MULTIPLAYER

View online: https://www.construct.net/en/make-games/manuals/construct-3/plugin-reference/multiplayer

The **Multiplayer object** provides features to develop real-time online multiplayer games. It uses WebRTC DataChannels, a peer-to-peer networking feature of modern browsers, to transmit gameplay data.

Scripting

When using JavaScript or TypeScript coding, the features of this object can be accessed via the IMultiplayerObjectType script interface.

Multiplayer features

The Multiplayer object supports a number of features to enable low-latency gameplay over the Internet:

- UDP-based transmission for minimal latency avoiding head-of-line blocking, with optional reliable modes
- NAT traversal to connect through common router/network setups
- Compensation for poor quality connections with high latency, packet delay variation (PDV), and packet loss
- Interpolation and extrapolation modes to ensure smooth in-game motion
- Scirra-hosted signalling server to connect peers to each other
- Scirra-hosted TURN server to improve connectivity across some types of networks
- Authoritative host model to prevent cheating, with local input prediction to hide input latency
- LAN game support for near-zero latency gameplay, including support for mixed LAN/Internet games
- Automatic data compression to save bandwidth
- Automatic bandwidth reduction when objects are not changing, avoiding redundant repeated transmissions of identical data
- Binary transmission with control over specific datatypes to minimise bandwidth
- Large binary transfers, such as to send an entire file between peers
- Latency and packet loss simulation for realistic local testing
- Support for lag compensation
- Support for both peer-to-peer games (not needing a server) and centrally-hosted games (using a server)

Learning to make Multiplayer games

Designing Multiplayer games is challenging. It is essential to follow the introductory tutorial series to gain an understanding of how to use the Multiplayer object correctly. You are likely to struggle if you try to skip ahead and start making a game without fully understanding how various aspects of multiplayer games work. Mistakes can also result in degraded gameplay quality, with unnecessary lag.

The four introductory tutorials are:

1. Multiplayer tutorial 1: concepts
2. Multiplayer tutorial 2: chat room
3. Multiplayer tutorial 3: pong
4. Multiplayer tutorial 4: real-time game

You can also find examples of the Multiplayer features by searching for *Multiplayer* in the Start Page.

Signalling and hosting

The signalling server is a central server where players go to find each other. Scirra host an official signalling server at **wss://multiplayer.construct.net**. The signalling server does not transmit any gameplay data; it serves only to connect peers to the game host by relaying connection information like IP addresses. Players must connect and log in to the signalling server before they can join any rooms.

The first player in to a room becomes the host. The host acts as the server for the game, transferring actual gameplay data. Any player can be the host. This means games can run without needing any server hosting, saving you from having to pay bandwidth bills to run your multiplayer game.

If you have a large or particularly latency-sensitive game you can still run your game with a dedicated server host to take advantage of its better quality connection. This can be achieved by starting a browser on the server, starting the game and make sure the server is the first to join the room so it becomes the host. Now the server connection will be hosting the game. You can host multiple games on a server by opening multiple browser tabs (a hosted game can continue to work in a background tab). Note this means your server is genuinely running the game - as mentioned, the signalling server only helps peers connect to the room host, and actual gameplay data will be transmitted through your server if it is the room host. It is not necessary to run your own signalling server to achieve this.

Peer IDs

The signalling server assigns every player who connects a *Peer ID*. This is a short string of random characters that uniquely identifies them, such as "ABCD". When designing multiplayer

games, it is best to identify peers by their peer ID instead of their alias (display name), since their alias could potentially change but their peer ID never changes so long as they remain connected.

Sending messages

Peers only connect to the host. In other words, the host has a connection to every peer, and peers only have one connection to the host. In order for two peers to communicate, the information must be relayed via the host.

Consequently, the host can directly message any peer in the room, but peers can only directly message the host. See the chat tutorial for an example of how to relay messages through the host so other peers can receive messages from another peer.

The host also has the unique ability to *broadcast* a message. This sends the same message to every peer in the room.

