic.

Get the Photon Server SDK at: photon.exitgames.com

1.9 Lite Application

The Lite Application is the example application for room-based games on Photon and (hopefully) a flexible basis for your own games. It offers rooms, joining and leaving them, sending **events** to the other players in a room and handles properties.

The Lite Application is tightly integrated with the client libraries and used as example throughout most documentation.

1.9.1 Properties on Photon

The Lite Application implements a general purpose mechanism to set and fetch key/value pairs on the server side (in memory). They are associated to a room/game or a player within a room and can be fetched or updated by anyone in that game.

Each entry in the properties Hashtable is considered a separate property and can be overwritten independently. The value of a property can be of any **serializable datatype**. The keys must be either of type string or byte. Bytes are preferred, as they mean the less overhead.

To avoid confusion, don't mix string and byte as key-types. Mixed types of keys, require separate requests to fetch them.

Property broadcasting and events

Property changes in a game can be "broadcasted", which triggers **events** for the other players to update them. The player who changed the property does not get the update (again).

Any change that uses the broadcast option will trigger a property update event. This event carries the changed properties (only), who changed the properties and where the properties belong to.

Your clients need to "merge" the changes (if properties are cached at all).

Properties can be set by these methods:

- **LitePeer.OpSetPropertiesOfActor Method** sets a player's properties
- **LitePeer.OpSetPropertiesOfGame Method** sets a game's properties
- **LitePeer.OpJoin Method** also allows you to set properties if the game did not exist yet

And fetched with these methods:

- OpGetPropertiesOfActor
- OpGetPropertiesOfGame

Broadcast Events

Any change that uses the broadcast option will trigger a property update event **LiteEventCode.PropertiesChanged**. This event carries the properties as value of key **LiteEventKey.Properties**.

Additionally, there is information about who changed the properties in key **LiteEventKey.ActorNr**.

The key **LiteEventKey.TargetActorNr** will only be available if the property-set belongs to a certain player. If it's not present, the properties are game-properties.

Notes

The current Lite application is not able to delete properties and does not support wildcard characters in string keys to fetch properties.

Other types of keys could be used but to keep things simple, we decided against adding those. If needed, we would help you with the implementation.

The property handling is likely to be updated and extended in the future.

1.10 **Custom Authentication**

By default, all users are allowed to connect to your Photon Cloud applications. With "Custom Authentication", you can combine Photon with more or less any community, check account credentials and customize who has access to your game.

The Photon Cloud Dashboard lets you configure a web service to check when a user connects. The client's credentials will be passed on to that service (via https).

Credentials in this case are anything you want to pass. It can even be a token/session provided by a third party service that the clients contact before connecting to Photon.

Values for CustomAuthentication are stored in the class AuthenticationValues. LoadBalancingClient.Connect has an overload with AuthenticationValues. Alternatively you can set the values as property: LoadBalancingClient.CustomAuthenticationValues.

Details about the workflow and the communication between Photon and the custom authentication service can be found online:

http://doc.exitgames.com/photon-cloud/CustomAuthentication

This is a LoadBalancing and Photon Cloud feature and it's not available in Lite.

1.11 **Further Help**

Developer Network

Visit: doc.exitgames.com/photon-server

Developer Forum

Visit: forum.exitgames.com

Mail Support

Don't hesitate to mail us: developer@exitgames.com

2 Symbol Reference

2.1 Classes

The following table lists classes in this documentation.

Classes

	Name	Description
	EventData	Contains all components of a Photon **Event**. **Event Parameters**, like OperationRequests and OperationResults, consist of a Dictionary with byte-typed keys per value.
	LiteEventCode	Lite - **Event** codes. These codes are defined by the Lite application's logic on the server side. Other application's won't necessarily use these.
	LiteEventKey	Lite - Keys of event-parameters that are defined by the Lite application logic. To keep things lean (in terms of bandwidth), we use byte keys to identify values in **events** within Photon. In Lite, you can send custom **events** by defining a EventCode and some content. This custom content is a Hashtable, which can use any type for keys and values. The parameter for operation RaiseEvent and the resulting **Events** use key (byte)245 for the custom content. The constant for this is: **Data** or **LiteEventKey.CustomContent Field**.
	LiteOpCode	Lite - Operation Codes. This enumeration contains the codes that are given to the Lite Application's operations. Instead of sending "**Join**", this enables us to send the byte 255.
	LiteOpKey	Lite - keys for parameters of operation requests and responses (short: OpKey).
	LitePeer	A LitePeer is an extended **PhotonPeer** and implements the operations offered by the "Lite" Application of the Photon Server SDK.
	NetworkSimulationSet	A set of network simulation settings, enabled (and disabled) by **PhotonPeer.IsSimulationEnabled**.
	OperationRequest	Container for an Operation request, which is a code and parameters.
	OperationResponse	Contains the server's response for an operation called by this peer. The indexer of this class actually provides access to the **Parameters** Dictionary.
	PhotonPeer	Instances of the PhotonPeer class are used to connect to a Photon server and communicate with it.
	Protocol	Provides tools for the Exit Games Protocol
	PRPCAttribute	This is class PRPCAttribute.
	SupportClass	Contains several (more or less) useful static methods, mostly used for debugging.
	TrafficStats	This is class TrafficStats.
	TrafficStatsGameLevel	Only in use as long as **PhotonPeer.TrafficStatsEnabled** = true;

2.1.1 EventData Class

Contains all components of a Photon **Event**. **Event Parameters**, like OperationRequests and OperationResults, consist of a Dictionary with byte-typed keys per value.

C#

```
public class EventData;
```

Remarks

The indexer of this class actually provides access to the **Parameters** Dictionary. The operation RaiseEvent of the Lite application allows you to provide custom event content. Defined in Lite, this CustomContent will be made the value of key LiteEventKey.OperationRaiseEvent which is (byte)42. Enums and constants for the Lite-Application codes are defined in the **LitePeer** namespace. Check: **LiteEventKey**, etc.

EventData Fields

	Name	Description
◆	**Code**	The event code identifies the type of event.
◆	**Parameters**	The Parameters of an event is a Dictionary.

EventData Methods

	Name	Description
◆	**this**	Alternative access to the **Parameters**.
◆	**ToString**	ToString() override.
◆	**ToStringFull**	Extensive output of the event content.

2.1.1.1 EventData Fields

2.1.1.1.1 EventData.Code Field

The event code identifies the type of event.

C#

```
public byte Code;
```

2.1.1.1.2 EventData.Parameters Field

The Parameters of an event is a Dictionary.

C#

```
public Dictionary<byte, object> Parameters;
```

2.1.1.2 EventData Methods

2.1.1.2.1 EventData.this Indexer

Alternative access to the **Parameters**.

C#

```
public object this[byte key];
```

Parameters

Parameters	Description
byte key	The key byte-code of a event value.

Returns

The **Parameters** value, or null if the key does not exist in **Parameters**.

2.1.1.2.2 EventData.ToString Method

ToString() override.

C#

```csharp
public override string ToString();
```

Returns

Short output of "**Event**" and it's **Code**.

2.1.1.2.3 EventData.ToStringFull Method

Extensive output of the event content.

C#

```csharp
public string ToStringFull();
```

Returns

To be used in debug situations only, as it returns a string for each value.

2.1.2 LiteEventCode Class

Lite - **Event** codes. These codes are defined by the Lite application's logic on the server side. Other application's won't necessarily use these.

C#

```csharp
public static class LiteEventCode;
```

Remarks

If your game is built as extension of Lite, don't re-use these codes for your custom **events**.

LiteEventCode Fields

	Name	Description
●	**Join**	(255) **Event** Join: someone joined the game
●	**Leave**	(254) **Event** Leave: someone left the game
●	**PropertiesChanged**	(253) **Event** PropertiesChanged

2.1.2.1 LiteEventCode Fields

2.1.2.1.1 LiteEventCode.Join Field

(255) **Event** Join: someone joined the game

C#

```csharp
public const byte Join = 255;
```

2.1.2.1.2 LiteEventCode.Leave Field

(254) **Event** Leave: someone left the game

C#

```csharp
public const byte Leave = 254;
```

2.1.2.1.3 LiteEventCode.PropertiesChanged Field

(253) **Event** PropertiesChanged

C#

```
public const byte PropertiesChanged = 253;
```

2.1.3 LiteEventKey Class

Lite - Keys of event-parameters that are defined by the Lite application logic. To keep things lean (in terms of bandwidth), we use byte keys to identify values in **events** within Photon. In Lite, you can send custom **events** by defining a EventCode and some content. This custom content is a Hashtable, which can use any type for keys and values. The parameter for operation RaiseEvent and the resulting **Events** use key (byte)245 for the custom content. The constant for this is: **Data** or **LiteEventKey.CustomContent Field**.

C#

```
public static class LiteEventKey;
```

Remarks

If your game is built as extension of Lite, don't re-use these codes for your custom **events**.

LiteEventKey Fields

	Name	Description
♦	**ActorList**	(252) List of playernumbers currently in the room.
♦	**ActorNr**	(254) Playernumber of the player who triggered the event.
♦	**ActorProperties**	(249) Key for actor (player) property set (Hashtable).
♦	**CustomContent**	(245) The Lite operation RaiseEvent will place the Hashtable with your custom event-content under this key.
♦	**Data**	(245) Custom Content of an event (a Hashtable in Lite).
♦	**GameProperties**	(248) Key for game (room) property set (Hashtable).
♦	**Properties**	(251) Set of properties (a Hashtable).
♦	**TargetActorNr**	(253) Playernumber of the player who is target of an event (e.g. changed properties).

2.1.3.1 LiteEventKey Fields

2.1.3.1.1 LiteEventKey.ActorList Field

(252) List of playernumbers currently in the room.

C#

```
public const byte ActorList = 252;
```

2.1.3.1.2 LiteEventKey.ActorNr Field

(254) Playernumber of the player who triggered the event.

C#

```
public const byte ActorNr = 254;
```

2.1.3.1.3 **LiteEventKey.ActorProperties Field**

(249) Key for actor (player) property set (Hashtable).

C#

```
public const byte ActorProperties = 249;
```

2.1.3.1.4 **LiteEventKey.CustomContent Field**

(245) The Lite operation RaiseEvent will place the Hashtable with your custom event-content under this key.

C#

```
public const byte CustomContent = Data;
```

Remarks

Alternative for: **Data**!

2.1.3.1.5 **LiteEventKey.Data Field**

(245) Custom Content of an event (a Hashtable in Lite).

C#

```
public const byte Data = 245;
```

2.1.3.1.6 **LiteEventKey.GameProperties Field**

(248) Key for game (room) property set (Hashtable).

C#

```
public const byte GameProperties = 248;
```

2.1.3.1.7 **LiteEventKey.Properties Field**

(251) Set of properties (a Hashtable).

C#

```
public const byte Properties = 251;
```

2.1.3.1.8 **LiteEventKey.TargetActorNr Field**

(253) Playernumber of the player who is target of an event (e.g. changed properties).

C#

```
public const byte TargetActorNr = 253;
```

2.1.4 **LiteOpCode Class**

Lite - Operation Codes. This enumeration contains the codes that are given to the Lite Application's operations. Instead of sending "**Join**", this enables us to send the byte 255.

C#

```
public static class LiteOpCode;
```

Remarks

Other applications (the MMO demo or your own) could define other operations and other codes. If your game is built as

extension of Lite, don't re-use these codes for your custom **events**.

LiteOpCode Fields

	Name	Description
●	**ChangeGroups**	(248) Operation code to change interest groups in Rooms (Lite application and extending ones).
●	**ExchangeKeysForEncryption**	This is ExchangeKeysForEncryption, a member of class LiteOpCode.
●	**GetProperties**	(251) Operation code for OpGetProperties.
●	**Join**	(255) Code for OpJoin, to get into a room.
●	**Leave**	(254) Code for OpLeave, to get out of a room.
●	**RaiseEvent**	(253) Code for OpRaiseEvent (not same as eventCode).
●	**SetProperties**	(252) Code for OpSetProperties.

2.1.4.1 LiteOpCode Fields

2.1.4.1.1 LiteOpCode.ChangeGroups Field

(248) Operation code to change interest groups in Rooms (Lite application and extending ones).

C#

```
public const byte ChangeGroups = 248;
```

2.1.4.1.2 LiteOpCode.ExchangeKeysForEncryption Field

C#

```
[Obsolete("Exchanging encrpytion keys is done internally in the lib now. Don't expect this
operation-result.")]
public const byte ExchangeKeysForEncryption = 250;
```

Description

This is ExchangeKeysForEncryption, a member of class LiteOpCode.

2.1.4.1.3 LiteOpCode.GetProperties Field

(251) Operation code for OpGetProperties.

C#

```
public const byte GetProperties = 251;
```

2.1.4.1.4 LiteOpCode.Join Field

(255) Code for OpJoin, to get into a room.

C#

```
public const byte Join = 255;
```

2.1.4.1.5 LiteOpCode.Leave Field

(254) Code for OpLeave, to get out of a room.

C#

```
public const byte Leave = 254;
```

2

2.1.4.1.6 **LiteOpCode.RaiseEvent Field**

(253) Code for OpRaiseEvent (not same as eventCode).

C#

```
public const byte RaiseEvent = 253;
```

2.1.4.1.7 **LiteOpCode.SetProperties Field**

(252) Code for OpSetProperties.

C#

```
public const byte SetProperties = 252;
```

2.1.5 **LiteOpKey Class**

Lite - keys for parameters of operation requests and responses (short: OpKey).

C#

```
public static class LiteOpKey;
```

Remarks

These keys match a definition in the Lite application (part of the server SDK). If your game is built as extension of Lite, don't re-use these codes for your custom **events**.

These keys are defined per application, so Lite has different keys than MMO or your custom application. This is why these are not an enumeration. Lite and Lite Lobby will use the keys 255 and lower, to give you room for your own codes.

Keys for operation-parameters could be assigned on a per operation basis, but it makes sense to have fixed keys for values which are used throughout the whole application.

LiteOpKey Fields

	Name	Description
♦	**ActorList**	(252) **Code** for list of players in a room. Currently not used.
♦	**ActorNr**	(254) **Code** of the Actor of an operation. Used for property get and set.
♦	**ActorProperties**	(249) **Code** for property set (Hashtable).
♦	**Add**	(238) The "Add" operation-parameter can be used to add something to some list or set. E.g. add groups to player's interest groups.
♦	**Asid**	(255) **Code** of the room name. Used in OpJoin (Asid = Application Session ID).
♦	**Broadcast**	(250) **Code** for broadcast parameter of OpSetProperties method.
♦	**Cache**	(247) **Code** for caching **events** while raising them.
♦	**Code**	(244) Code used when sending some code-related parameter, like OpRaiseEvent's event-code.
♦	**Data**	(245) **Code** of data of an event. Used in OpRaiseEvent.
♦	**GameId**	(255) **Code** of the game id (a unique room name). Used in OpJoin.
♦	**GameProperties**	(248) **Code** for property set (Hashtable).
♦	**Group**	(240) **Code** for "group" operation-parameter (as used in Op RaiseEvent).
♦	**Properties**	(251) **Code** for property set (Hashtable). This key is used when sending only one set of properties. If either **ActorProperties** or **GameProperties** are used (or both), check those keys.
♦	**ReceiverGroup**	(246) **Code** to select the receivers of **events** (used in Lite, Operation RaiseEvent).

●	**Remove**	(239) The "Remove" operation-parameter can be used to remove something from a list. E.g. remove groups from player's interest groups.
●	**RoomName**	(255) **Code** of the room name. Used in OpJoin.
●	**TargetActorNr**	(253) **Code** of the target Actor of an operation. Used for property set. Is 0 for game

2.1.5.1 **LiteOpKey Fields**

2.1.5.1.1 **LiteOpKey.ActorList Field**

(252) **Code** for list of players in a room. Currently not used.

C#

```
public const byte ActorList = 252;
```

2.1.5.1.2 **LiteOpKey.ActorNr Field**

(254) **Code** of the Actor of an operation. Used for property get and set.

C#

```
public const byte ActorNr = 254;
```

2.1.5.1.3 **LiteOpKey.ActorProperties Field**

(249) **Code** for property set (Hashtable).

C#

```
public const byte ActorProperties = 249;
```

2.1.5.1.4 **LiteOpKey.Add Field**

(238) The "Add" operation-parameter can be used to add something to some list or set. E.g. add groups to player's interest groups.

C#

```
public const byte Add = 238;
```

2.1.5.1.5 **LiteOpKey.Asid Field**

(255) **Code** of the room name. Used in OpJoin (Asid = Application Session ID).

C#

```
[Obsolete("Use GameId")]
public const byte Asid = 255;
```

2.1.5.1.6 **LiteOpKey.Broadcast Field**

(250) **Code** for broadcast parameter of OpSetProperties method.

C#

```
public const byte Broadcast = 250;
```

2.1.5.1.7 **LiteOpKey.Cache Field**

(247) **Code** for caching **events** while raising them.

