

A Solution to P2P Gaming Online in Godot

Introduction

To enable two computers behind NAT routers to communicate in a P2P fashion requires an intermediary “handshake” server running on a public IP (i.e. not behind a NAT router). Computers wishing to set up a P2P connection first communicate with the server, either registering their information or joining another computer that has registered their information. For more information about this problem, and the methodology generally employed to overcome it, see <http://www.brynosaurus.com/pub/net/p2pnat>.

The solution described by this documentation is made up of a public python server script and a Godot script. The public server script requires Python 3.5 or higher (might work with lower versions) and the *twisted* python library installed. It is called “godot-server.py” and can be run inside a local network- even on the same computer as one of the peers- but doing so precludes people from outside the network joining a game. Run the script as you would any other python script (the only configuration you might want to do is change the single constant at the top of the file- the SERVER_PORT). The Godot script should be made an auto-loaded singleton (presuming you want the connections to remain across scenes).

Constants

The interface to the Godot script consists of a set of functions and a set of signals which you can call / connect to in any scene. There are, however, a few constants you can change in the script:

- `secs_between_peer_checks` There is no standard for the amount of time before a NAT router will forget you sent a packet out to an address and begin rejecting incoming packets *from* that address.
- `secs_reg_valid` This is the amount of time the handshake server will keep your registration details before forgetting you.
- `secs_between_reg_refresh` Heartbeats are sent to the server just as they are sent to other peers- sending a refresh to the handshake server also resets the `secs_reg_valid`.
- `secs_to_await_reply` This is the amount of time to wait for replies to packets sent. Some packet don't require a reply, but some do- such as peer check packets. After this amount of time the attempt to send the packet will be counted as a failure and the packet will be resent (if it is a packet that has that attribute)
- `attempts_before_expiration` This is the amount of times a packet can fail before expiring completely. A packet expiring has different consequences depending on the packet type and depending on whether the player is a client or server.
- `_HS_SERVER_NAME` This is the peer name used by players to store the handshake server's details. No other peer can be named the same.

Generally, you should leave this constants as they are.

Functions

```
/*
 * This can be called on a server or a client.
 * After calling this on, the player will reject
 * any packets from the handshake server
 * (though a server will remain registered until it
 * times out). From that point, the connection is
 * self-sustaining P2P.
 */
drop_connection_with_handshake_server()

/*
 * Has no effect unless called by a server. Drops a peer from
 * a session.
 *     peer_name: <string> name of the peer recipient
 */
drop_peer(peer_name)

/*
 * Returns an array of the names of the peers for this session.
 * If there is no current session, returns an empty array.
 */
get_peers()

/*
 * Returns the address of the peer that is the server for this session.
 * The format is [<string> ip, <int> port]
 * If there is no current session, return Null
 */
get_server_address()

/*
 * Returns the name of the peer that is the server for this session.
 * If there is no current session, return Null
 */
get_server_name()

/*
 * Returns the user name this player.
 * If there is no current session, return Null
 */
get_user_name()

/*
 * Returns a Boolean: true if this player is the server for
 * the current session
 * If there is no current session, return Null
 */
i_am_server()
```

```
/*
 * Initialises this player as a client. Will get details of a
 * registered server from a handshake server.
 *     handshake_address: [<string> ip, <int> port]
 *                       address of the handshake server
 *     local_address:    [<string> ip, <int> port]
 *                       local address of the player. If you know the
 *                       the other peers are outside the local network,
 *                       you can just out 0.0.0.0 for this.
 *     user_name: <string> this is the user name for this player
 *     server_name: <string> this is the name of the server you
 *                   want to join.
 *     password: <string> Set this to join password-protected
 *                   server players.
 *                   You will be rejected if the password is not a match
 */
init_client(handshake_address, local_address, user_name,
server_name, password=null)

/*
 * Initialises this player as a server. Will register with
 * a handshake server and await other players.
 *     handshake_address: [<string> ip, <int> port]
 *                       address of the handshake server
 *     local_address:    [<string> ip, <int> port]
 *                       local address of the player. If you know the
 *                       the other peers are outside the local network,
 *                       you can just out 0.0.0.0 for this.
 *     server_name: <string> this is the user name for this player
 *                   and the server name under which they will be
 *                   registered on the handshake server.
 *     password: <string> Set this if you want to be password-protected.
 *                   Only players that give the same password when they
 *                   join will be accepted.
 */
init_server(handshake_address, local_address, server_name, password=null)

/*
 * Returns Boolean: true if there is a current session
 * happening.
 */
is_connected()

/*
 * Ends the current session.
 */
quit_connection()

/*
 * Asks a handshake server for a list of registered servers.
 *     handshake_address: [<string> ip, <int> port]
 *                       address of the handshake server
 */
request_server_list(handshake_address)
```

```
/*
 * Sends a message to a connected peer and awaits confirmation- resending
 * if no confirmation arrives. After a certain number of attempts, the
 * message will expire and the appropriate signal will be emitted.
 *     peer_name: <string> name of the peer recipient
 *     message: can be anything that can be a member of a
 *               JSON-serialisable dictionary (including being
 *               a dictionary itself, etc.)
 */
send_reliable_message_to_peer(peer_name, message)

/*
 * Sends a message to a connected peer with a send-and-forget attitude.
 *     peer_name: <string> name of the peer recipient
 *     message: can be anything that can be a member of a
 *               JSON-serialisable dictionary (including being
 *               a dictionary itself, etc.)
 */
send_unreliable_message_to_peer(peer_name, message)
```