Messages can be sent with different reliability modes. These are:

- **Reliable ordered**: all messages are guaranteed to arrive, and the order received will be the same as the order sent. However this can have the highest latency since a lost message will need retransmitting and will hold up every message sent after it. This mode is suitable for chat messages.
- **Reliable unordered:** all messages are guaranteed to arrive, but the order received can be different from the order sent. This allows improved latency since a lost message will be held up as it is retransmitted, but subsequent messages can still get through quickly without being held up. This mode is suitable for gameplay events that occur independently of each other, such as "door opened" or "explosion occurred".
- **Unreliable:** messages are not guaranteed to arrive, and the order received can be different from the order sent. The network will make a "best effort" attempt to deliver the message, but the message may be dropped and the recipient will never receive it and will have no indication it was even attempted to be sent. This mode is suitable for high-bandwidth or regularly-sent messages, where if a message is dropped it is likely to be shortly followed up by another one. Note the Multiplayer object uses this mode internally for object positions and data when using *Sync object*; avoid re-implementing this functionality with this mode.

Updating games

The host is authoritative for both gameplay data and its settings. If you change any aspect of your game, such as the synced objects or variables, the client input values, or bandwidth profile, the host's values are authoritative when there is a mismatch between the peer and the host. In order to avoid confusion or broken games, use a different game instance name when distributing an update, so only peers using the same version end up connecting to each other.

Binary transfers

The Multiplayer object supports sending binary messages by transmitting the contents of a Binary Data object. However due to the underlying networking technologies, a single binary

message cannot be too large (common limits are in the range 64 KB - 256 KB), which poses a problem if you want to send a larger file to another peer.

The Multiplayer object provides a **Binary transfer** feature to help send larger amounts of data. See the Multiplayer file transfer example for a demonstration of how it works. When making a binary transfer, the process works like this:

1. The sending peer starts a binary transfer. The *receive started* event fires for the receiving peer, where the receiver can access metadata like the filename and transfer size.
2. The Multiplayer object internally splits the data up into smaller chunks that can each be safely transmitted. The sending peer then starts sending these chunks over the network. Both sides will fire progress triggers to be able to observe how the transfer is progressing.
3. Once all chunks have been successfully transmitted, the transfer completes and the receiving peer as an identical copy of the binary data that was transmitted.

Binary transfers can have an optional tag assigned to identify different transfers. For convenience each transfer can also specify a filename and MIME type that the receiver can also access for purposes like transferring a named file. Transfers can also be cancelled while still in progress, such as if a user aborts a transfer. Transfers also benefit from automatic data compression to save bandwidth. Binary data transfers don't allow specifying a reliability mode - they always implicitly send with *reliable ordered* mode as the feature is designed to guarantee transmission of an identical copy of the data.

As noted in the section above, peers only connect to the host and so direct communication is only possible between the host and peers. However the binary transfer feature supports an automatic relay through the host. This means two non-host peers can transfer binary data to each other. In this case each chunk will be sent first to the host, and then the host will send it on to the destination peer. This saves you having to implement a relay yourself. However it is important to note the privacy and security implications of this. While data transmitted over the network is automatically encrypted, the host will see unencrypted binary transfers while relaying them. Bear this in mind if transferring any privacy or security sensitive data between non-host peers. Also note this will increase the bandwidth used by the host, especially if multiple binary transfers occur simultaneously.

Multiplayer conditions

On (binary transfer) complete

On any (binary transfer) complete

Triggered on the receiving end of a binary transfer when the transfer has completed. The full transferred data is then available in the chosen Binary Data object. The *On complete* trigger fires only for the specified transfer tag, whereas the *any* variant trigger fires for any transfer (with the tag available in the *Tag* expression).

On (binary transfer) event

On any (binary transfer) event

These triggers fire at various stages of a binary transfer, either for a transfer with a specific tag, or for any transfer (with the tag available in the *Tag* expression). The types of events are:

- *receive started*: triggered for the receiving peer when a binary transfer starts. Metadata is available through expressions like *Filename* and *TransferSize*.
- *progress*: triggered for both sending and receiving peers as the transfer progresses, with the progess available in the *TransferProgress* expression.
- *cancelled*: triggered for both sending and receiving peers when a transfer is explicitly cancelled by the sender with the *Cancel binary transfer* action.
- *send complete*: triggered for the sending peer when the transfer has finished being sent (as the *On complete* trigger is only fired by the receiver).

Compare peer count

Compare the number of peers currently in the room, if a room has been joined. The peer count includes the host so is at least 1 if in a room.

Is host

True if in a room and acting as the host. The host of the room is effectively the server for the game. Peers only connect to the host, and the host must relay data if two other peers are to communicate.

On any peer message

Triggered when a message with any tag is received. The *Message*, *Tag*, *FromID* and *FromAlias* expressions can be used to retrieve information about the received message. The order messages are received, or whether a sent message is received at all, depends on the reliability mode chosen when the message was originally sent.