C#

```
public const byte Cache = 247;
```

2.1.5.1.8 LiteOpKey.Code Field

(244) Code used when sending some code-related parameter, like OpRaiseEvent's event-code.

C#

```
public const byte Code = 244;
```

Remarks

This is not the same as the Operation's code, which is no longer sent as part of the parameter Dictionary in Photon 3.

2.1.5.1.9 LiteOpKey.Data Field

(245) **Code** of data of an event. Used in OpRaiseEvent.

C#

```
public const byte Data = 245;
```

2.1.5.1.10 LiteOpKey.GameId Field

(255) **Code** of the game id (a unique room name). Used in OpJoin.

C#

```
public const byte GameId = 255;
```

2.1.5.1.11 LiteOpKey.GameProperties Field

(248) **Code** for property set (Hashtable).

C#

```
public const byte GameProperties = 248;
```

2.1.5.1.12 LiteOpKey.Group Field

(240) **Code** for "group" operation-parameter (as used in Op RaiseEvent).

C#

```
public const byte Group = 240;
```

2.1.5.1.13 LiteOpKey.Properties Field

(251) **Code** for property set (Hashtable). This key is used when sending only one set of properties. If either **ActorProperties** or **GameProperties** are used (or both), check those keys.

C#

```
public const byte Properties = 251;
```

2.1.5.1.14 LiteOpKey.ReceiverGroup Field

(246) **Code** to select the receivers of **events** (used in Lite, Operation RaiseEvent).

C#

```
public const byte ReceiverGroup = 246;
```

2.1.5.1.15 **LiteOpKey.Remove Field**

(239) The "Remove" operation-parameter can be used to remove something from a list. E.g. remove groups from player's interest groups.

C#

```
public const byte Remove = 239;
```

2.1.5.1.16 **LiteOpKey.RoomName Field**

(255) **Code** of the room name. Used in OpJoin.

C#

```
[Obsolete("Use GameId")]
public const byte RoomName = 255;
```

Remarks

Alternative for: **Asid**!

2.1.5.1.17 **LiteOpKey.TargetActorNr Field**

(253) **Code** of the target Actor of an operation. Used for property set. Is 0 for game

C#

```
public const byte TargetActorNr = 253;
```

2.1.6 **LitePeer Class**

A LitePeer is an extended **PhotonPeer** and implements the operations offered by the "Lite" Application of the Photon Server SDK.

C#

```
public class LitePeer : PhotonPeer;
```

Remarks

This class is used by our samples and allows rapid development of simple games. You can use rooms and properties and send **events**. For many games, this is a good start.

Operations are prefixed as "Op" and are always asynchronous. In most cases, an OperationResult is provided by a later call to OnOperationResult.

Methods

	Name	Description
≡●	**PhotonPeer**	Creates a new PhotonPeer instance to communicate with Photon. Connection is UDP based, except for Silverlight.

LitePeer Class

	Name	Description
≡●?	**LitePeer**	Creates a LitePeer instance to connect and communicate with a Photon server. Uses UDP as protocol (except in the Silverlight library).

PhotonPeer Methods

|		Name	Description
⇒◆𝕍		**Connect**	This method does a DNS lookup (if necessary) and connects to the given serverAddress. The return value gives you feedback if the address has the correct format. If so, this starts the process to establish the connection itself, which might take a few seconds. When the connection is established, a callback to **IPhotonPeerListener.OnStatusChanged** will be done. If the connection can't be established, despite having a valid address, the OnStatusChanged is called with an error-value. The applicationName defines the application logic to use server-side and it should match the name of one of the apps in your server's config. By default,... **more**
⇒◆𝕍		**Disconnect**	This method initiates a mutual disconnect between this client and the server.
⇒◆𝕍		**DispatchIncomingCommands**	This method directly causes the callbacks for **events**, responses and state changes within a **IPhotonPeerListener**. DispatchIncomingCommands only executes a single received command per call. If a command was dispatched, the return value is true and the method should be called again. This method is called by **Service**() until currently available commands are dispatched.
⇒◆		**EstablishEncryption**	This method creates a public key for this client and exchanges it with the server.
⇒◆𝕍		**FetchServerTimestamp**	This will fetch the server's timestamp and update the approximation for property ServerTimeInMilliseconds. The server time approximation will NOT become more accurate by repeated calls. Accuracy currently depends on a single roundtrip which is done as fast as possible. The command used for this is immediately acknowledged by the server. This makes sure the roundtrip time is low and the timestamp + rountriptime / 2 is close to the original value.
⇒◆𝕍		**OpCustom**	Allows the client to send any operation to the Photon Server by setting any opCode and the operation's parameters.
⇒◆𝕊		**RegisterType**	Registers new types/classes for de/serialization and the fitting methods to call for this type.
⇒◆𝕍		**SendAcksOnly**	This is SendAcksOnly, a member of class PhotonPeer.
⇒◆𝕍		**SendOutgoingCommands**	This method creates a UDP/TCP package for outgoing commands (operations and acknowledgements) and sends them to the server. This method is also called by **Service**().
⇒◆𝕍		**Service**	This method excutes **DispatchIncomingCommands** and **SendOutgoingCommands** in your application Thread-context.
⇒◆𝕍		**StopThread**	This method immediately closes a connection (pure client side) and ends related listening Threads.
⇒◆		**TrafficStatsReset**	Creates new instances of **TrafficStats** and starts a new timer for those.
⇒◆		**VitalStatsToString**	Returns a string of the most interesting connection statistics. When you have issues on the client side, these might contain hints about the issue's cause.

LitePeer Class

		Name	Description
⇒◆𝕍		**OpChangeGroups**	Operation to handle this client's interest groups (for **events** in room).
⇒◆𝕍		**OpGetProperties**	Gets all properties of the game and each actor.
⇒◆𝕍		**OpGetPropertiesOfActor**	Gets selected properties of some actors.
⇒◆𝕍		**OpGetPropertiesOfGame**	Gets selected properties of current game.

⇒◆ Ⅶ		**OpJoin**	This operation will join an existing room by name or create one if the name is not in use yet.
			Rooms (or games) are simply identified by name. We assume that users always want to get into a room - no matter if it existed before or not, so it might be a new one. If you want to make sure a room is created (new, empty), the client side might come up with a unique name for it (make sure the name was not taken yet).
			The application "Lite Lobby" lists room names and effectively allows the user to... **more**
⇒◆ Ⅶ		**OpLeave**	Leave operation of the Lite Application (also in Lite Lobby). Leaves a room / game, but keeps the connection. This operations triggers the event **LiteEventCode.Leave** for the remaining clients. The event includes the actorNumber of the player who left in key **LiteEventKey.ActorNr**.
⇒◆ Ⅶ		**OpRaiseEvent**	RaiseEvent tells the server to send an event to the other players within the same room.
⇒◆ Ⅶ		**OpSetPropertiesOfActor**	Attaches or updates properties of the specified actor.
⇒◆ Ⅶ		**OpSetPropertiesOfGame**	Attaches or updates properties of the current game.

PhotonPeer Properties

	Name	Description
🗐	**ByteCountCurrentDispatch**	Gets the size of the dispatched event or operation-result in bytes. This value is set before OnEvent() or OnOperationResponse() is called (within **DispatchIncomingCommands**()).
🗐	**ByteCountLastOperation**	Gets the size of the last serialized operation call in bytes. The value includes all headers for this single operation but excludes those of UDP, Enet Package Headers and TCP.
🗐	**BytesIn**	Gets count of all bytes coming in (including headers, excluding UDP/TCP overhead)
🗐	**BytesOut**	Gets count of all bytes going out (including headers, excluding UDP/TCP overhead)
🗐	**ChannelCount**	Gets / sets the number of channels available in UDP connections with Photon. Photon Channels are only supported for UDP. The default ChannelCount is 2. Channel IDs start with 0 and 255 is a internal channel.
🗐	**CommandBufferSize**	Initial size internal lists for incoming/outgoing commands (reliable and unreliable).
🗐	**CrcEnabled**	While not connected, this controls if the next connection(s) should use a per-package CRC checksum.
🗐	**DebugOut**	Sets the level (and amount) of debug output provided by the library.
🗐	**DisconnectTimeout**	Milliseconds after which a reliable UDP command triggers a timeout disconnect, unless acknowledged by server. This value currently only affects UDP connections. DisconnectTimeout is not an exact value for a timeout. The exact timing of the timeout depends on the frequency of **Service**() calls and commands that are sent with long roundtrip-times and variance are checked less often for re-sending!
		DisconnectTimeout and **SentCountAllowance** are competing settings: either might trigger a disconnect on the client first, depending on the values and Rountrip Time. Default: 10000 ms.
🗐	**HttpUrlParameters**	This string is added as simple GET parameters to the end of the server url + peerID combination. Include a starting "&" in your parameters as the peerID is always added to the **ServerAddress**. This value will be added to all following (http) operations, so make sure to clear it if needed. Can be applied even to "connect" (set before calling connect, clear / reset when connection is established). Set before making a Operation call and don't forget to clear or set it again before the next operation.

2

	IsEncryptionAvailable	This property is set internally, when OpExchangeKeysForEncryption successfully finished. While it's true, encryption can be used for operations.
	IsSendingOnlyAcks	
	IsSimulationEnabled	Gets or sets the network simulation "enabled" setting. Changing this value also locks this peer's sending and when setting false, the internally used queues are executed (so setting to false can take some cycles).
	LimitOfUnreliableCommands	Limits the queue of received unreliable commands within **DispatchIncomingCommands** before dispatching them. This works only in UDP. This limit is applied when you call **DispatchIncomingCommands**. If this client (already) received more than LimitOfUnreliableCommands, it will throw away the older ones instead of dispatching them. This can produce bigger gaps for unreliable commands but your client catches up faster.
	Listener	Gets the **IPhotonPeerListener** of this instance (set in constructor). Can be used in derived classes for Listener.DebugReturn().
	LocalMsTimestampDelegate	This setter for the (local-) timestamp delegate replaces the default Environment.TickCount with any equal function.
	LocalTimeInMilliSeconds	Gets a local timestamp in milliseconds by calling **SupportClass.GetTickCount**(). See **LocalMsTimestampDelegate**.
	MaximumTransferUnit	The Maximum Trasfer Unit (MTU) defines the (network-level) packet-content size that is guaranteed to arrive at the server in one piece. The Photon **Protocol** uses this size to split larger data into packets and for receive-buffers of packets.
	NetworkSimulationSettings	Gets the settings for built-in Network Simulation for this peer instance while **IsSimulationEnabled** will enable or disable them. Once obtained, the settings can be modified by changing the properties.
	OutgoingStreamBufferSize	Defines the initial size of an internally used MemoryStream for Tcp. The MemoryStream is used to aggregate operation into (less) send calls, which uses less resoures.
	PacketLossByCrc	Count of packages dropped due to failed CRC checks for this connection.
	PeerID	This peer's ID as assigned by the server or 0 if not using UDP. Will be 0xFFFF before the client connects.
	PeerState	This is the (low level) state of the connection to the server of a **PhotonPeer**. It is managed internally and read-only.
	QueuedIncomingCommands	Count of all currently received but not-yet-Dispatched reliable commands (**events** and operation results) from all channels.
	QueuedOutgoingCommands	Count of all commands currently queued as outgoing, including all channels and reliable, unreliable.
	ResentReliableCommands	Count of commands that got repeated (due to local repeat-timing before an ACK was received).
	RoundTripTime	Time until a reliable command is acknowledged by the server. The value measures network latency and for UDP it includes the server's ACK-delay (setting in config). In TCP, there is no ACK-delay, so the value is slightly lower (if you use default settings for Photon). RoundTripTime is updated constantly. Every reliable command will contribute a fraction to this value. This is also the approximate time until a raised event reaches another client or until an operation result is available.
	RoundTripTimeVariance	Changes of the roundtriptime as variance value. Gives a hint about how much the time is changing.

		SentCountAllowance	Number of send retries before a peer is considered lost/disconnected. Default: 5. The initial timeout countdown of a command is calculated by the current roundTripTime + 4 * roundTripTimeVariance. Please note that the timeout span until a command will be resent is not constant, but based on the roundtrip time at the initial sending, which will be doubled with every failed retry. **DisconnectTimeout** and SentCountAllowance are competing settings: either might trigger a disconnect on the client first, depending on the values and Rountrip Time.
		ServerAddress	The server address which was used in **PhotonPeer.Connect**() or null (before **Connect**() was called).
		ServerTimeInMilliSeconds	Approximated Environment.TickCount value of server (while connected).
		TimePingInterval	Sets the milliseconds without reliable command before a ping command (reliable) will be sent (Default: 1000ms). The ping command is used to keep track of the connection in case the client does not send reliable commands by itself. A ping (or reliable commands) will update the **RoundTripTime** calculation.
		TimestampOfLastSocketReceive	Stores timestamp of the last time anything (!) was received from the server (including low level Ping and ACKs but also **events** and operation-returns). This is not the time when something was dispatched. If you enable NetworkSimulation, this value is affected as well.
		TrafficStatsElapsedMs	Returns the count of milliseconds the stats are enabled for tracking.
		TrafficStatsEnabled	Enables the traffic statistics of a peer: **TrafficStatsIncoming**, **TrafficStatsOutgoing** and TrafficstatsGameLevel (nothing else). Default value: false (disabled).
		TrafficStatsGameLevel	Gets a statistic of incoming and outgoing traffic, split by operation, operation-result and event. **Operations** are outgoing traffic, results and **events** are incoming. Includes the per-command header sizes (Udp: Enet Command Header or Tcp: Message Header).
		TrafficStatsIncoming	Gets the byte-count of incoming "low level" messages, which are either Enet Commands or Tcp Messages. These include all headers, except those of the underlying internet protocol Udp or Tcp.
		TrafficStatsOutgoing	Gets the byte-count of outgoing "low level" messages, which are either Enet Commands or Tcp Messages. These include all headers, except those of the underlying internet protocol Udp or Tcp.
		UsedProtocol	The protocol this Peer uses to connect to Photon.
		WarningSize	The WarningSize is used test all message queues for congestion (in and out, reliable and unreliable). OnStatusChanged will be called with a warning if a queue holds WarningSize commands or a multiple of it. Default: 100.

2.1.6.1 LitePeer Constructor

2.1.6.1.1 LitePeer.LitePeer Constructor ()

Creates a LitePeer instance to connect and communicate with a Photon server.

Uses UDP as protocol (except in the Silverlight library).

C#

```
protected LitePeer();
```

2.1.6.1.2 LitePeer.LitePeer Constructor (ConnectionProtocol)

Creates a LitePeer instance to connect and communicate with a Photon server.

C#

```
protected LitePeer(ConnectionProtocol protocolType);
```

2.1.6.1.3 LitePeer.LitePeer Constructor (IPhotonPeerListener)

Creates a LitePeer instance to connect and communicate with a Photon server.

Uses UDP as protocol (except in the Silverlight library).

C#

```
public LitePeer(IPhotonPeerListener listener);
```

Parameters

Parameters	Description
IPhotonPeerListener listener	Your **IPhotonPeerListener** implementation.

2.1.6.1.4 LitePeer.LitePeer Constructor (IPhotonPeerListener, ConnectionProtocol)

Creates a LitePeer instance to communicate with Photon with your selection of protocol. We recommend UDP.

C#

```
public LitePeer(IPhotonPeerListener listener, ConnectionProtocol protocolType);
```

Parameters

Parameters	Description
IPhotonPeerListener listener	Your **IPhotonPeerListener** implementation.
ConnectionProtocol protocolType	**Protocol** to use to connect to Photon.

2.1.6.2 LitePeer Methods

2.1.6.2.1 LitePeer.OpChangeGroups Method

Operation to handle this client's interest groups (for **events** in room).

C#

```
public virtual bool OpChangeGroups(byte[] groupsToRemove, byte[] groupsToAdd);
```

Parameters

Parameters	Description
byte[] groupsToRemove	Groups to remove from interest. Null will not leave any. A byte[0] will remove all.
byte[] groupsToAdd	Groups to add to interest. Null will not add any. A byte[0] will add all current.

Remarks

Note the difference between passing null and byte[0]: null won't add/remove any groups. byte[0] will add/remove all (existing) groups. First, removing groups is executed. This way, you could leave all groups and join only the ones provided.

2.1.6.2.2 LitePeer.OpGetProperties Method

Gets all properties of the game and each actor.

C#

```
public virtual bool OpGetProperties(byte channelId);
```

Parameters

Parameters	Description
byte channelId	Number of channel to use (starting with 0).

Returns

If operation could be enqueued for sending

Remarks

Please read the general description of **Properties on Photon**.

2.1.6.2.3 OpGetPropertiesOfActor Method

2.1.6.2.3.1 LitePeer.OpGetPropertiesOfActor Method (int[], byte[], byte)

Gets selected properties of some actors.

C#

```
public virtual bool OpGetPropertiesOfActor(int[] actorNrList, byte[] properties, byte
channelId);
```

Parameters

Parameters	Description
int[] actorNrList	optional, a list of actornumbers to get the properties of
byte[] properties	array of property keys to fetch. optional (can be null).
byte channelId	Number of channel to use (starting with 0).