Signals

```
/*
 * Emitted on the server player when a client has joined.
 *   new_peer: <string> name of the new peer
 */
signal client_confirmed

/*
 * Emitted when a player has connected successfully
 * with a server player.
 *   server_address: [<string> ip, <int> port]
 *                   address of the server player
 */
signal confirmed_as_client

/*
 * Emitted when a player has registered successfully
 * with a handshake server.
 *   handshake_address: [<string> ip, <int> port]
 *                      address of the handshake server
 */
signal confirmed_as_server

/*
 * Emitted on all kinds of errors.
 *   message: <string> info on the error
 */
signal error

/*
 * Emitted when a packet has been blocked
 *   sender_address: [<string> ip, <int> port]
 *                  address of the sender
 */
signal packet_blocked

/*
 * Emitted when a packet has been received
 *   packet_data: <dictionary> Copy of the packet data
 */
signal packet_received

/*
 * Emitted when a packet has been sent
 *   packet_data: <dictionary> Copy of the packet data
 */
signal packet_sent
```

```
/*
 * Emitted when a peer fails to connect to another
 * player. If this is a server player, nothing happens.
 * If it is someone joining a server, their session will end.
 *     peer_name: <string> name of the peer
 */
signal peer_connection_failed

/*
 * Emitted when a change in peers (addition or removal)
 * occurs in the network.
 *     peer_names: <array> names of current peers
 */
signal peer_list_updated

/*
 * Emitted when a peer-check has failed. If this is a
 * server player, the peer will be dropped. If it is
 * not a server player, the session will end.
 *     peer_name: <string> name of the peer
 */
signal peer_timeout

/*
 * Emitted when a reliable message has been received.
 *     packet_data: <dictionary> Copy of the packet data
 *                 A subset of this data is {
 *                 from: <string> The name of the sender
 *                 to: <string> will be the player's name
 *                 message: Will be the message- which can
 *                     be anything.
 *                 message-id: <int> Unique per sender
 *                 }
 */
signal received_reliable_message_from_peer

/*
 * Emitted when a handshake server has responded with
 * a list of registered servers.
 *     server_names: <array> names of registered servers
 */
signal received_server_list

/*
 * Emitted when an unreliable message has been received.
 *     packet_data: <dictionary> Copy of the packet data
 *                 A subset of this data is {
 *                 from: <string> The name of the sender
 *                 to: <string> will be the player's name
 *                 message: Will be the message- which can
 *                     be anything.
 *                 }
 */
signal received_unreliable_message_from_peer
```

```
/*
 * Emitted when a reliable message has failed to send (or, at
 * least, failed to be confirmed by the receiver)
 *   packet_data: <dictionary> Copy of the packet data
 *               A subset of this data is {
 *               from: <string> The name of the sender
 *               to: <string> will be the player's name
 *               message: Will be the message- which can
 *                   be anything.
 *               message-id: <int> Unique per sender
 *               }
 */
signal reliable_message_timeout
```

```
/*
 * Emitted when a session has been terminated
 */
signal session_terminated
```