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).

You can do away with the handshake server if everyone is on the same network. Registering as a server host with the local address and handshake server address equal will result in that player also acting as a handshake server. You may still want to use the python script server- one possible reason is that server hosts become easier to manage as players on the network increase.

Security

It is important to note that the security offered by this solution extends as far as password authorisation. The password is added to the packet data and that data is hashed, the resulting hash is checked on the other side and if they don't match the packet is blocked. This way, the passwords never travel over the network. However, the hash does not provide integrity very well because it does not hash all the data, and it does not provide confidentiality because everything is passed in clear text. As a result, at times, the local network address of the player is sometimes passed as clear text into the internet (you can avoid this by passing a dummy local IP like '0.0.0.0' when you know the handshake server is outside the local network).

There are other measures, such as checking the sender address matches an existing peer's when possible, etc., which restrict communication at the application level quite well, but this is a solution for games, and low-sensitivity game data.

Also note that reliable messages are reliable by delivery, not by order. Each reliable message does have an ascending `message-id` (edge-case: on overflow it cycles back to 0), which could be used to reconstruct ordering, but no such services are inbuilt. Also note that after packets timeout a number of times they do expire- there is a signal for expired reliable messages.

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 probably leave this constants as they are.

Functions

Note: addresses are `[<string> ip, <int> port]`

`drop_connection_with_handshake_server()`

This can be called on a server or a client. After calling this, the player will reject any packets from the handshake server. From that point, the connection is self-sustaining P2P.

`drop_peer(peer_name)`

Only valid when called by a server. Drops a peer from a session.

`get_handshake_server_address() returns <address>`

Only valid when called by a server. Returns the address of the current handshake server. If there is no current connection, returns null.

`get_password() returns <string>`

Returns the password for this session, or null if there is no current session.

`get_peer_info(peer_name) returns <info>`

Returns a dictionary of info about a peer. There are two keys: “name” which holds a string, and “address” which holds an address. If there is no current session or no peer with the given peer name, this function returns null.

`get_peers() returns [peer_names]`

Returns an array of the names of the peers for this session. If there is no current session, returns an empty array.

`get_server_address() returns <address>`

Returns the address of the peer that is the server for this session. If there is no current session, returns null.

`get_server_name() returns <string>`

Returns the name of the peer that is the server for this session. If there is no current session, returns null.

`get_user_name() returns <string>`

Returns the user name of this player. If there is no current session, returns null.

`i_am_server() returns <Boolean>`

Returns true if this player is the server for the current session. If there is no current session, returns null.

`init_client(handshake_address, local_address, user_name, server_name, password=null)`

Initialises this player as a peer client. Details of a registered server under the given server name will be retrieved from the handshake server. The password defaults to the server’s name (as it does on the peer server’s side).

`init_server(handshake_address, local_address, server_name, password=null)`

Initialises this player as a peer server. Details of this host will be registered with the handshake server. If the handshake address is the same as the local address, this host will act as a handshake server, registering with itself (note that no other peer server can register with this setup, and this only works if everybody is on the same network). The password defaults to the server's name (as it does on the peer client's side).

`is_connected() returns <Boolean>`

Returns true if this peer server is at least past registration, or this peer client has successfully joined with a peer server.

`quit_connection()`

Ends the current session, removing all pending packets, blocking any incoming packets, and resetting all user data.

`request_server_list(handshake_address)`

Sends a packet to the handshake address asking for a list of registered servers. There does not need to be current session. The `received_server_list` signal is emitted when a response comes in.

`send_reliable_message_to_peer(peer_name, message)`

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, apart from the regular timeout and expiry signals, the `reliable_message_timeout` signal will be emitted. The message can be anything that can be a member of a JSON-serialisable dictionary.

`send_unreliable_message_to_peer(peer_name, message)`

Sends a message to a connected peer with a send-and-forget attitude. The message can be anything that can be a member of a JSON-serialisable dictionary.

Signals

Note: addresses are `[<string> ip, <int> port]`
Note: `packet_data` is a dictionary with a few guaranteed fields:

- **`'__destination-address'`**
- **`'__destination-name'`**
- **`'__sender-name'`**

`signal confirmed_as_client(server_address)`

Emitted on a player client when they have connected successfully to a player peer.

`signal confirmed_as_server(handshake_server_address)`

Emitted when a player server has registered successfully with a handshake server.

`signal error(message)`

Emitted on all kinds of messages with a string error message.

`signal packet_blocked(sender_address)`

Emitted when a packet has been blocked.

`signal packet_expired(packet_data)`

Emitted when a packet has expired (i.e. a packet that awaits confirmation did not receive that confirmation and has run out of timeouts.

`signal packet_received(packet_data)`

Emitted when a packet has been received.

`signal packet_sent(packet_data)`

Emitted when a packet has been sent.

`signal packet_timeout(packet_data, will_resend)`

Emitted when a packet has timed out (i.e. a packet that awaits confirmation did not receive that confirmation). `will_resend` is a Boolean that is true if the packet has retries remaining.

`signal peer_confirmed_reliable_message_received(packet_data)`

Emitted upon peer confirmation of having received a reliable message. There are at least three keys: “from” is the name of the sender (i.e. you), “to” is the name of the recipient, and “message” is the original message sent by you.

`signal peer_dropped(peer_name)`

Emitted when a peer is dropped from the P2P network.

`signal peer_joined(peer_name)`

Emitted when a new peer is found on the P2P network.

`signal received_reliable_message_from_peer(packet_data)`

Emitted when a reliable message has been received. There are at least four keys: “from” is the name of the sender, “to” is the name of the recipient (i.e. you), “message” is the message, and “message-id” is a unique ascending (edge case is on overflow, the id allocator resets to 0) integer per peer. Note that there is no guarantee the packets will *arrive* in the order they were sent.

`signal received_server_list(servers, handshake_address)`

Emitted when a handshake server has responded with registered servers as an array of dictionaries, each with two entries: the “name” of the peer and “password-is-required”, which is true if the password for the peer server is not the default. The handshake address is the address from which this information came.

`signal received_unreliable_message_from_peer`

Emitted when an unreliable message has been received. There are at least three keys: “from” is the name of the sender, “to” is the name of the recipient (i.e. you), and “message” is the message.

`signal reliable_message_timeout`

Emitted when a reliable message has failed to send (or, at least, failed to be confirmed by the receiver). There are at least four keys: “from” is the name of the sender (i.e. you), “to” is the name of the intended recipient, “message” is the message, and “message-id” is the unique ascending (edge case is on overflow, the id allocator resets to 0) integer per peer.

`signal session_terminated`

Emitted when a session has been terminated.