On kicked

Triggered if kicked from the current room. This can occur if the host quits, the connection to the host could not be established, or the host otherwise decides to forcibly remove you from the room. After *On kicked* the player is no longer in the room and must re-join a room to be able to participate in a game.

On peer connected

Triggered when another peer joins the same room. It also triggers once per peer already in the room when joining an existing room, including the host. The *PeerID* and *PeerAlias* expressions identify the relevant peer.

On peer disconnected

Triggered when a peer disconnects from the room. The *PeerID* and *PeerAlias* expressions identify the peer that left. The *LeaveReason* expression can indicate why the peer left, such as if they intentionally quit or timed out.

On host disconnected

Triggered when a peer is disconnected from the host. This trigger works the same as using *On peer disconnected* and checking that the disconnected peer ID is the host's ID. When the host disconnects, it ends all communication as the host was the peer's only point of contact, and typically represents the end of the session.

On peer message

Triggered when a message sent with a specific tag is received. The *Message*, *FromID* and *FromAlias* expressions can be used to retrieve information about the received message. The order messages are received, or whether a sent message is received at all, depends on the reliability mode chosen when the message was originally sent.

On peer binary message

As with *On peer message*, but triggered when receiving a binary message sent with the *Send binary message* action. The received data is stored in the chosen Binary Data object. Note that binary messages do not have tags, but this data can be included in the transmitted binary data.

Is ready for input

True when a peer is ready to send input to the host. This means *On client update* has triggered at least once, or is about to trigger. Do not allow players using input prediction to move or act before this condition is true or *On client update* has triggered: doing so will simply cause an input prediction error since the host is not yet ready to receive input.

On client update

Triggered when a peer is about to send its input state to the host. The input state should be updated in this trigger using the *Set client state* action.

Is connected

True if currently connected to the signalling server. It is not necessary to be connected to a signalling server once connected to the room host.

Is in room

True if currently in a room on the signalling server.

Is logged in

True if currently connected to the signalling server and successfully logged in.

On connected

Triggered after successfully connecting to the signalling server. In order to join rooms, it is necessary to next log in to the server.

On disconnected

Triggered after disconnecting from the signalling server.

On error

Triggered if an error occurs with the signalling server. The *ErrorMessage* expression indicates the type of error that occurred.

On game instance list

Triggered after *Request game instance list* when the list has been received from the signalling server. The *List...* expressions can be used to retrieve the list details.

On joined room

Triggered after the *Join room* or *Auto-join room* actions when the room has been successfully joined. The *Is host* condition can be used to determine if the player is the first joining peer and has been assigned the room host.

On left room

Triggered after the *Leave room* action when the room has been left. The room is also left if *On kicked* triggers.

On logged in

Triggered after the *Log in* action if the login is successful. Once logged in it is possible to join rooms. Note the signalling server may have assigned a different alias to the one requested if it was already taken; use the *MyAlias* expression to determine the actual alias in use.

On room list

Triggered after the *Request room list* action when the room list has been successfully received. The room list expressions can then be used to inspect the received list.

Multiplayer actions

Transfer peer binary

Begin a binary data transfer to a peer identified by their peer ID. The data to send is in a specified Binary Data object. An optional tag, filename and MIME type may be specified which the receiver can also use. For more details see the section on *Binary transfers* above.

Cancel binary transfer

Cancel a currently in progress binary transfer specified by the peer ID it is being sent to and the tag of the transfer. Both the sender and receiver will fire the *cancelled* event and no further data will be transmitted.

Add client input value

Use on startup to add a value that peers send to the host to indicate their input state. Each client input value has a tag to identify it; use this tag to update the value with the *Set client state* action. To avoid wasting bandwidth, use the lowest *Precision* that can still hold all the values that need to be set. If using *setbit* / *getbit* to send key states, you must use *None* for *Interpolation*; otherwise use *Linear* for values like positions, or *Angular* if representing an angle.

Associate object with peer

If *Sync object* is used on objects which represent peers in the game, use this action to associate an instance of an object with a given peer. Typically this is used in the object's *On created* trigger. Both the host and peers must associate objects with peers. Associated objects are automatically destroyed when the corresponding peer leaves.

Disconnect

Disconnect from the room. If the room host, all players are kicked; otherwise the peer disconnects from the host. The room is also left on the signalling server, so another room can be joined afterwards.

