Roum Documentation

Release 0.1.0dev

Roum Inc.

Feb 24, 2019

Contents:

CHAPTER 1

Roum Multiplayer Protocol Specification

This section of the documentation describes the protocol used to communicate between the client and server.

The protocol is based on a single TCP Connection per client. This allows for great compatibility and connection security, due to TCPs automatic packet validation and re-sending.

1.1 Protocol Specification Index

1.1.1 Abstract

The Roum Multiplayer Protocol is designed to let multiple clients share game state information via a single central server.

The protocol is designed in a way that allows users to easily host their own servers on any semi-powerful computer. The standalone server does not need any graphical libraries and can thus run on headless server machines.

Central Server Design

A central server design was chosen for security reasons, because only the central server has to be trusted. This means that all player data is stored on the server. If any data is to shared with other players, it must first go through the central server. One of the disadvantages of this design is however that scaling to very large (100+) players on a single server becomes difficult without very powerful and thus expensive server machines.

Security Considerations

The protocol specification is designed to be secure in a way that tampering with connections or stealing the identity of clients or servers should be nearly impossible.

Additionally, since the protocol will usually be implemented in Python, common attacks used on C/C++ based architectures are not possible. This assumes that Python itself does not have any unknown security vulnerabilities.

1.1.2 Low-Level Implementation

Framework

The actual handling of sending and receiving packets is done via `peng3dnet`. This module was developed specifically for use with `peng3d`, which is the graphics framework used for rendering.

The `peng3dnet` module also includes extensions that allow for simple and (somewhat) secure pinging and location syncing.

On the Wire

The actual format of packets including headers is described in the `peng3dnet` documentation.

The payloads sent are encoded using MessagePack due to its various advantages, including low overhead in both speed and serialized data size.

In this documentation, all specifications will only describe the contents of the payload as if it were encoded as a Python `dict`. This is done mainly to simplify the documentation, as all the low-level encoding is done by `peng3dnet` and not needed for a good understanding of the protocol itself.

1.1.3 `ping` Connection Mode

The pinging mechanism is handled by the `peng3dnet.ext.ping` module.

This mode is also different in that it actually is a different `peng3dnet` connection type called `peng3dnet.ext.ping.PingConnectionType`, whereas all other connection modes use the same connection type and can be switched between at any time.

Purpose

A ping is intended to be used to query publicly available information from the server. This information may include the number of players on the server, the name of the server and many other pieces of information.

A ping is usually sent by the game client, but it may also be sent by other applications to gather information about a server, e.g. in a website presenting a list of servers.

As a side-effect, the roundtrip time from client to server and back is also measured. This ping time automatically includes packet encoding, network lag, packet decoding/re-encoding by the server and handling by the client. The roundtrip time is thus an accurate representation of the delays to be expected when querying information from the server during other modes.

Client Side

Pinging is done by the multiplayer server list found when clicking the *Select Server* button in the main menu.

The multiplayer server list uses the `opendig.client.Client.asyncPingServer()` wrapper to ensure that the returned server data has been normalized

---

Todo: How does this work for Roum

See also:

See the /format/serverlist documentation for how the server list is stored.

Server Side

Since ping requests are automatically handled, only a callback that can add server statistics is used for customization.

The callback that is used is `opendig.server.dedicated.ODServer.getPingData()`.

Todo: How does this work for Roum

Transmitted Data

Todo: Add this subsection

1.1.4 `auth` Connection Mode

Every connection that is not a *ping* connection will start in this mode. Once this connection mode has been changed away from, it cannot be reentered, due to security reasons.

Connections in this mode are considered to be not authenticated and not trustworthy.

The main purpose of this connection mode is to exchange credentials to mutually verify the identity of both server and client.

Handshaking

These procedures need to be followed to ensure that every connection is secure and identity theft is prevented.

Note that I am not a security expert, so please point out any security vulnerabilities privately. Contact information can be found on my GitHub Profile @not-na.