Returns

If operation could be enqueued for sending

Remarks

Please read the general description of **Properties on Photon**.

2.1.6.2.3.2 LitePeer.OpGetPropertiesOfActor Method (int[], string[], byte)

Gets selected properties of an actor.

C#

```
public virtual bool OpGetPropertiesOfActor(int[] actorNrList, string[] properties, byte
channelId);
```

Parameters

Parameters	Description
int[] actorNrList	optional, a list of actornumbers to get the properties of
string[] properties	optional, array of property keys to fetch
byte channelId	Number of channel to use (starting with 0).

Returns

If operation could be enqueued for sending

Remarks

Please read the general description of **Properties on Photon**.

2

2.1.6.2.4 OpGetPropertiesOfGame Method

2.1.6.2.4.1 LitePeer.OpGetPropertiesOfGame Method (byte[], byte)

Gets selected properties of current game.

C#

```csharp
public virtual bool OpGetPropertiesOfGame(byte[] properties, byte channelId);
```

Parameters

Parameters	Description
byte[] properties	array of property keys to fetch. optional (can be null).
byte channelId	Number of channel to use (starting with 0).

Returns

If operation could be enqueued for sending

Remarks

Please read the general description of **Properties on Photon**.

2.1.6.2.4.2 LitePeer.OpGetPropertiesOfGame Method (string[], byte)

Gets selected properties of current game.

C#

```csharp
public virtual bool OpGetPropertiesOfGame(string[] properties, byte channelId);
```

Parameters

Parameters	Description
string[] properties	array of property keys to fetch. optional (can be null).
byte channelId	Number of channel to use (starting with 0).

Returns

If operation could be enqueued for sending

Remarks

Please read the general description of **Properties on Photon**.

2.1.6.2.5 OpJoin Method

2.1.6.2.5.1 LitePeer.OpJoin Method (string)

This operation will join an existing room by name or create one if the name is not in use yet.

Rooms (or games) are simply identified by name. We assume that users always want to get into a room - no matter if it existed before or not, so it might be a new one. If you want to make sure a room is created (new, empty), the client side might come up with a unique name for it (make sure the name was not taken yet).

The application "Lite Lobby" lists room names and effectively allows the user to select a distinct one.

Each actor (a.k.a. player) in a room will get **events** that are raised for the room by any player.

To distinguish the actors, each gets a consecutive actornumber. This is used in **events** to mark who triggered the event. A client finds out it's own actornumber in the return callback for operation Join. Number 1 is the lowest actornumber in each room and the client with that actornumber created the room.

Each client could easily send custom data around. If the data should be available to newcomers, it makes sense to use

Properties.

Joining a room will trigger the event **LiteEventCode.Join**, which contains the list of actorNumbers of current players inside the room (**LiteEventKey.ActorList**). This also gives you a count of current players.

C#

```csharp
public virtual bool OpJoin(string gameName);
```

Parameters

Parameters	Description
string gameName	Any identifying name for a room / game.

Returns

If operation could be enqueued for sending

2.1.6.2.5.2 LitePeer.OpJoin Method (string, Hashtable, Hashtable, bool)

This operation will join an existing room by name or create one if the name is not in use yet.

Rooms (or games) are simply identified by name. We assume that users always want to get into a room - no matter if it existed before or not, so it might be a new one. If you want to make sure a room is created (new, empty), the client side might come up with a unique name for it (make sure the name was not taken yet).

The application "Lite Lobby" lists room names and effectively allows the user to select a distinct one.

Each actor (a.k.a. player) in a room will get **events** that are raised for the room by any player.

To distinguish the actors, each gets a consecutive actornumber. This is used in **events** to mark who triggered the event. A client finds out it's own actornumber in the return callback for operation Join. Number 1 is the lowest actornumber in each room and the client with that actornumber created the room.

Each client could easily send custom data around. If the data should be available to newcomers, it makes sense to use Properties.

Joining a room will trigger the event **LiteEventCode.Join**, which contains the list of actorNumbers of current players inside the room (**LiteEventKey.ActorList**). This also gives you a count of current players.

C#

```csharp
public virtual bool OpJoin(string gameName, Hashtable gameProperties, Hashtable
actorProperties, bool broadcastActorProperties);
```

Parameters

Parameters	Description
string gameName	Any identifying name for a room / game.
Hashtable gameProperties	optional, set of game properties, by convention: only used if game is new/created
Hashtable actorProperties	optional, set of actor properties
bool broadcastActorProperties	optional, broadcast actor proprties in join-event

Returns

If operation could be enqueued for sending

2.1.6.2.6 LitePeer.OpLeave Method

Leave operation of the Lite Application (also in Lite Lobby). Leaves a room / game, but keeps the connection. This operations triggers the event **LiteEventCode.Leave** for the remaining clients. The event includes the actorNumber of the player who left in key **LiteEventKey.ActorNr**.

C#

```csharp
public virtual bool OpLeave();
```

Returns

Consecutive invocationID of the OP. Will throw Exception if not connected.

2.1.6.2.7 OpRaiseEvent Method

2.1.6.2.7.1 LitePeer.OpRaiseEvent Method (byte, Hashtable, bool)

RaiseEvent tells the server to send an event to the other players within the same room.

C#

```csharp
public virtual bool OpRaiseEvent(byte eventCode, Hashtable customEventContent, bool
sendReliable);
```

Parameters

Parameters	Description
byte eventCode	Identifies this type of event (and the content). Your game's event codes can start with 0.
Hashtable customEventContent	Custom data you want to send along (use null, if none).
bool sendReliable	If this event has to arrive reliably (potentially repeated if it's lost).

Returns

If operation could be enqueued for sending

Remarks

This method is described in one of its overloads.

2.1.6.2.7.2 LitePeer.OpRaiseEvent Method (byte, Hashtable, bool, byte)

RaiseEvent tells the server to send an event to the other players within the same room.

C#

```csharp
public virtual bool OpRaiseEvent(byte eventCode, Hashtable customEventContent, bool
sendReliable, byte channelId);
```

Parameters

Parameters	Description
byte eventCode	Identifies this type of event (and the content). Your game's event codes can start with 0.
Hashtable customEventContent	Custom data you want to send along (use null, if none).
bool sendReliable	If this event has to arrive reliably (potentially repeated if it's lost).
byte channelId	Number of channel (sequence) to use (starting with 0).

Returns

If operation could be enqueued for sending.

Remarks

Type and content of the event can be defined by the client side at will. The server only forwards the content and eventCode to others in the same room.

The eventCode should be used to define the event's type and content respectively./// Lite and Loadbalancing are using a few eventCode values already but those start with 255 and go down. Your eventCodes can start at 1, going up.

The customEventContent is a Hashtable with any number of key-value pairs of **serializable datatypes** or null. Receiving clients can access this Hashtable as Parameter **LiteEventKey.Data** (see below).

RaiseEvent can be used reliable or unreliable. Both result in ordered **events** but the unreliable ones might be lost and allow gaps in the resulting event sequence. On the other hand, they cause less overhead and are optimal for data that is replaced soon.

Like all operations, RaiseEvent is not done immediately but when you call **SendOutgoingCommands**.

It is recommended to keep keys (and data) as simple as possible (e.g. byte or short as key), as the data is typically sent multiple times per second. This easily adds up to a huge amount of data otherwise.

Example

```csharp
//send some position data (using byte-keys, as they are small):

Hashtable evInfo = new Hashtable();
Player local = (Player)players[playerLocalID];
evInfo.Add((byte)STATUS_PLAYER_POS_X, (int)local.posX);
evInfo.Add((byte)STATUS_PLAYER_POS_Y, (int)local.posY);

peer.OpRaiseEvent(EV_MOVE, evInfo, true);  //EV_MOVE = (byte)1

//receive this custom event in OnEvent():
Hashtable data = (Hashtable)photonEvent[LiteEventKey.Data];
switch (eventCode) {
  case EV_MOVE:                //1 in this sample
      p = (Player)players[actorNr];
      if (p != null) {
          p.posX = (int)data[(byte)STATUS_PLAYER_POS_X];
          p.posY = (int)data[(byte)STATUS_PLAYER_POS_Y];
      }
      break;
```

Events from the Photon Server are internally buffered until they are **Dispatched**, just like OperationResults.

2.1.6.2.7.3 LitePeer.OpRaiseEvent Method (byte, Hashtable, bool, byte, EventCaching, ReceiverGroup)

Calls operation RaiseEvent on the server, with full control of event-caching and the target receivers.

C#

```csharp
public virtual bool OpRaiseEvent(byte eventCode, Hashtable customEventContent, bool
sendReliable, byte channelId, EventCaching cache, ReceiverGroup receivers);
```

Parameters

Parameters	Description
byte eventCode	Identifies this type of event (and the content). Your game's event codes can start with 0.
Hashtable customEventContent	Custom data you want to send along (use null, if none).
bool sendReliable	If this event has to arrive reliably (potentially repeated if it's lost).
byte channelId	Number of channel to use (starting with 0).
EventCaching cache	**Events** can be cached (merged and removed) for players joining later on.
ReceiverGroup receivers	Controls who should get this event.

Returns

If operation could be enqueued for sending.

Remarks

This method is described in one of its overloads.

The cache parameter defines if and how this event will be cached server-side. Per event-code, your client can store **events** and update them and will send cached **events** to players joining the same room.

The option EventCaching.DoNotCache matches the default behaviour of RaiseEvent. The option

EventCaching.MergeCache will merge the costomEventContent into existing one. Values in the customEventContent Hashtable can be null to remove existing values.

With the receivers parameter, you can chose who gets this event: Others (default), All (includes you as sender) or MasterClient. The MasterClient is the connected player with the lowest ActorNumber in this room. This player could get some privileges, if needed.

Read more about Cached **Events** in the DevNet: http://doc.exitgames.com

2.1.6.2.7.4 **LitePeer.OpRaiseEvent Method (byte, Hashtable, bool, byte, int[])**

RaiseEvent tells the server to send an event to the other players within the same room.

C#

```
public virtual bool OpRaiseEvent(byte eventCode, Hashtable customEventContent, bool
sendReliable, byte channelId, int[] targetActors);
```

Parameters

Parameters	Description
byte eventCode	Identifies this type of event (and the content). Your game's event codes can start with 0.
Hashtable customEventContent	Custom data you want to send along (use null, if none).
bool sendReliable	If this event has to arrive reliably (potentially repeated if it's lost).
byte channelId	Number of channel to use (starting with 0).
int[] targetActors	List of actorNumbers that receive this event.

Returns

If operation could be enqueued for sending.

Remarks

This method is described in one of its overloads.

This variant has an optional list of targetActors. Use this to send the event only to specific actors in the same room, each identified by an actorNumber (or ID).

This can be useful to implement private messages inside a room or similar.

2.1.6.2.7.5 **LitePeer.OpRaiseEvent Method (byte, bool, object)**

Send an event with custom code/type and any content to the other players in the same room.

C#

```
public virtual bool OpRaiseEvent(byte eventCode, bool sendReliable, object
customEventContent);
```

Parameters

Parameters	Description
byte eventCode	Identifies this type of event (and the content). Your game's event codes can start with 0.
bool sendReliable	If this event has to arrive reliably (potentially repeated if it's lost).
object customEventContent	Any serializable datatype (including Hashtable like the other OpRaiseEvent overloads).

Returns

If operation could be enqueued for sending. Sent when calling: **Service** or **SendOutgoingCommands**.

Remarks

This override explicitly uses another parameter order to not mix it up with the implementation for Hashtable only.

2.1.6.2.7.6 **LitePeer.OpRaiseEvent Method (byte, bool, object, byte, EventCaching, int[], ReceiverGroup, byte)**

Send an event with custom code/type and any content to the other players in the same room.

C#

```csharp
public virtual bool OpRaiseEvent(byte eventCode, bool sendReliable, object
customEventContent, byte channelId, EventCaching cache, int[] targetActors, ReceiverGroup
receivers, byte interestGroup);
```

Parameters

Parameters	Description
byte eventCode	Identifies this type of event (and the content). Your game's event codes can start with 0.
bool sendReliable	If this event has to arrive reliably (potentially repeated if it's lost).
object customEventContent	Any serializable datatype (including Hashtable like the other OpRaiseEvent overloads).
byte channelId	Command sequence in which this command belongs. Must be less than value of **ChannelCount** property. Default: 0.
EventCaching cache	Affects how the server will treat the event caching-wise. Can cache **events** for players joining later on or remove previously cached **events**. Default: DoNotCache.
int[] targetActors	List of ActorNumbers (in this room) to send the event to. Overrides caching. Default: null.
ReceiverGroup receivers	Defines a target-player group. Default: Others.
byte interestGroup	Defines to which interest group the event is sent. Players can subscribe or unsibscribe to groups. Group 0 is always sent to all. Default: 0.

Returns

If operation could be enqueued for sending. Sent when calling: **Service** or **SendOutgoingCommands**.

Remarks

This override explicitly uses another parameter order to not mix it up with the implementation for Hashtable only.

2.1.6.2.7.7 **LitePeer.OpRaiseEvent Method (byte, byte, Hashtable, bool)**

Send your custom data as event to an "interest group" in the current Room.

C#

```csharp
public virtual bool OpRaiseEvent(byte eventCode, byte interestGroup, Hashtable
customEventContent, bool sendReliable);
```

Parameters

Parameters	Description
byte eventCode	Identifies this type of event (and the content). Your game's event codes can start with 0.
byte interestGroup	The ID of the interest group this event goes to (exclusively). Grouo 0 sends to all.
Hashtable customEventContent	Custom data you want to send along (use null, if none).
bool sendReliable	If this event has to arrive reliably (potentially repeated if it's lost).

Returns

If operation could be enqueued for sending

Remarks

No matter if reliable or not, when an event is sent to a interest Group, some users won't get this data. Clients can control the groups they are interested in by using **OpChangeGroups**.

2.1.6.2.8 LitePeer.OpSetPropertiesOfActor Method

Attaches or updates properties of the specified actor.

C#

```
public virtual bool OpSetPropertiesOfActor(int actorNr, Hashtable properties, bool
broadcast, byte channelId);
```

Parameters

Parameters	Description
int actorNr	the actorNr is used to identify a player/peer in a game
Hashtable properties	Hashtable containing the properties to add or update.
bool broadcast	true will trigger an event LiteEventKey.PropertiesChanged with the updated properties in it
byte channelId	Number of channel to use (starting with 0).

Returns

If operation could be enqueued for sending

Remarks

Please read the general description of **Properties on Photon**.

2.1.6.2.9 LitePeer.OpSetPropertiesOfGame Method

Attaches or updates properties of the current game.

C#

```
public virtual bool OpSetPropertiesOfGame(Hashtable properties, bool broadcast, byte
channelId);
```

Parameters

Parameters	Description
Hashtable properties	hashtable containing the properties to add or overwrite
bool broadcast	true will trigger an event LiteEventKey.PropertiesChanged with the updated properties in it
byte channelId	Number of channel to use (starting with 0).

Returns

If operation could be enqueued for sending

Remarks

Please read the general description of **Properties on Photon**.

2.1.7 NetworkSimulationSet Class

A set of network simulation settings, enabled (and disabled) by **PhotonPeer.IsSimulationEnabled**.

C#

```
public class NetworkSimulationSet;
```

Remarks

For performance reasons, the lag and jitter settings can't be produced exactly. In some cases, the resulting lag will be up to 20ms bigger than the lag settings. Even if all settings are 0, simulation will be used. Set **PhotonPeer.IsSimulationEnabled** to false to disable it if no longer needed.

All lag, jitter and loss is additional to the current, real network conditions. If the network is slow in reality, this will add even more lag. The jitter values will affect the lag positive and negative, so the lag settings describe the medium lag even with jitter. The jitter influence is: [-jitter..+jitter]. Packets "lost" due to **OutgoingLossPercentage** count for BytesOut and **LostPackagesOut**. Packets "lost" due to **IncomingLossPercentage** count for BytesIn and **LostPackagesIn**.

NetworkSimulationSet Fields

	Name	Description
	NetSimManualResetEvent	This is NetSimManualResetEvent, a member of class NetworkSimulationSet.

NetworkSimulationSet Methods

	Name	Description
	ToString	This is ToString, a member of class NetworkSimulationSet.

NetworkSimulationSet Properties

	Name	Description
	IncomingJitter	Randomizes **IncomingLag** by [-IncomingJitter..+IncomingJitter]. Default: 0.
	IncomingLag	Incoming packages delay in ms. Default: 100.
	IncomingLossPercentage	Percentage of incoming packets that should be lost. Between 0..100. Default: 1. TCP ignores this setting.
	LostPackagesIn	Counts how many incoming packages actually got lost. TCP connections ignore loss and this stays 0.
	LostPackagesOut	Counts how many outgoing packages actually got lost. TCP connections ignore loss and this stays 0.
	OutgoingJitter	Randomizes **OutgoingLag** by [-OutgoingJitter..+OutgoingJitter]. Default: 0.
	OutgoingLag	Outgoing packages delay in ms. Default: 100.
	OutgoingLossPercentage	Percentage of outgoing packets that should be lost. Between 0..100. Default: 1. TCP ignores this setting.