Send message

Send a message to a specific peer with a given reliability mode. Peers can only message the host (and the *Peer ID* field must be left empty), but the host can send a message to any peer. Message tags can be used to identify messages for different purposes, such as "chat" or "gameplay-event". The message must be a text string, but could also be JSON data such as

from an Array or Dictionary *AsJSON* expression; however be sure to avoid wasting bandwidth. The order messages arrive, or whether it is guaranteed to arrive at all, depends on the reliability mode.

Send binary message

As with *Send message*, but the data to be sent is specified by a Binary Data object. Binary messages do not use a tag, although this data can be incorporated in the transmitted data.

> *The underlying networking technologies usually impose a limit on the maximum size of binary data that can be transmitted in a single message. Commonly this is in the range 64 KB - 256 KB (although this limit applies after automatic data compression, so sometimes larger messages may be allowed if they compress to within the allowed size). To send larger amounts of data such as for file transfers, use the binary transfer feature instead.*

Set bandwidth profile

Switch between *Internet* or *LAN* (Local Area Network) bandwidth modes. The bandwidth profile must be set before joining a room, and only the host's setting is used for all peers in the room. The default mode is *Internet*, which sends updates 30 times a second with an 80ms buffer. *LAN* mode sends updates 60 times a second with a 40ms buffer. *LAN* mode will use about double the bandwidth of *Internet* mode and will degrade gameplay quality more if there is latency or PDV in the connection. *LAN* mode should never be used for Internet games - it is intended for networks where bandwidth is effectively unlimited and latency effectively zero, which is typically only the case with local area networks, and taking advantage of this can improve gameplay quality. *Internet* mode should however work well over LANs, so if in doubt leave it on that.

Simulate latency

Simulate latency, PDV and packet loss on all inbound and outbound messages. This can be useful for making local testing more realistic, since unlike the Internet latency is effectively non-existent. For local testing it is only necessary to simulate latency on the host, since that guarantees every message in the game will have delay added; it is not necessary to also simulate latency on the peers. The latency for an individual message is calculated as the latency plus a random value from zero to the PDV. The packet loss indicates the chance an unreliable message is lost entirely, or in the case of reliable messages that retransmission is necessary and the latency is multiplied.

Sync object

Automatically sync an object. The host sends information about synced objects to peers. This is one-way transmission; peers sync with what is happening on the host. As synced objects are created, moved and destroyed on the host, they are correspondingly created, moved and destroyed on all connected peers. It is important to disable any behaviors and

deactivate any events on the peers that may attempt to move the objects themselves; this will conflict with what *Sync object* is trying to do, and will not have any effect on the host. Peers should use their client input values as their sole way of influencing the game.
Synced objects can optionally include their position and/or angle with a given precision. If no position or angle is selected, then it simply ensures the same numbers of objects are created.

> *Syncing without a position, or with only one axis, causes objects to be created at a co-ordinate of -1000, as there is not full information about the synced object's position being sent. In that case its up to your project to correctly position the object.*

Bandwidth can be used to reduce the number of updates it is necessary to send for a synced objects.

- *Normal bandwidth (unpredictable)* will send updates for the object at most every update (30 times a second in Internet mode) and is suitable for objects with unpredictable movement.

- *Low bandwidth (highly predictable)* will send updates at most 10 times a second, which should only be used for highly predictable motion such as moving in a straight line at the same speed (it is not enough to handle changes in motion smoothly).

- *Very low bandwidth (essentially static)* will send updates at most twice a second, which should only be used for objects which are not expected to move but nevertheless can occasionally be created or destroyed, such as scenery.

Note that even in *Normal bandwidth* mode, objects which are not changing gradually reduce their bandwidth to twice a second anyway, so static objects will still end up using *Very low bandwidth* mode. Therefore it is not normally necessary to change this, and it is suitable to use *Normal bandwidth* even for objects which rarely change.

> *Note Sync object does not support objects in containers. You should make sure any synced objects are not in a container, sync each object separately, and if necessary associate them through events.*

Sync instance variable

Add an instance variable to sync with an object. The host sends the values of the instance variable for each object to the peers, keeping them up to date. The chosen object must already first be synced with *Sync object*.
The *Precision* corresponds to the precision for *Sync object*, with an additional *Very low (uint8, 1 byte)* option, useful for bitwise flags.
Interpolation can be *None* (updates in steps), *linear* (linearly smoothed interpolation between values, suitable for positions), or *angular* (rotational interpolation between angles).
The *Client value tag* should be the name of the corresponding client input value, if any, to help

ensure the host can sync the instance variable with minimal latency.
Note: text instance variables cannot be synced, only numbers. To share text data between peers, use messages instead.