On First Server Start

On the very first server start, the server registers itself to the authserver calling the `roumauth.ServerSession.createServer()` method after creating its own instance of `roumauth.ServerSession()`.

Note that when first creating the server object, only `serverdata` needs to be supplied, as the other values will be automatically generated.

This call populates the `password` and `serverid` attributes. These values should be stored securely, as they define the identity of the server. The `serverid` attribute will be shared with clients and is not secret, while the `password` attribute should be kept as secret as a private key.

The `password` defaults to a random string of length 32.

On Every Server Start

This routine should be done on every server start.

First, the credentials generated in the last subsection should be loaded again and used to create a `roumauth.ServerSession()`.

If needed, the `serverdata` may be updated to reflect changes in the configuration.

The ServerSession should be kept around for as long as the server is running.

On Every New Connection (Server Preparation)

Whenever a client requests a new connection, the session validity should be checked and renewed. This can be done by calling the `roumauth.ServerSession.refreshToken()` method.

Once this has concluded, the authentication may continue.

Further Protocol

First, the client uses the API to create a new `roumauth.Session()` with the user-supplied credentials. The E-Mail address used to login should be stored locally to speed up further login attempts.

Then, the client opens a new `peng3dnet` connection and waits for the `peng3dnet` handshake to finish before proceeding.

The client then sends a request to the server asking for the server ID and waits for the answer.

This server ID is used by the client to create a MPSession ID on the authserver. The authserver checks the server ID for validity and returns a MPSession ID and secret.

The MPSession ID is then sent to the server for further processing.

The server checks the MPSession ID with the authserver, also setting a flag that the server has verified the connection. It also receives the client ID it did not know before and the secret, which are all sent to the client in order to prove the identity of the server.

Then, the client sends the MPSession ID and secret to the authserver to verify that the server has set the verification flag. This also sets the client verification flag.

The client also sends a go-ahead packet to the server to signal that it is ready.

The server then checks with the authserver that the client verification flag has been set. If the flag has been set, the server indicates to the client that the connection has been successfully verified.

After this security handshake has concluded, both parties can mutually trust their supposed identities.

Internally, numbers from 1 to 12 are used to describe the stage of the handshake.

The full table is represented here:

No.	Direction	Name	Description
1-2	—	Handshake	Peng3dnet handshake
3.	Client->Server	ServerID #1	Request for serverID
4.	Server->Client	ServerID #2	Sends serverID
5.	Client->Auth	Create MPSession	Creates MPSession with serverID and clientID, receives sessionID
6.	Client->Server	Send MPSession	Sends MPSession token, shared between parties
7.	Server->Auth	Check Session	Validates Token, sets servercheck, receives secret
8.	Server->Client	Verify	Sends secret and clientID to client for verification
9.	Client->Auth	Check Session	Checks secret and servercheck, and sets clientcheck
10.	Client->Server	Ready	Indicates to server that it is ready
11.	Server->Auth	Verify	Checks clientcheck, receives client information
12.	Server->Client	Ready	Indicates to client that it is ready

Note: After the results of stage 11 have been collected by the authserver, the MPSession is deleted. This makes it impossible to verify a connection after it has been established.

`roumauth` Client Library

The `roumauth` library can be used to easily implement the authentication interface.

Use the `roumauth.Session.connectToServer()` or `roumauth.ServerSession.connectToClient()` methods to create a connector object that will handle authentication.

After this object has been created, you will first need to set the callback function via `roumauth._Connector.setCallback()`.

For further information, see the `roumauth._Connector()` class documentation.

Packets

1.1.5 `match` Connection Mode

Purpose

After leaving the lobby, all data transmissions concerning the game itself (all non-chat data) will use this mode.

Packets

`roum:match.init` - Initialize a new match

`roum:match.init`

This packet is sent by the server when the client has been added to a match.