2.1.7.1 NetworkSimulationSet Fields

2.1.7.1.1 NetworkSimulationSet.NetSimManualResetEvent Field

C#

```
public readonly ManualResetEvent NetSimManualResetEvent = new ManualResetEvent(false);
```

Description

This is NetSimManualResetEvent, a member of class NetworkSimulationSet.

2.1.7.2 NetworkSimulationSet Methods

2.1.7.2.1 NetworkSimulationSet.ToString Method

C#

```
public override string ToString();
```

Description

This is ToString, a member of class NetworkSimulationSet.

2.1.7.3 NetworkSimulationSet Properties

2.1.7.3.1 NetworkSimulationSet.IncomingJitter Property

Randomizes **IncomingLag** by [-IncomingJitter..+IncomingJitter]. Default: 0.

C#

```
public int IncomingJitter;
```

2.1.7.3.2 NetworkSimulationSet.IncomingLag Property

Incoming packages delay in ms. Default: 100.

C#

```
public int IncomingLag;
```

2.1.7.3.3 NetworkSimulationSet.IncomingLossPercentage Property

Percentage of incoming packets that should be lost. Between 0..100. Default: 1. TCP ignores this setting.

C#

```
public int IncomingLossPercentage;
```

2.1.7.3.4 NetworkSimulationSet.LostPackagesIn Property

Counts how many incoming packages actually got lost. TCP connections ignore loss and this stays 0.

C#

```
public int LostPackagesIn;
```

2.1.7.3.5 NetworkSimulationSet.LostPackagesOut Property

Counts how many outgoing packages actually got lost. TCP connections ignore loss and this stays 0.

C#

```
public int LostPackagesOut;
```

2.1.7.3.6 NetworkSimulationSet.OutgoingJitter Property

Randomizes **OutgoingLag** by [-OutgoingJitter..+OutgoingJitter]. Default: 0.

C#

```
public int OutgoingJitter;
```

2.1.7.3.7 NetworkSimulationSet.OutgoingLag Property

Outgoing packages delay in ms. Default: 100.

C#

```csharp
public int OutgoingLag;
```

2.1.7.3.8 NetworkSimulationSet.OutgoingLossPercentage Property

Percentage of outgoing packets that should be lost. Between 0..100. Default: 1. TCP ignores this setting.

C#

```csharp
public int OutgoingLossPercentage;
```

2.1.8 OperationRequest Class

Container for an Operation request, which is a code and parameters.

C#

```csharp
public class OperationRequest;
```

Remarks

On the lowest level, Photon only allows byte-typed keys for operation parameters. The values of each such parameter can be any serializable datatype: byte, int, hashtable and many more.

OperationRequest Fields

	Name	Description
●	OperationCode	Byte-typed code for an operation - the short identifier for the server's method to call.
●	Parameters	The parameters of the operation - each identified by a byte-typed code in Photon.

2.1.8.1 OperationRequest Fields

2.1.8.1.1 OperationRequest.OperationCode Field

Byte-typed code for an operation - the short identifier for the server's method to call.

C#

```csharp
public byte OperationCode;
```

2.1.8.1.2 OperationRequest.Parameters Field

The parameters of the operation - each identified by a byte-typed code in Photon.

C#

```csharp
public Dictionary<byte, object> Parameters;
```

2.1.9 OperationResponse Class

Contains the server's response for an operation called by this peer. The indexer of this class actually provides access to the **Parameters** Dictionary.

C#

```csharp
public class OperationResponse;
```

Remarks

The **OperationCode** defines the type of operation called on Photon and in turn also the **Parameters** that are set in the request. Those are provided as Dictionary with byte-keys. There are pre-defined constants for various codes defined in the Lite application. Check: **LiteOpCode**, **LiteOpKey**, etc.

An operation's request is summarized by the **ReturnCode**: a short typed code for "Ok" or some different result. The code's meaning is specific per operation. An optional **DebugMessage** can be provided to simplify debugging.

Each call of an operation gets an ID, called the "invocID". This can be matched to the IDs returned with any operation calls. This way, an application could track if a certain OpRaiseEvent call was successful.

OperationResponse Fields

	Name	Description
◆	**DebugMessage**	An optional string sent by the server to provide readable feedback in error-cases. Might be null.
◆	**OperationCode**	The code for the operation called initially (by this peer).
◆	**Parameters**	A Dictionary of values returned by an operation, using byte-typed keys per value.
◆	**ReturnCode**	A code that "summarizes" the operation's success or failure. Specific per operation. 0 usually means "ok".

OperationResponse Methods

	Name	Description
⇒◆	**this**	Alternative access to the **Parameters**, which wraps up a TryGetValue() call on the **Parameters** Dictionary.
⇒◆	**ToString**	ToString() override.
⇒◆	**ToStringFull**	Extensive output of operation results.

2.1.9.1 OperationResponse Fields

2.1.9.1.1 OperationResponse.DebugMessage Field

An optional string sent by the server to provide readable feedback in error-cases. Might be null.

C#

```
public string DebugMessage;
```

2.1.9.1.2 OperationResponse.OperationCode Field

The code for the operation called initially (by this peer).

C#

```
public byte OperationCode;
```

Remarks

Use enums or constants to be able to handle those codes, like **LiteOpCode** does.

2.1.9.1.3 OperationResponse.Parameters Field

A Dictionary of values returned by an operation, using byte-typed keys per value.

C#

```
public Dictionary<byte, object> Parameters;
```

2.1.9.1.4 OperationResponse.ReturnCode Field

A code that "summarizes" the operation's success or failure. Specific per operation. 0 usually means "ok".

C#

```
public short ReturnCode;
```

2.1.9.2 OperationResponse Methods

2.1.9.2.1 OperationResponse.this Indexer

Alternative access to the **Parameters**, which wraps up a TryGetValue() call on the **Parameters** Dictionary.

C#

```
public object this[byte parameterCode];
```

Parameters

Parameters	Description
byte parameterCode	The byte-code of a returned value.

Returns

The value returned by the server, or null if the key does not exist in **Parameters**.

2.1.9.2.2 OperationResponse.ToString Method

ToString() override.

C#

```
public override string ToString();
```

Returns

Relatively short output of OpCode and returnCode.

2.1.9.2.3 OperationResponse.ToStringFull Method

Extensive output of operation results.

C#

```
public string ToStringFull();
```

Returns

To be used in debug situations only, as it returns a string for each value.

2.1.10 PhotonPeer Class

Instances of the PhotonPeer class are used to connect to a Photon server and communicate with it.

C#

```
public class PhotonPeer;
```

Remarks

A PhotonPeer instance allows communication with the Photon Server, which in turn distributes messages to other

2

PhotonPeer clients.

An application can use more than one PhotonPeer instance, which are treated as separate users on the server. Each should have its own listener instance, to separate the operations, callbacks and **events**.

Methods

	Name	Description
⇒◆	**PhotonPeer**	Creates a new PhotonPeer instance to communicate with Photon. Connection is UDP based, except for Silverlight.

PhotonPeer Methods

	Name	Description
⇒◆𝕍	**Connect**	This method does a DNS lookup (if necessary) and connects to the given serverAddress. The return value gives you feedback if the address has the correct format. If so, this starts the process to establish the connection itself, which might take a few seconds. When the connection is established, a callback to **IPhotonPeerListener.OnStatusChanged** will be done. If the connection can't be established, despite having a valid address, the OnStatusChanged is called with an error-value. The applicationName defines the application logic to use server-side and it should match the name of one of the apps in your server's config. By default,... **more**
⇒◆𝕍	**Disconnect**	This method initiates a mutual disconnect between this client and the server.
⇒◆𝕍	**DispatchIncomingCommands**	This method directly causes the callbacks for **events**, responses and state changes within a **IPhotonPeerListener**. DispatchIncomingCommands only executes a single received command per call. If a command was dispatched, the return value is true and the method should be called again. This method is called by **Service**() until currently available commands are dispatched.
⇒◆	**EstablishEncryption**	This method creates a public key for this client and exchanges it with the server.
⇒◆𝕍	**FetchServerTimestamp**	This will fetch the server's timestamp and update the approximation for property ServerTimeInMilliseconds. The server time approximation will NOT become more accurate by repeated calls. Accuracy currently depends on a single roundtrip which is done as fast as possible. The command used for this is immediately acknowledged by the server. This makes sure the roundtrip time is low and the timestamp + rountriptime / 2 is close to the original value.
⇒◆𝕍	**OpCustom**	Allows the client to send any operation to the Photon Server by setting any opCode and the operation's parameters.
⇒◆𝕊	**RegisterType**	Registers new types/classes for de/serialization and the fitting methods to call for this type.
⇒◆𝕍	**SendAcksOnly**	This is SendAcksOnly, a member of class PhotonPeer.
⇒◆𝕍	**SendOutgoingCommands**	This method creates a UDP/TCP package for outgoing commands (operations and acknowledgements) and sends them to the server. This method is also called by **Service**().
⇒◆𝕍	**Service**	This method excutes **DispatchIncomingCommands** and **SendOutgoingCommands** in your application Thread-context.
⇒◆𝕍	**StopThread**	This method immediately closes a connection (pure client side) and ends related listening Threads.
⇒◆	**TrafficStatsReset**	Creates new instances of **TrafficStats** and starts a new timer for those.
⇒◆	**VitalStatsToString**	Returns a string of the most interesting connection statistics. When you have issues on the client side, these might contain hints about the issue's cause.

2

PhotonPeer Properties

	Name	Description
	ByteCountCurrentDispatch	Gets the size of the dispatched event or operation-result in bytes. This value is set before OnEvent() or OnOperationResponse() is called (within **DispatchIncomingCommands**()).
	ByteCountLastOperation	Gets the size of the last serialized operation call in bytes. The value includes all headers for this single operation but excludes those of UDP, Enet Package Headers and TCP.
	BytesIn	Gets count of all bytes coming in (including headers, excluding UDP/TCP overhead)
	BytesOut	Gets count of all bytes going out (including headers, excluding UDP/TCP overhead)
	ChannelCount	Gets / sets the number of channels available in UDP connections with Photon. Photon Channels are only supported for UDP. The default ChannelCount is 2. Channel IDs start with 0 and 255 is a internal channel.
	CommandBufferSize	Initial size internal lists for incoming/outgoing commands (reliable and unreliable).
	CrcEnabled	While not connected, this controls if the next connection(s) should use a per-package CRC checksum.
	DebugOut	Sets the level (and amount) of debug output provided by the library.
	DisconnectTimeout	Milliseconds after which a reliable UDP command triggers a timeout disconnect, unless acknowledged by server. This value currently only affects UDP connections. DisconnectTimeout is not an exact value for a timeout. The exact timing of the timeout depends on the frequency of **Service**() calls and commands that are sent with long roundtrip-times and variance are checked less often for re-sending! DisconnectTimeout and **SentCountAllowance** are competing settings: either might trigger a disconnect on the client first, depending on the values and Rountrip Time. Default: 10000 ms.
	HttpUrlParameters	This string is added as simple GET parameters to the end of the server url + peerID combination. Include a starting "&" in your parameters as the peerID is always added to the **ServerAddress**. This value will be added to all following (http) operations, so make sure to clear it if needed. Can be applied even to "connect" (set before calling connect, clear / reset when connection is established). Set before making a Operation call and don't forget to clear or set it again before the next operation.
	IsEncryptionAvailable	This property is set internally, when OpExchangeKeysForEncryption successfully finished. While it's true, encryption can be used for operations.
	IsSendingOnlyAcks	
	IsSimulationEnabled	Gets or sets the network simulation "enabled" setting. Changing this value also locks this peer's sending and when setting false, the internally used queues are executed (so setting to false can take some cycles).
	LimitOfUnreliableCommands	Limits the queue of received unreliable commands within **DispatchIncomingCommands** before dispatching them. This works only in UDP. This limit is applied when you call **DispatchIncomingCommands**. If this client (already) received more than LimitOfUnreliableCommands, it will throw away the older ones instead of dispatching them. This can produce bigger gaps for unreliable commands but your client catches up faster.
	Listener	Gets the **IPhotonPeerListener** of this instance (set in constructor). Can be used in derived classes for Listener.DebugReturn().
	LocalMsTimestampDelegate	This setter for the (local-) timestamp delegate replaces the default Environment.TickCount with any equal function.
	LocalTimeInMilliSeconds	Gets a local timestamp in milliseconds by calling **SupportClass.GetTickCount**(). See **LocalMsTimestampDelegate**.

	MaximumTransferUnit	The Maximum Trasfer Unit (MTU) defines the (network-level) packet-content size that is guaranteed to arrive at the server in one piece. The Photon **Protocol** uses this size to split larger data into packets and for receive-buffers of packets.
	NetworkSimulationSettings	Gets the settings for built-in Network Simulation for this peer instance while **IsSimulationEnabled** will enable or disable them. Once obtained, the settings can be modified by changing the properties.
	OutgoingStreamBufferSize	Defines the initial size of an internally used MemoryStream for Tcp. The MemoryStream is used to aggregate operation into (less) send calls, which uses less resoures.
	PacketLossByCrc	Count of packages dropped due to failed CRC checks for this connection.
	PeerID	This peer's ID as assigned by the server or 0 if not using UDP. Will be 0xFFFF before the client connects.
	PeerState	This is the (low level) state of the connection to the server of a PhotonPeer. It is managed internally and read-only.
	QueuedIncomingCommands	Count of all currently received but not-yet-Dispatched reliable commands (**events** and operation results) from all channels.
	QueuedOutgoingCommands	Count of all commands currently queued as outgoing, including all channels and reliable, unreliable.
	ResentReliableCommands	Count of commands that got repeated (due to local repeat-timing before an ACK was received).
	RoundTripTime	Time until a reliable command is acknowledged by the server. The value measures network latency and for UDP it includes the server's ACK-delay (setting in config). In TCP, there is no ACK-delay, so the value is slightly lower (if you use default settings for Photon). RoundTripTime is updated constantly. Every reliable command will contribute a fraction to this value. This is also the approximate time until a raised event reaches another client or until an operation result is available.
	RoundTripTimeVariance	Changes of the roundtriptime as variance value. Gives a hint about how much the time is changing.
	SentCountAllowance	Number of send retries before a peer is considered lost/disconnected. Default: 5. The initial timeout countdown of a command is calculated by the current roundTripTime + 4 * roundTripTimeVariance. Please note that the timeout span until a command will be resent is not constant, but based on the roundtrip time at the initial sending, which will be doubled with every failed retry. **DisconnectTimeout** and SentCountAllowance are competing settings: either might trigger a disconnect on the client first, depending on the values and Rountrip Time.
	ServerAddress	The server address which was used in **PhotonPeer.Connect**() or null (before **Connect**() was called).
	ServerTimeInMilliSeconds	Approximated Environment.TickCount value of server (while connected).
	TimePingInterval	Sets the milliseconds without reliable command before a ping command (reliable) will be sent (Default: 1000ms). The ping command is used to keep track of the connection in case the client does not send reliable commands by itself. A ping (or reliable commands) will update the **RoundTripTime** calculation.
	TimestampOfLastSocketReceive	Stores timestamp of the last time anything (!) was received from the server (including low level Ping and ACKs but also **events** and operation-returns). This is not the time when something was dispatched. If you enable NetworkSimulation, this value is affected as well.
	TrafficStatsElapsedMs	Returns the count of milliseconds the stats are enabled for tracking.
	TrafficStatsEnabled	Enables the traffic statistics of a peer: **TrafficStatsIncoming**, **TrafficStatsOutgoing** and TrafficstatsGameLevel (nothing else). Default value: false (disabled).

2

		TrafficStatsGameLevel	Gets a statistic of incoming and outgoing traffic, split by operation, operation-result and event. **Operations** are outgoing traffic, results and **events** are incoming. Includes the per-command header sizes (Udp: Enet Command Header or Tcp: Message Header).
		TrafficStatsIncoming	Gets the byte-count of incoming "low level" messages, which are either Enet Commands or Tcp Messages. These include all headers, except those of the underlying internet protocol Udp or Tcp.
		TrafficStatsOutgoing	Gets the byte-count of outgoing "low level" messages, which are either Enet Commands or Tcp Messages. These include all headers, except those of the underlying internet protocol Udp or Tcp.
		UsedProtocol	The protocol this Peer uses to connect to Photon.
		WarningSize	The WarningSize is used test all message queues for congestion (in and out, reliable and unreliable). OnStatusChanged will be called with a warning if a queue holds WarningSize commands or a multiple of it. Default: 100.

2.1.10.1 PhotonPeer Constructor

2.1.10.1.1 PhotonPeer.PhotonPeer Constructor (IPhotonPeerListener)

Creates a new PhotonPeer instance to communicate with Photon.

Connection is UDP based, except for Silverlight.