Broadcast message

As with *Send message*, but can only be used by the host. This sends a message to every peer in the room. *From ID* can be used to indicate the message is being sent on behalf of another peer; if it is used, when peers receive the message the *FromID* and *FromAlias* will be set to this peer. Also the message will *not* be sent to the specified *From ID* peer, since usually this is redundant. If it is empty, it will be sent to all peers and received as from the host.

Kick peer

When host, forcibly disconnect a peer from the room so they are no longer participating. The kicked peer will be notified that they have been disconnected and optionally the kick reason can be displayed. Peers cannot kick anyone, only the host can.

Enable local input prediction

Enable local input prediction on an object representing the local player. The object must be associated with the local peer's ID, and the game logic must correctly attempt to move both the local player and the player on the host in exactly the same way with correct use of client input values. This allows local controls to have immediate effect, but apply correction if the host position starts to deviate from the local position. See the fourth multiplayer tutorial for more information. There are two additional options for customizing how local input prediction is handled:

- **Avoid solids:** if enabled, input prediction will avoid moving the object into any solid. If input prediction would have moved the object so that it overlaps a solid, it instead leaves the object in its current position.
- **Platform mode:** if enabled with an object using the Platform behavior, this detects if the Platform behavior is currently standing on the floor, and when it is then input prediction on the Y axis is disabled. This can help avoid a jittery movement when landing on the floor.

The precise settings to use depend on the type of game and movement in use - some experimentation may be required to determine the best input prediction settings.

Set client state

In *On client update*, set the value of a client input value by its tag. The value must be a number, and *Add client input value* must have been used on startup to add a value with the given tag in advance.

Add ICE server

Add a custom Interactive Connectivity Establishment (ICE) server used by WebRTC to establish connections between peers. There are a couple of built-in public STUN servers used, but you can also provide your own TURN servers to enable connectivity through certain kinds of NAT. A username and credential can also be optionally provided if the server requires them. This action should be used on startup, before any connections are made.

Auto-join room

Join the first available room with the given game, instance and first room name. The player must be connected and logged in to the signalling server. The first player to join a room becomes the host. When rooms are full, the signalling server will create a new room. For example if "myroom" is full, it will try "myroom2", "myroom3", etc. This effectively arranges all joining peers in to games of a particular size. If the room is locked when full, then late-joiners are not allowed; if left unlocked and a peer leaves after the game starts, a newly joining peer may be added back to the game to top it up to the *Max peers* again. Upon joining, *On joined room* triggers.

Connect

Connect to a signalling server. The official Scirra signalling server is at **wss://multiplayer.construct.net**. Upon successful connection, *On connected* will trigger.

Disconnect

Disconnect from the signalling server. This can be done once peer-to-peer connections are established if the signalling server is no longer necessary, but note that will prevent any new peers from joining the game late.

Join room

Join a specific room in the given game instance. The player must be connected and logged in to the signalling server. The first player to join a room becomes the host. *Max peers* can be used to limit the number of peers that join. Only the host's value is used. If the room is full, subsequently joining peers will receive a "room full" error. The peer count includes the host, so 2 is the minimum value, or it can be left as 0 to allow an unlimited number of peers to join. Upon successfully joining, *On joined room* triggers.

Leave room

If in a room, leaves the room on the signalling server. *On left room* triggers upon the server acknowledging the request to leave. Note the room has not really been left until that trigger runs.

Log in

Once connected, log in to the signalling server. Players must log in before they can join rooms. The *Alias* is the requested display name to use. Note that if the requested alias is already taken, the server will automatically assign an alternative; be sure to use the *MyAlias* expression after logging in to determine the actual alias in use. Upon a successful login, *On logged in* triggers.

Request game instance list

Request a list of active game instances within the given game. When the response is received *On game instance list* triggers and the name and total number of peers in the returned instances can be listed using the *List...* expressions.

Request room list

Request a list of active rooms within a given game instance. The returned list can include all rooms, only rooms which are unlocked, or only rooms which are available to join (unlocked and not full). When the requested list is received, *On room list* triggers.

Multiplayer expressions

Filename

MIMEType

In a binary transfer, the filename and MIME type specified by the sender. These expressions are available on the receiving end in both the *receive started* and *completed* triggers.