Internal Name	*roum:match.init*
Direction	Clientbound
Since Version	v0.1.0dev
Valid Modes	`lobby` only

Purpose

This packet signals the client to initialize its datastructures for a new match. Usually, this packet will only be sent if the player is in the queue, but it may also be sent at any other time.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.:

```
{
   "uuid":"443b47701bfc44c3a38a22e943c087f5",

   "players_a": ["1a0ab7f4322542ea9d62052c874e25e7", ...],
   "players_b": ["7082a2f1f09a4a4fa264281628db179a", ...],

   "mode": "classic",
}
```

`uuid` is the UUID of the match. It is generated by the server and shared by all clients.

`players_a` and `players_b` are lists containing the UUIDs of all players part of the match. The two lists represent the two different factions.

`mode` is the game mode this match is in. Currently, only `classic` is supported.

`roum:match.status.update` - Update the Game Status

`roum:match.status.update`

This packet is used to update the game status.

Internal Name	*roum:match.status.update*
Direction	Cliendbound
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is intended to update major information concerning the game status, like the begin and the end of an match.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.:

```
{
"time":0

"status":{...}
}
```

The `time` is the timestamp in *game ticks*, at which the `status` was updated. Here, the `time` is the serverside time.

`status` Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.

Todo: Specify the attributes of `roum.Match`

`roum:match.synchronise.time` - Synchronising Game Ticks

`roum:match.synchronise.time`

This packet is used to synchronise the clientside and serverside *game ticks*.

Internal Name	*`roum:match.synchronise.time`*
Direction	Clientbound
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is intended to synchronise the inferior clientside *game ticks* to the superior serverside game ticks. This synchronisation occurs each second (60 game ticks) and is sent to all players.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.:

```
{
"time":8345
}
```

The `time` is the serverside time in *game ticks*.

`roum:match.player.update.pos` - Update Player Position

`roum:match.player.update.pos`

This packet is used to pass the position of a player to each other player.

Internal Name	*roum:match.player.update.pos*
Direction	both
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is intended to be sent to all players in the vicinity of the emitting player each *game tick*. If the recipient is further away, he will receive this packet less frequent.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.:

```
{
"time":9162

"player":"1a0ab7f4-3225-42ea-9d62-052c874e25e7"
"position":(531.25, 133.39, 927.78)
}
```

The `time` is the timestamp in *game ticks*, at which the `player` was at the specified `position`. Here, the `time` is the clientside time of the `player`, not the serverside game time.

`roum:match.entity.update.pos` - Update Entity Position

`roum:match.entity.update.pos`

This packet is used to pass the position of an entity to each other player.

Internal Name	*roum:match.entity.update.pos*
Direction	Clientbound
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is intended to be sent to all players in the vicinity of the emitting entity each *game tick*. If the recipient is further away, he will receive this packet less frequent.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.:

```
{
"time":13465

"entity":"1ba7070a-22a7-4a0f-9f3b-df91f4ea7b65"
"position":(331.29, 971.01, 566.98)
}
```

The `time` is the timestamp in *game ticks*, at which the `entity` was at the specified `position`. Here, the `time` is the serverside game time.

`roum:match.player.update` - Update Player Status

`roum:match.entity.update.pos`

This packet is used to pass the status of a player to each other player.

Internal Name	`roum:match.player.update`
Direction	both
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is intended to update the player status every time it changed. It will be sent to all players concerned by the change.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.

The structure of this packet is similar to /format/player_data but does not contain all keys

Todo: Specify the attributes of `roum.Player`

`roum:match.player.takedamage` - Signalise the player that he took damage

`roum:match.player.takedamage`

This packet is used to tell a player that he took damage.

Internal Name	*`roum:match.player.takedamage`*
Direction	Clientbound
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is intended to inform a player that he took damage. It will be sent each time he takes damage.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.:

```
{
"time":8376

"player":"1a0ab7f4-3225-42ea-9d62-052c874e25e7"
"origin":"7082a2f1-f09a-4a4f-a264-281628db179a"
"components":{...}
}
```

The `time` is the timestamp in *game ticks*, at which the `player` took damage. The `origin` is the source which damaged the `player`. If the source was another player or one of his projectiles, the player's uuid will be transmitted. If the player was damaged by an passive object, the objects uuid will be transmitted. Here, the `time` is the serverside time.