C#

```
[Obsolete("Use the constructor with ConnectionProtocol instead.")]
public PhotonPeer(IPhotonPeerListener listener);
```

Parameters

Parameters	Description
IPhotonPeerListener listener	a **IPhotonPeerListener** implementation

2.1.10.1.2 PhotonPeer.PhotonPeer Constructor (IPhotonPeerListener, ConnectionProtocol)

Creates a new PhotonPeer instance to communicate with Photon and selects either UDP or TCP as protocol. We recommend UDP.

C#

```
public PhotonPeer(IPhotonPeerListener listener, ConnectionProtocol protocolType);
```

Parameters

Parameters	Description
IPhotonPeerListener listener	a **IPhotonPeerListener** implementation
ConnectionProtocol protocolType	**Protocol** to use to connect to Photon.

2.1.10.1.3 PhotonPeer.PhotonPeer Constructor (IPhotonPeerListener, bool)

Deprecated. Please use: PhotonPeer(**IPhotonPeerListener** listener, **ConnectionProtocol** protocolType).

C#

```
[Obsolete("Use the constructor with ConnectionProtocol instead.")]
public PhotonPeer(IPhotonPeerListener listener, bool useTcp);
```

2.1.10.2 PhotonPeer Methods

2.1.10.2.1 PhotonPeer.Connect Method

This method does a DNS lookup (if necessary) and connects to the given serverAddress.

The return value gives you feedback if the address has the correct format. If so, this starts the process to establish the connection itself, which might take a few seconds.

When the connection is established, a callback to **IPhotonPeerListener.OnStatusChanged** will be done. If the connection can't be established, despite having a valid address, the OnStatusChanged is called with an error-value.

The applicationName defines the application logic to use server-side and it should match the name of one of the apps in your server's config.

By default, the applicationName is "Lite" but other samples use "LiteLobby" and "MmoDemo" in Connect(). You can setup your own application and name it any way you like.

C#

```
public virtual bool Connect(String serverAddress, String applicationName);
```

Parameters

Parameters	Description
String serverAddress	Address of the Photon server. Format: ip:port (e.g. 127.0.0.1:5055) or hostname:port (e.g. localhost:5055)
String applicationName	The name of the application to use within Photon or the appId of PhotonCloud. Should match a "Name" for an application, as setup in your PhotonServer.config.

Returns

true if IP is available (DNS name is resolved) and server is being connected. false on error.

2.1.10.2.2 PhotonPeer.Disconnect Method

This method initiates a mutual disconnect between this client and the server.

C#

```
public virtual void Disconnect();
```

Remarks

Calling this method does not immediately close a connection. Disconnect lets the server know that this client is no longer listening. For the server, this is a much faster way to detect that the client is gone but it requires the client to send a few final messages.

On completition, OnStatusChanged is called with the StatusCode.Disconnect.

If the client is disconnected already or the connection thread is stopped, then there is no callback.

Lite: The default server logic will leave any joined game and trigger the respective event (**LiteEventCode.Leave**) for the remaining players.

2.1.10.2.3 PhotonPeer.DispatchIncomingCommands Method

This method directly causes the callbacks for **events**, responses and state changes within a **IPhotonPeerListener**. DispatchIncomingCommands only executes a single received command per call. If a command was dispatched, the return value is true and the method should be called again. This method is called by **Service**() until currently available commands are dispatched.

C#

```
public virtual bool DispatchIncomingCommands();
```

Remarks

In general, this method should be called until it returns false. In a few cases, it might make sense to pause dispatching (if a certain state is reached and the app needs to load data, before it should handle new **events**).

The callbacks to the peer's **IPhotonPeerListener** are executed in the same thread that is calling DispatchIncomingCommands. This makes things easier in a game loop: **Event** execution won't clash with painting objects or the game logic.

2.1.10.2.4 PhotonPeer.EstablishEncryption Method

This method creates a public key for this client and exchanges it with the server.

C#

```
public bool EstablishEncryption();
```

Returns

If operation could be enqueued for sending

Remarks

Encryption is not instantly available but calls OnStatusChanged when it finishes. Check for **StatusCode** EncryptionEstablished and EncryptionFailedToEstablish.

Calling this method sets **IsEncryptionAvailable** to false. This method must be called before the "encrypt" parameter of **OpCustom** can be used.

2.1.10.2.5 PhotonPeer.FetchServerTimestamp Method

This will fetch the server's timestamp and update the approximation for property ServerTimeInMilliseconds.

The server time approximation will NOT become more accurate by repeated calls. Accuracy currently depends on a single roundtrip which is done as fast as possible.

The command used for this is immediately acknowledged by the server. This makes sure the roundtrip time is low and the timestamp + rountriptime / 2 is close to the original value.

C#

```
public virtual void FetchServerTimestamp();
```

2.1.10.2.6 OpCustom Method

2.1.10.2.6.1 PhotonPeer.OpCustom Method (OperationRequest, bool, byte, bool)

Allows the client to send any operation to the Photon Server by setting any opCode and the operation's parameters.

C#

```
public virtual bool OpCustom(OperationRequest operationRequest, bool sendReliable, byte
channelId, bool encrypt);
```

Parameters

Parameters	Description
OperationRequest operationRequest	The operation to call on Photon.
bool sendReliable	Use unreliable (false) if the call might get lost (when it's content is soon outdated).
byte channelId	Defines the sequence of requests this operation belongs to.

| bool encrypt | Encrypt request before sending. Depends on **IsEncryptionAvailable**. |

Returns

If operation could be enqueued for sending

Remarks

Variant with an **OperationRequest** object.

This variant offers an alternative way to describe a operation request. Operation code and it's parameters are wrapped up in a object. Still, the parameters are a Dictionary.

2.1.10.2.6.2 PhotonPeer.OpCustom Method (byte, Dictionary<byte, object>, bool)

Channel-less wrapper for OpCustom().

C#

```
public virtual bool OpCustom(byte customOpCode, Dictionary<byte, object>
customOpParameters, bool sendReliable);
```

Parameters

Parameters	Description
byte customOpCode	**Operations** are handled by their byte-typed code. The codes of the "Lite" application are in the struct **LiteOpCode**.
Dictionary<byte, object> customOpParameters	Containing parameters as key-value pair. The key is byte-typed, while the value is any serializable datatype.
bool sendReliable	Selects if the operation must be acknowledged or not. If false, the operation is not guaranteed to reach the server.

Returns

If operation could be enqueued for sending

2.1.10.2.6.3 PhotonPeer.OpCustom Method (byte, Dictionary<byte, object>, bool, byte)

Allows the client to send any operation to the Photon Server by setting any opCode and the operation's parameters.

C#

```
public virtual bool OpCustom(byte customOpCode, Dictionary<byte, object>
customOpParameters, bool sendReliable, byte channelId);
```

Parameters

Parameters	Description
byte customOpCode	**Operations** are handled by their byte-typed code. The codes of the "Lite" application are in the struct **LiteOpCode**.
Dictionary<byte, object> customOpParameters	Containing parameters as key-value pair. The key is byte-typed, while the value is any serializable datatype.
bool sendReliable	Selects if the operation must be acknowledged or not. If false, the operation is not guaranteed to reach the server.
byte channelId	The channel in which this operation should be sent.

Returns

If operation could be enqueued for sending

Description

Photon can be extended with new operations which are identified by a single byte, defined server side and known as operation code (opCode). Similarly, the operation's parameters are defined server side as byte keys of values, which a client sends as customOpParameters accordingly.

This is explained in more detail as "**Custom Operations**".

2.1.10.2.6.4 **PhotonPeer.OpCustom Method (byte, Dictionary<byte, object>, bool, byte, bool)**

Allows the client to send any operation to the Photon Server by setting any opCode and the operation's parameters.

Variant with encryption parameter.

Use this only after encryption was established by **EstablishEncryption** and waiting for the OnStateChanged callback.

C#

```
public virtual bool OpCustom(byte customOpCode, Dictionary<byte, object>
customOpParameters, bool sendReliable, byte channelId, bool encrypt);
```

Parameters

Parameters	Description
byte customOpCode	**Operations** are handled by their byte-typed code. The codes of the "Lite" application are in the struct **LiteOpCode**.
Dictionary<byte, object> customOpParameters	Containing parameters as key-value pair. The key is byte-typed, while the value is any serializable datatype.
bool sendReliable	Selects if the operation must be acknowledged or not. If false, the operation is not guaranteed to reach the server.
byte channelId	The channel in which this operation should be sent.
bool encrypt	Can only be true, while **IsEncryptionAvailable** is true, too.

Returns

If operation could be enqueued for sending

2.1.10.2.7 **PhotonPeer.RegisterType Method**

Registers new types/classes for de/serialization and the fitting methods to call for this type.

C#

```
public static bool RegisterType(Type customType, byte code, SerializeMethod
serializeMethod, DeserializeMethod constructor);
```

Parameters

Parameters	Description
Type customType	Type (class) to register.
byte code	A byte-code used as shortcut during transfer of this Type.
SerializeMethod serializeMethod	Method delegate to create a byte[] from a customType instance.
DeserializeMethod constructor	Method delegate to create instances of customType's from byte[].

Returns

If the Type was registered successfully.

Remarks

SerializeMethod and **DeserializeMethod** are complementary: Feed the product of serializeMethod to the constructor, to get a comparable instance of the object.

After registering a Type, it can be used in **events** and operations and will be serialized like built-in types.

2.1.10.2.8 **PhotonPeer.SendAcksOnly Method**

C#

```
public virtual bool SendAcksOnly();
```

Description

This is SendAcksOnly, a member of class PhotonPeer.

2.1.10.2.9 **PhotonPeer.SendOutgoingCommands Method**

This method creates a UDP/TCP package for outgoing commands (operations and acknowledgements) and sends them to the server. This method is also called by **Service**().

C#

```
public virtual bool SendOutgoingCommands();
```

Returns

The if commands are not yet sent. Udp limits it's package size, Tcp doesnt.

Remarks

As the Photon library does not create any UDP/TCP packages by itself. Instead, the application fully controls how many packages are sent and when. A tradeoff, an application will lose connection, if it is no longer calling SendOutgoingCommands or **Service**.

If multiple operations and ACKs are waiting to be sent, they will be aggregated into one package. The package fills in this order: ACKs for received commands A "Ping" - only if no reliable data was sent for a while Starting with the lowest Channel-Nr: Reliable Commands in channel Unreliable Commands in channel

This gives a higher priority to lower channels.

A longer interval between sends will lower the overhead per sent operation but increase the internal delay (which adds "lag").

Call this 2..20 times per second (depending on your target platform).

2.1.10.2.10 **PhotonPeer.Service Method**

This method excutes **DispatchIncomingCommands** and **SendOutgoingCommands** in your application Thread-context.

C#

```
public virtual void Service();
```

Remarks

The Photon client libraries are designed to fit easily into a game or application. The application is in control of the context (thread) in which incoming **events** and responses are executed and has full control of the creation of UDP/TCP packages.

Sending packages and dispatching received messages are two separate tasks. Service combines them into one method at the cost of control. It calls **DispatchIncomingCommands** and **SendOutgoingCommands**.

Call this method regularly (2..20 times a second).

This will Dispatch ANY remaining buffered responses and **events** AND will send queued outgoing commands. Fewer calls might be more effective if a device cannot send many packets per second, as multiple operations might be combined into one package.

See Also

PhotonPeer.DispatchIncomingCommands, **PhotonPeer.SendOutgoingCommands**

Example

You could replace Service by:

while (**DispatchIncomingCommands**()); //Dispatch until everything is Dispatched... **SendOutgoingCommands**(); //Send a UDP/TCP package with outgoing messages

2.1.10.2.11 **PhotonPeer.StopThread Method**

This method immediately closes a connection (pure client side) and ends related listening Threads.

C#

```
public virtual void StopThread();
```

Remarks

Unlike **Disconnect**, this method will simply stop to listen to the server. Udp connections will timeout. If the connections was open, this will trigger a callback to OnStatusChanged with code StatusCode.**Disconnect**.

2.1.10.2.12 **PhotonPeer.TrafficStatsReset Method**

Creates new instances of **TrafficStats** and starts a new timer for those.

C#

```
public void TrafficStatsReset();
```

2.1.10.2.13 **PhotonPeer.VitalStatsToString Method**

Returns a string of the most interesting connection statistics. When you have issues on the client side, these might contain hints about the issue's cause.

C#

```
public string VitalStatsToString(bool all);
```

Parameters

Parameters	Description
bool all	If true, Incoming and Outgoing low-level stats are included in the string.

Returns

Stats as string.

2.1.10.3 **PhotonPeer Properties**

2.1.10.3.1 **PhotonPeer.ByteCountCurrentDispatch Property**

Gets the size of the dispatched event or operation-result in bytes. This value is set before OnEvent() or OnOperationResponse() is called (within **DispatchIncomingCommands**()).

C#

```
public int ByteCountCurrentDispatch;
```

Remarks

Get this value directly in OnEvent() or OnOperationResponse(). Example: void OnEvent(...) { int eventSizeInBytes = this.peer.ByteCountCurrentDispatch; //...

void OnOperationResponse(...) { int resultSizeInBytes = this.peer.ByteCountCurrentDispatch; //...

2.1.10.3.2 **PhotonPeer.ByteCountLastOperation Property**

Gets the size of the last serialized operation call in bytes. The value includes all headers for this single operation but excludes those of UDP, Enet Package Headers and TCP.

C#

```
public int ByteCountLastOperation;
```

Remarks

Get this value immediately after calling an operation. Example: this.litepeer.OpJoin("myroom"); int opjoinByteCount = this.peer.ByteCountLastOperation;

2.1.10.3.3 PhotonPeer.BytesIn Property

Gets count of all bytes coming in (including headers, excluding UDP/TCP overhead)

C#

```
public long BytesIn;
```

2.1.10.3.4 PhotonPeer.BytesOut Property

Gets count of all bytes going out (including headers, excluding UDP/TCP overhead)

C#

```
public long BytesOut;
```

2.1.10.3.5 PhotonPeer.ChannelCount Property

Gets / sets the number of channels available in UDP connections with Photon. Photon Channels are only supported for UDP. The default ChannelCount is 2. Channel IDs start with 0 and 255 is a internal channel.

C#

```
public byte ChannelCount;
```

2.1.10.3.6 PhotonPeer.CommandBufferSize Property

Initial size internal lists for incoming/outgoing commands (reliable and unreliable).

C#

```
public int CommandBufferSize;
```

Remarks

This sets only the initial size. All lists simply grow in size as needed. This means that incoming or outgoing commands can pile up and consume heap size if **Service** is not called often enough to handle the messages in either direction.

Configure the **WarningSize**, to get callbacks when the lists reach a certain size.

UDP: Incoming and outgoing commands each have separate buffers for reliable and unreliable sending. There are additional buffers for "sent commands" and "ACKs". TCP: Only two buffers exist: incoming and outgoing commands.

2.1.10.3.7 PhotonPeer.CrcEnabled Property

While not connected, this controls if the next connection(s) should use a per-package CRC checksum.

C#

```
public bool CrcEnabled;
```

Remarks

While turned on, the client and server will add a CRC checksum to every sent package. The checksum enables both sides to detect and ignore packages that were corrupted during transfer. Corrupted packages have the same impact as lost packages: They require a re-send, adding a delay and could lead to timeouts.

Building the checksum has a low processing overhead but increases integrity of sent and received data. Packages discarded

2

due to failed CRC cecks are counted in **PhotonPeer.PacketLossByCrc**.

2.1.10.3.8 **PhotonPeer.DebugOut Property**

Sets the level (and amount) of debug output provided by the library.

C#

```
public DebugLevel DebugOut;
```

Remarks

This affects the callbacks to **IPhotonPeerListener.DebugReturn**. Default Level: Error.

2.1.10.3.9 **PhotonPeer.DisconnectTimeout Property**

Milliseconds after which a reliable UDP command triggers a timeout disconnect, unless acknowledged by server. This value currently only affects UDP connections. DisconnectTimeout is not an exact value for a timeout. The exact timing of the timeout depends on the frequency of **Service**() calls and commands that are sent with long roundtrip-times and variance are checked less often for re-sending!

DisconnectTimeout and **SentCountAllowance** are competing settings: either might trigger a disconnect on the client first, depending on the values and Rountrip Time. Default: 10000 ms.

C#

```
public int DisconnectTimeout;
```

2.1.10.3.10 **PhotonPeer.HttpUrlParameters Property**

This string is added as simple GET parameters to the end of the server url + peerID combination. Include a starting "&" in your parameters as the peerID is always added to the **ServerAddress**. This value will be added to all following (http) operations, so make sure to clear it if needed. Can be applied even to "connect" (set before calling connect, clear / reset when connection is established). Set before making a Operation call and don't forget to clear or set it again before the next operation.

C#

```
public string HttpUrlParameters;
```

2.1.10.3.11 **PhotonPeer.IsEncryptionAvailable Property**

This property is set internally, when OpExchangeKeysForEncryption successfully finished. While it's true, encryption can be used for operations.