TransferProgress

During a binary transfer and in a *progress* event (fired for both senders and receivers), this returns the current transfer progress on a scale of 0 to 1 (e.g. 0.5 being 50% complete).

TransferSize

In a binary transfer, the size in bytes of the data to be transmitted. This is set on the receiving end in the *receive started* event so the receiver knows ahead of time how big the data will be. (Once the transfer has completed, the size can also be obtained from the chosen Binary Data object.)

ListInstanceCount

After *On game instance list* triggers, the number of game instances in the received list.

ListInstanceName(index)

ListInstancePeerCount(index)

Get the name and peer count of a given game instance in the returned instance list.

ListRoomCount

After *On room list*, the number of rooms in the received list.

ListRoomName(index)

ListRoomPeerCount(index)

ListRoomMaxPeerCount(index)

ListRoomState(index)

After *On room list*, retrieve information for a room at an index in the received list. The state can be one of "available", "locked" or "full".

FromAlias

FromID

The alias and ID of the peer a message is from in *On message received* or *On any message received*.

HostAlias

HostID

When in a room, the alias and ID of the host of the room.

LeaveReason

A string identifying a reason for leaving in *On peer disconnected*, if known, e.g. "quit", "timeout", "network error"...

Message

The contents of the received message in *On message received* or *On any message received*.

PeerAlias

PeerID

The alias and ID of the relevant peer in a trigger like *On peer connected* or *On peer disconnected*.

PeerCount

The number of peers in the current room, including the host.

PeerAliasAt(index)

PeerIDAt(index)

The alias and ID of the nth peer in the current room, up to *PeerCount*.

PeerAliasFromID(peerid)

Get the alias of a peer in the current room from their peer ID.

PeerIDFromAlias(alias)

Get the ID of a peer in the current room by their alias.

PeerLatency(peerid)

PeerPDV(peerid)

Get the latency and packet delay variation (PDV) of a peer from their peer ID. Peers can only use this to get the stats for the host, since that is the only connection they have, but the host can use it for any peer.

Tag

The tag of the received message in *On any message received*. This expression is also used by binary transfers, returning the tag of the current binary transfer in any of the binary transfer triggers.

LagCompensateAngle(movingPeerID, fromPeerID)

LagCompensateX(movingPeerID, fromPeerID)

LagCompensateY(movingPeerID, fromPeerID)

Return the lag-compensated position and angle for *movingPeerID* as seen by *fromPeerID*. In other words, this returns the past position of *movingPeerID* going back by the amount of time that *fromPeerID* is delayed by, given their latency. For example this can be used to perform a lag-compensated hit-test when *fromPeerID* shoots a laser. This is covered in more detail in the fourth multiplayer tutorial.

PeerState(peerid, tag)

When host, retrieve the latest client state value with the given tag, for a given peer ID. The peer will have set this with the *Set client state* action to indicate their input state.

CurrentGame

CurrentInstance

CurrentRoom

Retrieve the current game, instance and room names, if joined on the signalling server.

ErrorMessage

In *On signalling error*, the error message if available.

MyAlias

MyID

The current player's own alias and ID, once connected and logged in to the signalling server.

SignallingMOTD

SignallingName

SignallingOperator

SignallingURL

SignallingVersion

Once connected to the signalling server, retrieve the Message Of The Day (MOTD), server name, server operator, website URL and server version for the connected server.

ClientXError

ClientYError

The input prediction error for peers, used for debugging.

HostX

HostY

The position the host has for the current peer, used for debugging.

StatInboundBandwidth

StatOutboundBandwidth

Return the total estimated inbound and outbound bandwidth transmitted over the network, in bytes per second. When automatic data compression is in use, this measures the compressed size of the data actually sent over the network.

StatInboundDecompressedBandwidth

StatOutboundDecompressedBandwidth

Return the total estimated decompressed inbound and outbound bandwidth for all data transmission through the Multiplayer object, in bytes per second. When automatic data compression is in use, this measures the size of the decompressed messages, which may be significantly larger than the data actually sent over the network, measured by the *StatInboundBandwidth* and *StatOutboundBandwidth* expressions. Together these expressions

can also be used to identify the compression ratio (i.e. how much bandwidth is saved by compression).

StatInboundCount

StatOutboundCount

Return the total number of separate inbound and outbound messages sent and received by the Multiplayer object. This includes internally-used messages for things like ping and synchronisation; generally the bandwidth is the more practically useful statistic.