`components` Structure

`components` is a mapping of all the ships components damaged.

Structure of `component` values::

```
{
"shield":{"amount":203.15, "type":"laser"},
"thruster_1":{"amount":15.93, "type":"laser"},
...
}
```

Todo: Add list for components (And how do they even work?)

`amount` indicates the amount of damage the specified component took. `type` indicates the type of the damage.

`roum:match.entity.create` - Create new Entity

`roum:match.entity.create`

This packet is used to create a new entity.

Internal Name	*roum:match.entity.create*
Direction	Clientbound
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is intended to signal the new creation of a new entity. This might either be a new launched drone or as well an object like an asteroid splitting in half or many ship wreckage debris after an explosion. Differently to the *roum:match.entity.update*, this packet is sent to all players and not only to those in the vicinity.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.

The structure of this packet is similar to /format/entity_data but does not contain all keys

Todo: Specify the attributes of `roum.Entity`

`roum:match.entity.update` - Update Entity

`roum:match.entity.update`

This packet is used to update the status of entities.

Internal Name	*roum:match.entity.update*
Direction	Clientbound
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is intended to pass the status of an entity to all players in the vicinity. It is always emitted when an attribute of the entity changes. Players to far away to be impacted by the status change will only receive an update flag that will trigger the necessary updates as soon as they get close enough to the entity.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.

The structure of this packet is similar to /format/entity_data but does not contain all keys

Todo: Specify the attributes of `roum.Entity`

`roum:match.entity.takedamage` - Signalise an entity took damage

`roum:match.entity.takedamage`

This packet is used to tell a player that an entity took damage.

Internal Name	*roum:match.entity.takedamage*
Direction	Clientbound
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is intended to signalise to a player that a certain entity took damage. This can be beneficial to inform the player that for example one of his drones or an allied weapon factory is being attacked.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.:

```
{
"time":8374

"entity":"94e28628-b83d-498a-8dce-8e2cbc8aff3f"
"origin":"0dd967e3-d418-4815-b347-feecb1d1eb6d"
"components":{...}
}
```

The `time` is the timestamp in *game ticks*, at which the `entity` took damage. The `origin` is the source which damaged the `entity`. If the source was another player or one of his projectiles, the player's uuid will be transmitted. If the player was damaged by an passive object, the objects uuid will be transmitted. Here, the `time` is the serverside time.

`components` Structure

`components` is a mapping of all the entities components damaged.

Structure of `component` values::

```
{
"warhead":{"amount":24.13, "type":"ion"},
"body":{"amount":96.67, "type":"ion"},
...
}
```

Todo: Add list for components (And how do they even work?)

`amount` indicates the amount of damage the specified component took. `type` indicates the type of the damage.

`roum:match.entity.remove` - Removes an Entity

`roum:match.entity.remove`

This packet is used to remove an entity.

Internal Name	*`roum:match.entity.remove`*
Direction	Clientbound
Since Version	v0.1.0dev
Valid Modes	`match` only

Purpose

This packet is used to inform players of the removal of an entity. Such a case might occur when a weapon factory is destroyed or when a projectile impacts another entity.

Structure

Note that all examples shown here contain placeholder data and will have different content in actual packets.:

```
{
"time":3780

"entity":"76874380-20f7-42e3-aefb-46a8dca98d03"
}
```

The `time` is the timestamp in *game ticks*, at which the `entity` was removed. Here, the `time` is the serverside game time.

1.1.6 Packets

`auth` Connection Mode

`lobby` Connection Mode

`match` Connection Mode

`chat` Connection Mode

CHAPTER 2

roum - Roum Core Package

CHAPTER 3

Glossary

game tick Smallest unit of time in-game. Always represented as an integer. Usually, one game tick is equivalent to one-sixtieth of a second.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Index

G

R