C#

```
public bool IsEncryptionAvailable;
```

2.1.10.3.12 **PhotonPeer.IsSendingOnlyAcks Property**

C#

```
public bool IsSendingOnlyAcks;
```

Todo

Comment this!

2.1.10.3.13 **PhotonPeer.IsSimulationEnabled Property**

Gets or sets the network simulation "enabled" setting. Changing this value also locks this peer's sending and when setting false, the internally used queues are executed (so setting to false can take some cycles).

C#

```
public virtual bool IsSimulationEnabled;
```

2.1.10.3.14 PhotonPeer.LimitOfUnreliableCommands Property

Limits the queue of received unreliable commands within **DispatchIncomingCommands** before dispatching them. This works only in UDP. This limit is applied when you call **DispatchIncomingCommands**. If this client (already) received more than LimitOfUnreliableCommands, it will throw away the older ones instead of dispatching them. This can produce bigger gaps for unreliable commands but your client catches up faster.

C#

```
public int LimitOfUnreliableCommands;
```

Remarks

This can be useful when the client couldn't dispatch anything for some time (cause it was in a room but loading a level). If set to 20, the incoming unreliable queues are truncated to 20. If 0, all received unreliable commands will be dispatched. This is a "per channel" value, so each channel can hold up to LimitOfUnreliableCommands commands. This value interacts with **DispatchIncomingCommands**: If that is called less often, more commands get skipped.

2.1.10.3.15 PhotonPeer.Listener Property

Gets the **IPhotonPeerListener** of this instance (set in constructor). Can be used in derived classes for Listener.DebugReturn().

C#

```
public IPhotonPeerListener Listener;
```

2.1.10.3.16 PhotonPeer.LocalMsTimestampDelegate Property

This setter for the (local-) timestamp delegate replaces the default Environment.TickCount with any equal function.

C#

```
public SupportClass.IntegerMillisecondsDelegate LocalMsTimestampDelegate;
```

Remarks

About Environment.TickCount: The value of this property is derived from the system timer and is stored as a 32-bit signed integer. Consequently, if the system runs continuously, TickCount will increment from zero to Int32..::.MaxValue for approximately 24.9 days, then jump to Int32..::.MinValue, which is a negative number, then increment back to zero during the next 24.9 days.

Exceptions

Exceptions	Description
Exception	Exception is thrown peer.**PeerState** is not PS_DISCONNECTED.

2.1.10.3.17 PhotonPeer.LocalTimeInMilliSeconds Property

Gets a local timestamp in milliseconds by calling **SupportClass.GetTickCount**(). See **LocalMsTimestampDelegate**.

C#

```
[Obsolete("Should be replaced by: SupportClass.GetTickCount(). Internally this is used,
too.")]
public int LocalTimeInMilliSeconds;
```

2.1.10.3.18 PhotonPeer.MaximumTransferUnit Property

The Maximum Trasfer Unit (MTU) defines the (network-level) packet-content size that is guaranteed to arrive at the server in one piece. The Photon **Protocol** uses this size to split larger data into packets and for receive-buffers of packets.

C#

```
public int MaximumTransferUnit;
```

Remarks

This value affects the Packet-content. The resulting UDP packages will have additional headers that also count against the package size (so it's bigger than this limit in the end) Setting this value while being connected is not allowed and will throw an Exception. Minimum is 520. Huge values won't speed up connections in most cases!

2.1.10.3.19 PhotonPeer.NetworkSimulationSettings Property

Gets the settings for built-in Network Simulation for this peer instance while **IsSimulationEnabled** will enable or disable them. Once obtained, the settings can be modified by changing the properties.

C#

```
public NetworkSimulationSet NetworkSimulationSettings;
```

2.1.10.3.20 PhotonPeer.OutgoingStreamBufferSize Property

Defines the initial size of an internally used MemoryStream for Tcp. The MemoryStream is used to aggregate operation into (less) send calls, which uses less resoures.

C#

```
public static int OutgoingStreamBufferSize;
```

Remarks

The size is not restricting the buffer and does not affect when poutgoing data is actually sent.

2.1.10.3.21 PhotonPeer.PacketLossByCrc Property

Count of packages dropped due to failed CRC checks for this connection.

C#

```
public int PacketLossByCrc;
```

Description

CrcEnabled

2.1.10.3.22 PhotonPeer.PeerID Property

This peer's ID as assigned by the server or 0 if not using UDP. Will be 0xFFFF before the client connects.

C#

```
public string PeerID;
```

Remarks

Used for debugging only. This value is not useful in everyday Photon usage.

2.1.10.3.23 PhotonPeer.PeerState Property

This is the (low level) state of the connection to the server of a **PhotonPeer**. It is managed internally and read-only.

C#

```
public PeerStateValue PeerState;
```

Remarks

Don't mix this up with the **StatusCode** provided in IPhotonListener.OnStatusChanged(). Applications should use the **StatusCode** of OnStatusChanged() to track their state, as it also covers the higher level initialization between a client and Photon.

2.1.10.3.24 PhotonPeer.QueuedIncomingCommands Property

Count of all currently received but not-yet-Dispatched reliable commands (**events** and operation results) from all channels.

C#

```
public int QueuedIncomingCommands;
```

2.1.10.3.25 PhotonPeer.QueuedOutgoingCommands Property

Count of all commands currently queued as outgoing, including all channels and reliable, unreliable.

C#

```
public int QueuedOutgoingCommands;
```

2.1.10.3.26 PhotonPeer.ResentReliableCommands Property

Count of commands that got repeated (due to local repeat-timing before an ACK was received).

C#

```
public int ResentReliableCommands;
```

2.1.10.3.27 PhotonPeer.RoundTripTime Property

Time until a reliable command is acknowledged by the server.

The value measures network latency and for UDP it includes the server's ACK-delay (setting in config). In TCP, there is no ACK-delay, so the value is slightly lower (if you use default settings for Photon).

RoundTripTime is updated constantly. Every reliable command will contribute a fraction to this value.

This is also the approximate time until a raised event reaches another client or until an operation result is available.

C#

```
public int RoundTripTime;
```

2.1.10.3.28 PhotonPeer.RoundTripTimeVariance Property

Changes of the roundtriptime as variance value. Gives a hint about how much the time is changing.

C#

```
public int RoundTripTimeVariance;
```

2.1.10.3.29 PhotonPeer.SentCountAllowance Property

Number of send retries before a peer is considered lost/disconnected. Default: 5. The initial timeout countdown of a command is calculated by the current roundTripTime + 4 * roundTripTimeVariance. Please note that the timeout span until a command will be resent is not constant, but based on the roundtrip time at the initial sending, which will be doubled with every failed retry.

DisconnectTimeout and SentCountAllowance are competing settings: either might trigger a disconnect on the client first, depending on the values and Rountrip Time.

C#

```
public int SentCountAllowance;
```

2.1.10.3.30 PhotonPeer.ServerAddress Property

The server address which was used in **PhotonPeer.Connect**() or null (before **Connect**() was called).

C#

```
public string ServerAddress;
```

Remarks

The ServerAddress can only be changed for HTTP connections (to replace one that goes through a Loadbalancer with a direct URL).

2.1.10.3.31 PhotonPeer.ServerTimeInMilliSeconds Property

Approximated Environment.TickCount value of server (while connected).

C#

```
public int ServerTimeInMilliSeconds;
```

Description

0 until connected. While connected, the value is an approximation of the server's current timestamp.

Remarks

UDP: The server's timestamp is automatically fetched after connecting (once). This is done internally by a command which is acknowledged immediately by the server. TCP: The server's timestamp fetched with each ping but set only after connecting (once).

The approximation will be off by +/- 10ms in most cases. Per peer/client and connection, the offset will be constant (unless **FetchServerTimestamp**() is used). A constant offset should be better to adjust for. Unfortunately there is no way to find out how much the local value differs from the original.

The approximation adds RoundtripTime / 2 and uses this.**LocalTimeInMilliSeconds** to calculate in-between values (this property returns a new value per tick).

The value sent by Photon equals Environment.TickCount in the logic layer (e.g. Lite).

2.1.10.3.32 PhotonPeer.TimePingInterval Property

Sets the milliseconds without reliable command before a ping command (reliable) will be sent (Default: 1000ms). The ping command is used to keep track of the connection in case the client does not send reliable commands by itself. A ping (or reliable commands) will update the **RoundTripTime** calculation.

C#

```
public int TimePingInterval;
```

2.1.10.3.33 PhotonPeer.TimestampOfLastSocketReceive Property

Stores timestamp of the last time anything (!) was received from the server (including low level Ping and ACKs but also **events** and operation-returns). This is not the time when something was dispatched. If you enable NetworkSimulation, this value is affected as well.

C#

```
public int TimestampOfLastSocketReceive;
```

2.1.10.3.34 **PhotonPeer.TrafficStatsElapsedMs Property**

Returns the count of milliseconds the stats are enabled for tracking.

C#

```
public long TrafficStatsElapsedMs;
```

2.1.10.3.35 **PhotonPeer.TrafficStatsEnabled Property**

Enables the traffic statistics of a peer: **TrafficStatsIncoming**, **TrafficStatsOutgoing** and TrafficstatsGameLevel (nothing else). Default value: false (disabled).

C#

```
public bool TrafficStatsEnabled;
```

2.1.10.3.36 **PhotonPeer.TrafficStatsGameLevel Property**

Gets a statistic of incoming and outgoing traffic, split by operation, operation-result and event. **Operations** are outgoing traffic, results and **events** are incoming. Includes the per-command header sizes (Udp: Enet Command Header or Tcp: Message Header).

C#

```
public TrafficStatsGameLevel TrafficStatsGameLevel;
```

2.1.10.3.37 **PhotonPeer.TrafficStatsIncoming Property**

Gets the byte-count of incoming "low level" messages, which are either Enet Commands or Tcp Messages. These include all headers, except those of the underlying internet protocol Udp or Tcp.

C#

```
public TrafficStats TrafficStatsIncoming;
```

2.1.10.3.38 **PhotonPeer.TrafficStatsOutgoing Property**

Gets the byte-count of outgoing "low level" messages, which are either Enet Commands or Tcp Messages. These include all headers, except those of the underlying internet protocol Udp or Tcp.

C#

```
public TrafficStats TrafficStatsOutgoing;
```

2.1.10.3.39 **PhotonPeer.UsedProtocol Property**

The protocol this Peer uses to connect to Photon.

C#

```
public ConnectionProtocol UsedProtocol;
```

2.1.10.3.40 **PhotonPeer.WarningSize Property**

The WarningSize is used test all message queues for congestion (in and out, reliable and unreliable). OnStatusChanged will be called with a warning if a queue holds WarningSize commands or a multiple of it. Default: 100.

C#

```
public int WarningSize;
```

Example

If command is received, OnStatusChanged will be called when the respective command queue has 100, 200, 300 ... items.

2.1.11 Protocol Class

Provides tools for the Exit Games Protocol

C#

```csharp
public class Protocol;
```

Protocol Methods

	Name	Description
⇒◈⑤	**Deserialize**	Deserialize returns an object reassembled from the given byte-array.
⇒◈⑤	**Serialize**	Serializes an float typed value into a byte-array (target) starting at the also given targetOffset. The altered offset is known to the caller, because it is given via a referenced parameter.

2.1.11.1 Protocol Methods

2.1.11.1.1 Deserialize Method

2.1.11.1.1.1 Protocol.Deserialize Method (byte[])

Deserialize returns an object reassembled from the given byte-array.

C#

```csharp
public static object Deserialize(byte[] serializedData);
```

Parameters

Parameters	Description
byte[] serializedData	The byte-array to be Deserialized

Returns

The Deserialized object

2.1.11.1.1.2 Protocol.Deserialize Method (out float, byte[], int)

Deserialize fills the given float typed value with the given byte-array (source) starting at the also given offset. The result is placed in a variable (value). There is no need to return a value because the parameter value is given by reference. The altered offset is this way also known to the caller.

C#

```csharp
public static void Deserialize(out float value, byte[] source, ref int offset);
```

Parameters

Parameters	Description
out float value	The float value to deserialize
target	The byte-array to deserialize from
targetOffset	The offset in the byte-array

2.1.11.1.1.3 Protocol.Deserialize Method (out int, byte[], int)

Deserialize fills the given int typed value with the given byte-array (source) starting at the also given offset. The result is placed in a variable (value). There is no need to return a value because the parameter value is given by reference. The altered offset is this way also known to the caller.

C#

```csharp
public static void Deserialize(out int value, byte[] source, ref int offset);
```

Parameters

Parameters	Description
out int value	The int value to deserialize into
target	The byte-array to deserialize from
targetOffset	The offset in the byte-array

2.1.11.1.1.4 Protocol.Deserialize Method (out short, byte[], int)

Deserialize fills the given short typed value with the given byte-array (source) starting at the also given offset. The result is placed in a variable (value). There is no need to return a value because the parameter value is given by reference. The altered offset is this way also known to the caller.

C#

```csharp
public static void Deserialize(out short value, byte[] source, ref int offset);
```

Parameters

Parameters	Description
out short value	The short value to deserialized into
target	The byte-array to deserialize from
targetOffset	The offset in the byte-array

2.1.11.1.2 Serialize Method

2.1.11.1.2.1 Protocol.Serialize Method (float, byte[], int)

Serializes an float typed value into a byte-array (target) starting at the also given targetOffset. The altered offset is known to the caller, because it is given via a referenced parameter.

C#

```csharp
public static void Serialize(float value, byte[] target, ref int targetOffset);
```

Parameters

Parameters	Description
float value	The float value to be serialized
byte[] target	The byte-array to serialize the short to
ref int targetOffset	The offset in the byte-array

2.1.11.1.2.2 Protocol.Serialize Method (int, byte[], int)

Serializes an int typed value into a byte-array (target) starting at the also given targetOffset. The altered offset is known to the caller, because it is given via a referenced parameter.

C#

```csharp
public static void Serialize(int value, byte[] target, ref int targetOffset);
```

Parameters

Parameters	Description
int value	The int value to be serialized
byte[] target	The byte-array to serialize the short to
ref int targetOffset	The offset in the byte-array

2.1.11.1.2.3 **Protocol.Serialize Method (object)**

Serialize creates a byte-array from the given object and returns it.

C#

```csharp
public static byte[] Serialize(object obj);
```

Parameters

Parameters	Description
serializedData	The object to serialize

Returns

The serialized byte-array

2.1.11.1.2.4 **Protocol.Serialize Method (short, byte[], int)**

Serializes a short typed value into a byte-array (target) starting at the also given targetOffset. The altered offset is known to the caller, because it is given via a referenced parameter.

C#

```csharp
public static void Serialize(short value, byte[] target, ref int targetOffset);
```

Parameters

Parameters	Description
short value	The short value to be serialized
byte[] target	The byte-array to serialize the short to
ref int targetOffset	The offset in the byte-array

2.1.12 **PRPCAttribute Class**

C#

```csharp
public class PRPCAttribute : Attribute;
```

Description

This is class PRPCAttribute.

2.1.13 **SupportClass Class**

Contains several (more or less) useful static methods, mostly used for debugging.

C#

```csharp
public class SupportClass;
```

SupportClass Classes

	Name	Description
	ThreadSafeRandom	Class to wrap static access to the random.**Next**() call in a thread safe manner.

SupportClass Delegates

Name	Description
IntegerMillisecondsDelegate	This is nested type SupportClass.IntegerMillisecondsDelegate.

SupportClass Methods

	Name	Description
⇒⧫S	**ByteArrayToString**	Converts a byte-array to string (useful as debugging output). Uses BitConverter.ToString(list) internally after a null-check of list.
⇒⧫S	**CalculateCrc**	This is CalculateCrc, a member of class SupportClass.
⇒⧫S	**CallInBackground**	This is CallInBackground, a member of class SupportClass.
⇒⧫S	**DictionaryToString**	This method returns a string, representing the content of the given IDictionary. Returns "null" if parameter is null.
⇒⧫S	**GetMethods**	This is GetMethods, a member of class SupportClass.
⇒⧫S	**GetTickCount**	Gets the local machine's "milliseconds since start" value (precision is described in remarks).
⇒⧫S	**HashtableToString**	This is HashtableToString, a member of class SupportClass.
⇒⧫S	**NumberToByteArray**	Inserts the number's value into the byte array, using Big-Endian order (a.k.a. Network-byte-order).
⇒⧫S	**WriteStackTrace**	Writes the exception's stack trace to the received stream.

2.1.13.1 SupportClass Classes

2.1.13.1.1 SupportClass.ThreadSafeRandom Class

Class to wrap static access to the random.**Next**() call in a thread safe manner.

C#

```
public class ThreadSafeRandom;
```

ThreadSafeRandom Methods

	Name	Description
⇒⧫S	**Next**	This is Next, a member of class ThreadSafeRandom.

2.1.13.1.1.1 ThreadSafeRandom Methods

2.1.13.1.1.1.1 SupportClass.ThreadSafeRandom.Next Method

C#

```
public static int Next();
```

Description

This is Next, a member of class ThreadSafeRandom.

2.1.13.2 SupportClass Methods

2.1.13.2.1 SupportClass.ByteArrayToString Method

Converts a byte-array to string (useful as debugging output). Uses BitConverter.ToString(list) internally after a null-check of list.

C#

```
public static string ByteArrayToString(byte[] list);
```

Parameters

Parameters	Description
byte[] list	Byte-array to convert to string.

Returns

List of bytes as string.

2.1.13.2.2 SupportClass.CalculateCrc Method

C#

```
public static uint CalculateCrc(byte[] buffer, int length);
```

Description

This is CalculateCrc, a member of class SupportClass.

2.1.13.2.3 SupportClass.CallInBackground Method

C#

```
public static void CallInBackground(Func<bool> myThread);
```

Description

This is CallInBackground, a member of class SupportClass.

2.1.13.2.4 DictionaryToString Method

2.1.13.2.4.1 SupportClass.DictionaryToString Method (IDictionary)

This method returns a string, representing the content of the given IDictionary. Returns "null" if parameter is null.

C#

```
public static string DictionaryToString(IDictionary dictionary);
```

Parameters

Parameters	Description
IDictionary dictionary	IDictionary to return as string.

Returns

The string representation of keys and values in IDictionary.

2.1.13.2.4.2 SupportClass.DictionaryToString Method (IDictionary, bool)

This method returns a string, representing the content of the given IDictionary. Returns "null" if parameter is null.

C#

```
public static string DictionaryToString(IDictionary dictionary, bool includeTypes);
```

Parameters

Parameters	Description
IDictionary dictionary	IDictionary to return as string.
bool includeTypes	

2.1.13.2.5 SupportClass.GetMethods Method

C#

```
public static List<MethodInfo> GetMethods(Type type, Type attribute);
```

Description

This is GetMethods, a member of class SupportClass.

2.1.13.2.6 SupportClass.GetTickCount Method

Gets the local machine's "milliseconds since start" value (precision is described in remarks).

C#

```
public static int GetTickCount();
```

Returns

Fraction of the current time in Milliseconds (this is not a proper datetime timestamp).

Remarks

This method uses Environment.TickCount (cheap but with only 16ms precision). **PhotonPeer.LocalMsTimestampDelegate** is available to set the delegate (unless already connected).

2.1.13.2.7 SupportClass.HashtableToString Method

C#

```
[Obsolete("Use DictionaryToString() instead.")]
public static string HashtableToString(Hashtable hash);
```

Description

This is HashtableToString, a member of class SupportClass.

2.1.13.2.8 NumberToByteArray Method

2.1.13.2.8.1 SupportClass.NumberToByteArray Method (byte[], int, int)

Inserts the number's value into the byte array, using Big-Endian order (a.k.a. Network-byte-order).

C#

```
[Obsolete("Use Protocol.Serialize() instead.")]
public static void NumberToByteArray(byte[] buffer, int index, int number);
```

Parameters

Parameters	Description
byte[] buffer	Byte array to write into.
int index	Index of first position to write to.
int number	Number to write.

2.1.13.2.8.2 SupportClass.NumberToByteArray Method (byte[], int, short)

Inserts the number's value into the byte array, using Big-Endian order (a.k.a. Network-byte-order).

C#

```
[Obsolete("Use Protocol.Serialize() instead.")]
public static void NumberToByteArray(byte[] buffer, int index, short number);
```

Parameters

Parameters	Description
byte[] buffer	Byte array to write into.
int index	Index of first position to write to.
short number	Number to write.

2.1.13.2.9 SupportClass.WriteStackTrace Method

Writes the exception's stack trace to the received stream.

C#

```
public static void WriteStackTrace(System.Exception throwable, System.IO.TextWriter stream);
```

Parameters

Parameters	Description
System.Exception throwable	Exception to obtain information from.
System.IO.TextWriter stream	Output sream used to write to.

2.1.13.3 SupportClass Delegates

2.1.13.3.1 SupportClass.IntegerMillisecondsDelegate Delegate

C#

```
public delegate int IntegerMillisecondsDelegate();
```

Description

This is nested type SupportClass.IntegerMillisecondsDelegate.

2.1.14 TrafficStats Class

C#

```
public class TrafficStats;
```

Description

This is class TrafficStats.

TrafficStats Methods

	Name	Description
⇒♦	**ToString**	This is ToString, a member of class TrafficStats.

TrafficStats Properties

	Name	Description
	ControlCommandBytes	This is ControlCommandBytes, a member of class TrafficStats.
	ControlCommandCount	This is ControlCommandCount, a member of class TrafficStats.
	FragmentCommandBytes	This is FragmentCommandBytes, a member of class TrafficStats.
	FragmentCommandCount	This is FragmentCommandCount, a member of class TrafficStats.
	PackageHeaderSize	Gets the byte-size of per-package headers.
	ReliableCommandBytes	This is ReliableCommandBytes, a member of class TrafficStats.

	ReliableCommandCount	Counts commands created/received by this client, ignoring repeats (out command count can be higher due to repeats).
	TotalCommandBytes	This is TotalCommandBytes, a member of class TrafficStats.
	TotalCommandCount	This is TotalCommandCount, a member of class TrafficStats.
	TotalCommandsInPackets	This is TotalCommandsInPackets, a member of class TrafficStats.
	TotalPacketBytes	Gets count of bytes as traffic, excluding UDP/TCP headers (42 bytes / x bytes).
	TotalPacketCount	This is TotalPacketCount, a member of class TrafficStats.
	UnreliableCommandBytes	This is UnreliableCommandBytes, a member of class TrafficStats.
	UnreliableCommandCount	This is UnreliableCommandCount, a member of class TrafficStats.

2.1.14.1 TrafficStats Methods

2.1.14.1.1 TrafficStats.ToString Method

C#

```
public override string ToString();
```

Description

This is ToString, a member of class TrafficStats.

2.1.14.2 TrafficStats Properties

2.1.14.2.1 TrafficStats.ControlCommandBytes Property

C#

```
public int ControlCommandBytes;
```

Description

This is ControlCommandBytes, a member of class TrafficStats.

2.1.14.2.2 TrafficStats.ControlCommandCount Property

C#

```
public int ControlCommandCount;
```

Description

This is ControlCommandCount, a member of class TrafficStats.

2.1.14.2.3 TrafficStats.FragmentCommandBytes Property

C#

```
public int FragmentCommandBytes;
```

Description

This is FragmentCommandBytes, a member of class TrafficStats.

2.1.14.2.4 TrafficStats.FragmentCommandCount Property

C#

```
public int FragmentCommandCount;
```

Description

This is FragmentCommandCount, a member of class TrafficStats.

2.1.14.2.5 TrafficStats.PackageHeaderSize Property

Gets the byte-size of per-package headers.

C#

```
public int PackageHeaderSize;
```

2.1.14.2.6 TrafficStats.ReliableCommandBytes Property

C#

```
public int ReliableCommandBytes;
```

Description

This is ReliableCommandBytes, a member of class TrafficStats.

2.1.14.2.7 TrafficStats.ReliableCommandCount Property

Counts commands created/received by this client, ignoring repeats (out command count can be higher due to repeats).

C#

```
public int ReliableCommandCount;
```

2.1.14.2.8 TrafficStats.TotalCommandBytes Property

C#

```
public int TotalCommandBytes;
```

Description

This is TotalCommandBytes, a member of class TrafficStats.

2.1.14.2.9 TrafficStats.TotalCommandCount Property

C#

```
public int TotalCommandCount;
```

Description

This is TotalCommandCount, a member of class TrafficStats.

2.1.14.2.10 TrafficStats.TotalCommandsInPackets Property

C#

```
public int TotalCommandsInPackets;
```

Description

This is TotalCommandsInPackets, a member of class TrafficStats.

2.1.14.2.11 TrafficStats.TotalPacketBytes Property

Gets count of bytes as traffic, excluding UDP/TCP headers (42 bytes / x bytes).

C#

```
public int TotalPacketBytes;
```

2.1.14.2.12 TrafficStats.TotalPacketCount Property

C#

```
public int TotalPacketCount;
```

Description

This is TotalPacketCount, a member of class TrafficStats.

2.1.14.2.13 TrafficStats.UnreliableCommandBytes Property

C#

```
public int UnreliableCommandBytes;
```

Description

This is UnreliableCommandBytes, a member of class TrafficStats.

2.1.14.2.14 TrafficStats.UnreliableCommandCount Property

C#

```
public int UnreliableCommandCount;
```

Description

This is UnreliableCommandCount, a member of class TrafficStats.

2.1.15 TrafficStatsGameLevel Class

Only in use as long as **PhotonPeer.TrafficStatsEnabled** = true;

C#

```
public class TrafficStatsGameLevel;
```

TrafficStatsGameLevel Methods

Name	Description
ResetMaximumCounters	Resets the values that can be maxed out, like **LongestDeltaBetweenDispatching**. See remarks.
ToString	This is ToString, a member of class TrafficStatsGameLevel.
ToStringVitalStats	This is ToStringVitalStats, a member of class TrafficStatsGameLevel.

TrafficStatsGameLevel Properties

Name	Description
DispatchCalls	Gets number of calls of DispatchIncomingCommands.
DispatchIncomingCommandsCalls	Gets number of calls of DispatchIncomingCommands.
EventByteCount	Gets sum of byte-cost of incoming **events**.
EventCount	Gets count of incoming **events**.

🖼		**LongestDeltaBetweenDispatching**	Gets longest time between subsequent calls to DispatchIncomginCommands in milliseconds.
🖼		**LongestDeltaBetweenSending**	Gets longest time between subsequent calls to SendOutgoingCommands in milliseconds.
🖼		**LongestEventCallback**	Gets longest time a call to OnEvent (in your code) took. If such a callback takes long, it will lower the network performance and might lead to timeouts.
🖼		**LongestEventCallbackCode**	Gets EventCode that caused the **LongestEventCallback**. See that description.
🖼		**LongestOpResponseCallback**	Gets longest time it took to complete a call to OnOperationResponse (in your code). If such a callback takes long, it will lower the network performance and might lead to timeouts.
🖼		**LongestOpResponseCallbackOpCode**	Gets OperationCode that causes the **LongestOpResponseCallback**. See that description.
🖼		**OperationByteCount**	Gets sum of outgoing operations in bytes.
🖼		**OperationCount**	Gets count of outgoing operations.
🖼		**ResultByteCount**	Gets sum of byte-cost of incoming operation-results.
🖼		**ResultCount**	Gets count of incoming operation-results.
🖼		**SendOutgoingCommandsCalls**	Gets number of calls of SendOutgoingCommands.
🖼		**TotalByteCount**	Gets sum of byte-cost of all "logic level" messages.
🖼		**TotalIncomingByteCount**	Gets sum of byte-cost of all incoming "logic level" messages.
🖼		**TotalIncomingMessageCount**	Gets sum of counted incoming "logic level" messages.
🖼		**TotalMessageCount**	Gets sum of counted "logic level" messages.
🖼		**TotalOutgoingByteCount**	Gets sum of byte-cost of all outgoing "logic level" messages (= **OperationByteCount**).
🖼		**TotalOutgoingMessageCount**	Gets sum of counted outgoing "logic level" messages (= **OperationCount**).

2.1.15.1 TrafficStatsGameLevel Methods

2.1.15.1.1 TrafficStatsGameLevel.ResetMaximumCounters Method

Resets the values that can be maxed out, like **LongestDeltaBetweenDispatching**. See remarks.

C#

```
public void ResetMaximumCounters();
```

Remarks

Set to 0: **LongestDeltaBetweenDispatching**, **LongestDeltaBetweenSending**, **LongestEventCallback**, **LongestEventCallbackCode**, **LongestOpResponseCallback**, **LongestOpResponseCallbackOpCode**. Also resets internal values: timeOfLastDispatchCall and timeOfLastSendCall (so intervals are tracked correctly).

2.1.15.1.2 TrafficStatsGameLevel.ToString Method

C#

```
public override string ToString();
```

Description

This is ToString, a member of class TrafficStatsGameLevel.

2

2.1.15.1.3 TrafficStatsGameLevel.ToStringVitalStats Method

C#

```
public string ToStringVitalStats();
```

Description

This is ToStringVitalStats, a member of class TrafficStatsGameLevel.

2.1.15.2 TrafficStatsGameLevel Properties

2.1.15.2.1 TrafficStatsGameLevel.DispatchCalls Property

Gets number of calls of DispatchIncomingCommands.

C#

```
[Obsolete("Use DispatchIncomingCommandsCalls, which has proper naming.")]
public int DispatchCalls;
```

2.1.15.2.2 TrafficStatsGameLevel.DispatchIncomingCommandsCalls Property

Gets number of calls of DispatchIncomingCommands.

C#

```
public int DispatchIncomingCommandsCalls;
```

2.1.15.2.3 TrafficStatsGameLevel.EventByteCount Property

Gets sum of byte-cost of incoming **events**.

C#

```
public int EventByteCount;
```

2.1.15.2.4 TrafficStatsGameLevel.EventCount Property

Gets count of incoming **events**.

C#

```
public int EventCount;
```

2.1.15.2.5 TrafficStatsGameLevel.LongestDeltaBetweenDispatching Property

Gets longest time between subsequent calls to DispatchIncomginCommands in milliseconds.

C#

```
public int LongestDeltaBetweenDispatching;
```

Notes

This is not a crucial timing for the networking. Long gaps just add "local lag" to **events** that are available already.

2.1.15.2.6 TrafficStatsGameLevel.LongestDeltaBetweenSending Property

Gets longest time between subsequent calls to SendOutgoingCommands in milliseconds.

C#

```
public int LongestDeltaBetweenSending;
```

Notes

This is a crucial value for network stability. Without calling SendOutgoingCommands, nothing will be sent to the server, who might time out this client.

2.1.15.2.7 TrafficStatsGameLevel.LongestEventCallback Property

Gets longest time a call to OnEvent (in your code) took. If such a callback takes long, it will lower the network performance and might lead to timeouts.

C#

```
public int LongestEventCallback;
```

2.1.15.2.8 TrafficStatsGameLevel.LongestEventCallbackCode Property

Gets EventCode that caused the **LongestEventCallback**. See that description.

C#

```
public byte LongestEventCallbackCode;
```

2.1.15.2.9 TrafficStatsGameLevel.LongestOpResponseCallback Property

Gets longest time it took to complete a call to OnOperationResponse (in your code). If such a callback takes long, it will lower the network performance and might lead to timeouts.

C#

```
public int LongestOpResponseCallback;
```

2.1.15.2.10 TrafficStatsGameLevel.LongestOpResponseCallbackOpCode Property

Gets OperationCode that causes the **LongestOpResponseCallback**. See that description.

C#

```
public byte LongestOpResponseCallbackOpCode;
```

2.1.15.2.11 TrafficStatsGameLevel.OperationByteCount Property

Gets sum of outgoing operations in bytes.

C#

```
public int OperationByteCount;
```

2.1.15.2.12 TrafficStatsGameLevel.OperationCount Property

Gets count of outgoing operations.

C#

```
public int OperationCount;
```

2.1.15.2.13 TrafficStatsGameLevel.ResultByteCount Property

Gets sum of byte-cost of incoming operation-results.

C#

```
public int ResultByteCount;
```

2.1.15.2.14 TrafficStatsGameLevel.ResultCount Property

Gets count of incoming operation-results.

C#

```
public int ResultCount;
```

2.1.15.2.15 TrafficStatsGameLevel.SendOutgoingCommandsCalls Property

Gets number of calls of SendOutgoingCommands.

C#

```
public int SendOutgoingCommandsCalls;
```

2.1.15.2.16 TrafficStatsGameLevel.TotalByteCount Property

Gets sum of byte-cost of all "logic level" messages.

C#

```
public int TotalByteCount;
```

2.1.15.2.17 TrafficStatsGameLevel.TotalIncomingByteCount Property

Gets sum of byte-cost of all incoming "logic level" messages.

C#

```
public int TotalIncomingByteCount;
```

2.1.15.2.18 TrafficStatsGameLevel.TotalIncomingMessageCount Property

Gets sum of counted incoming "logic level" messages.

C#

```
public int TotalIncomingMessageCount;
```

2.1.15.2.19 TrafficStatsGameLevel.TotalMessageCount Property

Gets sum of counted "logic level" messages.

C#

```
public int TotalMessageCount;
```

2.1.15.2.20 TrafficStatsGameLevel.TotalOutgoingByteCount Property

Gets sum of byte-cost of all outgoing "logic level" messages (= **OperationByteCount**).

C#

```
public int TotalOutgoingByteCount;
```

2.1.15.2.21 TrafficStatsGameLevel.TotalOutgoingMessageCount Property

Gets sum of counted outgoing "logic level" messages (= **OperationCount**).

C#

```
public int TotalOutgoingMessageCount;
```

2.2 Interfaces

The following table lists interfaces in this documentation.

Interfaces

	Name	Description
⊸	**IPhotonPeerListener**	Callback interface for the Photon client side. Must be provided to a new **PhotonPeer** in its constructor.

2.2.1 IPhotonPeerListener Interface

Callback interface for the Photon client side. Must be provided to a new **PhotonPeer** in its constructor.

C#

```
public interface IPhotonPeerListener;
```

Remarks

These methods are used by your **PhotonPeer** instance to keep your app updated. Read each method's description and check out the samples to see how to use them.

IPhotonPeerListener Methods

	Name	Description
≡●	**DebugReturn**	Provides textual descriptions for various error conditions and noteworthy situations. In cases where the application needs to react, a call to **OnStatusChanged** is used. **OnStatusChanged** gives "feedback" to the game, DebugReturn provies human readable messages on the background.
≡●	**OnEvent**	Called whenever an event from the Photon Server is dispatched.
≡●	**OnOperationResponse**	Callback method which gives you (async) responses for called operations.
≡●	**OnStatusChanged**	OnStatusChanged is called to let the game know when asyncronous actions finished or when errors happen.

2.2.1.1 IPhotonPeerListener Methods

2.2.1.1.1 IPhotonPeerListener.DebugReturn Method

Provides textual descriptions for various error conditions and noteworthy situations. In cases where the application needs to react, a call to **OnStatusChanged** is used. **OnStatusChanged** gives "feedback" to the game, DebugReturn provies human readable messages on the background.

C#

```
void DebugReturn(DebugLevel level, string message);
```

Parameters

Parameters	Description
DebugLevel level	**DebugLevel** (severity) of the message.
string message	Debug text. Print to System.Console or screen.

Remarks

All debug output of the library will be reported through this method. Print it or put it in a buffer to use it on-screen. Use **PhotonPeer.DebugOut** to select how verbose the output is.

2.2.1.1.2 **IPhotonPeerListener.OnEvent Method**

Called whenever an event from the Photon Server is dispatched.

C#

```
void OnEvent(EventData eventData);
```

Parameters

Parameters	Description
EventData eventData	The event currently being dispatched.

Remarks

Events are used for communication between clients and allow the server to update clients over time. The creation of an event is often triggered by an operation (called by this client or an other).

Each event carries its specific content in its Parameters. Your application knows which content to expect by checking the event's 'type', given by the event's Code.

Events can be defined and extended server-side.

If you use the Lite application as base, several **events** like EvJoin and EvLeave are already defined. For these **events** and their Parameters, the library provides constants, so check: **LiteEventCode** and **LiteEventKey** classes. Lite also allows you to come up with custom **events** on the fly, purely client-side. To do so, use **LitePeer.OpRaiseEvent**.

 Events are buffered on the client side and must be Dispatched. This way, OnEvent is always taking place in the same thread as a **PhotonPeer.DispatchIncomingCommands** call.

2.2.1.1.3 **IPhotonPeerListener.OnOperationResponse Method**

Callback method which gives you (async) responses for called operations.

C#

```
void OnOperationResponse(OperationResponse operationResponse);
```

Parameters

Parameters	Description
OperationResponse operationResponse	The response to an operation-call.

Remarks

Like methods, operations can have a result. As operation-calls are non-blocking, the response for any operation of this peer is provided by this method. As example: Joining a room on a Lite-based server will return a list of players currently in gamey your actorNumber and some other values.

This method is used as general callback for all operations. Each response corresponds to a certain "type" of operation by its OperationCode (see: **Operations**).

The "Lite Application" uses these OpCodes:

- **LiteOpCode.Join** for OpJoin, contains the actorNr of "this" player
- **LiteOpCode.Leave** when leaving a room
- **LiteOpCode.RaiseEvent** for OpRaiseEvent, if this was sent as reliable command
- **LiteOpCode.SetProperties** for OpSetPropertiesOfActor and OpSetPropertiesOfGame

Example

When you join a room, the server will assign a consecutive number to each client: the "actorNr" or "player number". This is sent back in the OperationResult's Parameters as value of key **LiteEventKey.ActorNr**.

Fetch your actorNr of a Join response like this:

```
int actorNr = (int)operationResponse[(byte)LiteOpKey.ActorNr];
```

2.2.1.1.4 IPhotonPeerListener.OnStatusChanged Method

OnStatusChanged is called to let the game know when asycronous actions finished or when errors happen.

C#

```
void OnStatusChanged(StatusCode statusCode);
```

Parameters

Parameters	Description
StatusCode statusCode	A code to identify the situation.

Remarks

Not all of the many **StatusCode** values will apply to your game. Example: If you don't use encryption, the respective status changes are never made.

The values are all part of the **StatusCode** enumeration and described value-by-value.

2.3 Structs, Records, Enums

The following table lists structs, records, enums in this documentation.

Enumerations

	Name	Description
🔧	**ConnectionProtocol**	These are the options that can be used as underlying transport protocol.
🔧	**DebugLevel**	Level / amount of DebugReturn callbacks. Each debug level includes output for lower ones: OFF, ERROR, WARNING, INFO, ALL.
🔧	**EventCaching**	Lite - OpRaiseEvent allows you to cache **events** and automatically send them to joining players in a room. **Events** are cached per event code and player: **Event** 100 (example!) can be stored once per player. Cached **events** can be modified, replaced and removed.
🔧	**GpType**	The gp type.
🔧	**LitePropertyTypes**	Lite - Flags for "types of properties", being used as filter in OpGetProperties.
🔧	**PeerStateValue**	Value range for a Peer's connection and initialization state, as returned by the PeerState property.
🔧	**ReceiverGroup**	Lite - OpRaiseEvent lets you chose which actors in the room should receive **events**. By default, **events** are sent to "Others" but you can overrule this.
🔧	**StatusCode**	Enumeration of situations that change the peers internal status. Used in calls to OnStatusChanged to inform your application of various situations that might happen.

2.3.1 ConnectionProtocol Enumeration

These are the options that can be used as underlying transport protocol.

C#

```
public enum ConnectionProtocol : byte {
  Udp = 0,
  Tcp = 1,
  Http = 2,
  RHttp = 3
}
```

Members

Members	Description
Udp = 0	Use UDP to connect to Photon, which allows you to send operations reliable or unreliable on demand.
Tcp = 1	Use TCP to connect to Photon.
Http = 2	Use HTTP connections to connect a Photon Master (not available in regular Photon SDK).
RHttp = 3	Use Realtime HTTP connections to connect a Photon Master (not available in regular Photon SDK).

2.3.2 DebugLevel Enumeration

Level / amount of DebugReturn callbacks. Each debug level includes output for lower ones: OFF, ERROR, WARNING, INFO, ALL.

C#

```
public enum DebugLevel : byte {
  OFF = 0,
  ERROR = 1,
  WARNING = 2,
  INFO = 3,
  ALL = 5
}
```

Members

Members	Description
OFF = 0	No debug out.
ERROR = 1	Only error descriptions.
WARNING = 2	Warnings and errors.
INFO = 3	Information about internal workflows, warnings and errors.
ALL = 5	Most complete workflow description (but lots of debug output), info, warnings and errors.

2.3.3 EventCaching Enumeration

Lite - OpRaiseEvent allows you to cache **events** and automatically send them to joining players in a room. **Events** are cached per event code and player: **Event** 100 (example!) can be stored once per player. Cached **events** can be modified, replaced and removed.

C#

```
public enum EventCaching : byte {
  DoNotCache = 0,
  MergeCache = 1,
  ReplaceCache = 2,
  RemoveCache = 3,
  AddToRoomCache = 4,
  AddToRoomCacheGlobal = 5,
  RemoveFromRoomCache = 6,
  RemoveFromRoomCacheForActorsLeft = 7
}
```

Members

Members	Description
DoNotCache = 0	Default value (not sent).
MergeCache = 1	Will merge this event's keys with those already cached.
ReplaceCache = 2	Replaces the event cache for this eventCode with this event's content.
RemoveCache = 3	Removes this event (by eventCode) from the cache.
AddToRoomCache = 4	Adds an event to the room's cache.
AddToRoomCacheGlobal = 5	Adds this event to the cache for actor 0 (becoming a "globally owned" event in the cache).
RemoveFromRoomCache = 6	Remove fitting event from the room's cache.
RemoveFromRoomCacheForActorsLeft = 7	Removes **events** of players who already left the room (cleaning up).

Remarks

Caching works only combination with **ReceiverGroup** options Others and All.

2.3.4 GpType Enumeration

The gp type.

C#

```
internal enum GpType : byte {
  Unknown = 0,
  Array = (byte)'y',
  Boolean = (byte)'o',
  Byte = (byte)'b',
  ByteArray = (byte)'x',
  ObjectArray = (byte)'z',
  Short = (byte)'k',
  Float = (byte)'f',
  Dictionary = (byte)'D',
  Double = (byte)'d',
  Hashtable = (byte)'h',
  Integer = (byte)'i',
  IntegerArray = (byte)'n',
  Long = (byte)'l',
  String = (byte)'s',
  StringArray = (byte)'a',
  Vector = (byte)'v',
  Custom = (byte)'c',
  Null = (byte)'*',
  EventData = (byte)'e',
  OperationRequest = (byte)'q',
  OperationResponse = (byte)'p'
}
```

2

Members

Members	Description
Unknown = 0	Unkown type.
Array = (byte)'y'	0x79 (121)
Boolean = (byte)'o'	0x6F
Byte = (byte)'b'	A byte value.
ByteArray = (byte)'x'	An array of bytes.
ObjectArray = (byte)'z'	An array of objects.
Short = (byte)'k'	A 16-bit integer value.
Float = (byte)'f'	A 32-bit floating-point value.
Dictionary = (byte)'D'	0x44 (68)
Double = (byte)'d'	A 64-bit floating-point value.
Hashtable = (byte)'h'	A Hashtable.
Integer = (byte)'i'	0x69 (105)
IntegerArray = (byte)'n'	An array of 32-bit integer values.
Long = (byte)'l'	A 64-bit integer value.
String = (byte)'s'	A string value.
StringArray = (byte)'a'	An array of string values.
Vector = (byte)'v'	A vector.
Custom = (byte)'c'	A costum type
Null = (byte)'*'	Null value don't have types.

2.3.5 LitePropertyTypes Enumeration

Lite - Flags for "types of properties", being used as filter in OpGetProperties.

C#

```csharp
[Flags]
public enum LitePropertyTypes : byte {
  None = 0x00,
  Game = 0x01,
  Actor = 0x02,
  GameAndActor = Game|Actor
}
```

Members

Members	Description
None = 0x00	(0x00) Flag type for no property type.
Game = 0x01	(0x01) Flag type for game-attached properties.
Actor = 0x02	(0x02) Flag type for actor related propeties.
GameAndActor = Game|Actor	(0x01) Flag type for game AND actor properties. Equal to 'Game'

2

2.3.6 PeerStateValue Enumeration

Value range for a Peer's connection and initialization state, as returned by the PeerState property.

C#

```csharp
public enum PeerStateValue : byte {
  Disconnected = 0,
```

```
    Connecting = 1,
    InitializingApplication = 10,
    Connected = 3,
    Disconnecting = 4
}
```

Members

Members	Description
Disconnected = 0	The peer is disconnected and can't call **Operations**. Call Connect().
Connecting = 1	The peer is establishing the connection: opening a socket, exchanging packages with Photon.
InitializingApplication = 10	The connection is established and now sends the application name to Photon.
Connected = 3	The peer is connected and initialized (selected an application). You can now use operations.
Disconnecting = 4	The peer is disconnecting. It sent a disconnect to the server, which will acknowledge closing the connection.

Remarks

While this is not the same as the **StatusCode** of **IPhotonPeerListener.OnStatusChanged**(), it directly relates to it. In most cases, it makes more sense to build a game's state on top of the OnStatusChanged() as you get changes.

2.3.7 ReceiverGroup Enumeration

Lite - OpRaiseEvent lets you chose which actors in the room should receive **events**. By default, **events** are sent to "Others" but you can overrule this.

C#

```
public enum ReceiverGroup : byte {
    Others = 0,
    All = 1,
    MasterClient = 2
}
```

Members

Members	Description
Others = 0	Default value (not sent). Anyone else gets my event.
All = 1	Everyone in the current room (including this peer) will get this event.
MasterClient = 2	The server sends this event only to the actor with the lowest actorNumber.

2.3.8 StatusCode Enumeration

Enumeration of situations that change the peers internal status. Used in calls to OnStatusChanged to inform your application of various situations that might happen.

C#

```
public enum StatusCode : int {
    Connect = 1024,
    Disconnect = 1025,
    Exception = 1026,
    ExceptionOnConnect = 1023,
    SecurityExceptionOnConnect = 1022,
```

```
        QueueOutgoingReliableWarning = 1027,
        QueueOutgoingUnreliableWarning = 1029,
        SendError = 1030,
        QueueOutgoingAcksWarning = 1031,
        QueueIncomingReliableWarning = 1033,
        QueueIncomingUnreliableWarning = 1035,
        QueueSentWarning = 1037,
        InternalReceiveException = 1039,
        TimeoutDisconnect = 1040,
        DisconnectByServer = 1041,
        DisconnectByServerUserLimit = 1042,
        DisconnectByServerLogic = 1043,
        TcpRouterResponseOk = 1044,
        TcpRouterResponseNodeIdUnknown = 1045,
        TcpRouterResponseEndpointUnknown = 1046,
        TcpRouterResponseNodeNotReady = 1047,
        EncryptionEstablished = 1048,
        EncryptionFailedToEstablish = 1049
}
```

Members

Members	Description
Connect = 1024	the **PhotonPeer** is connected.See {@link PhotonListener#OnStatusChanged}*
Disconnect = 1025	the **PhotonPeer** just disconnected.See {@link PhotonListener#OnStatusChanged}*
Exception = 1026	the **PhotonPeer** encountered an exception and will disconnect, too.See {@link PhotonListener#OnStatusChanged}*
ExceptionOnConnect = 1023	the **PhotonPeer** encountered an exception while opening the incoming connection to the server. The server could be down / not running or the client has no network or a misconfigured DNS.See {@link PhotonListener#OnStatusChanged}*
SecurityExceptionOnConnect = 1022	Used on platforms that throw a security exception on connect. Unity3d does this, e.g., if a webplayer build could not fetch a policy-file from a remote server.
QueueOutgoingReliableWarning = 1027	**PhotonPeer** outgoing queue is filling up. send more often.
QueueOutgoingUnreliableWarning = 1029	**PhotonPeer** outgoing queue is filling up. send more often.
SendError = 1030	Sending command failed. Either not connected, or the requested channel is bigger than the number of initialized channels.
QueueOutgoingAcksWarning = 1031	**PhotonPeer** outgoing queue is filling up. send more often.
QueueIncomingReliableWarning = 1033	**PhotonPeer** incoming queue is filling up. Dispatch more often.
QueueIncomingUnreliableWarning = 1035	**PhotonPeer** incoming queue is filling up. Dispatch more often.
QueueSentWarning = 1037	**PhotonPeer** incoming queue is filling up. Dispatch more often.
InternalReceiveException = 1039	Exception, if a server cannot be connected. Most likely, the server is not responding. Ask user to try again later.
TimeoutDisconnect = 1040	Disconnection due to a timeout (client did no longer receive ACKs from server).
DisconnectByServer = 1041	Disconnect by server due to timeout (received a disconnect command, cause server misses ACKs of client).
DisconnectByServerUserLimit = 1042	Disconnect by server due to concurrent user limit reached (received a disconnect command).
DisconnectByServerLogic = 1043	Disconnect by server due to server's logic (received a disconnect command).

| EncryptionEstablished = 1048 | (1048) Value for OnStatusChanged()-call, when the encryption-setup for secure communication finished successfully. |
| EncryptionFailedToEstablish = 1049 | (1049) Value for OnStatusChanged()-call, when the encryption-setup failed for some reason. Check debug logs. |

Remarks

Most of these codes are referenced somewhere else in the documentation when they are relevant to methods.

2.4 Types

The following table lists types in this documentation.

Types

Name	Description
DeserializeMethod	Type of deserialization methods to add custom type support. Use **PhotonPeer.RegisterType**() to register new types with serialization and deserialization methods.
SerializeMethod	Type of serialization methods to add custom type support. Use PhotonPeer.ReisterType() to register new types with serialization and deserialization methods.

2.4.1 DeserializeMethod Type

Type of deserialization methods to add custom type support. Use **PhotonPeer.RegisterType**() to register new types with serialization and deserialization methods.

C#

```
public delegate object DeserializeMethod(byte[] serializedCustomObject);
```

Parameters

Parameters	Description
serializedCustomObject	The framwork passes in the data it got by the associated **SerializeMethod**. The type code and length are stripped and applied before a DeserializeMethod is called.

Returns

Return a object of the type that was associated with this method through RegisterType().

2.4.2 SerializeMethod Type

Type of serialization methods to add custom type support. Use PhotonPeer.ReisterType() to register new types with serialization and deserialization methods.

C#

```
public delegate byte SerializeMethod(object customObject);
```

Parameters

Parameters	Description
customObject	The method will get objects passed that were registered with it in RegisterType().

Returns

Return a byte[] that resembles the object passed in. The framework will surround it with length and type info, so don't include it.

2

Index

M

N

O

Q

R