Cardgame Documentation

Release 0.1.0

guenthobald, notna

May 25, 2022

Contents:

CHAPTER 1

CardGame Multiplayer Protocol Specification

TODO

1.1 Protocol Specification Index

1.1.1 Cards

TODO

1.1.2 Doppelkopf: Rules

TODO

1.1.3 Doppelkopf: Penalties

TODO

1.1.4 `auth` - Authentication Packets

TODO

Packets

`cg:auth` - Authentication Packet

`cg:auth`

This packet is used to perform log-in and sign-up activities.

Internal Name	*cg:auth*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`auth` only

Purpose

Using this packet, the client can authenticate itself with the server as a specific account. It can also create new accounts using this packet.

Structure

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

This is the data sent by the client to the server:

```
{
   "username": "notna",
   "pwd": "...",

   "create": false,
}
```

`username` is the user name of the account to login as or create.

`pwd` is the encrypted password of the given account.

Warning: Currently, passwords aren't actually encrypted in transmission. In the future, all traffic will be either tunneled through SSL or an asymmetric cipher will be used to transmit passwords.

`create` is an optional flag indicating if the client wishes to create a new account. If it is not given, it will be assumed as `false`. This flag exists to prevent accidental account creation should a user mistype their username.

The server will respond with a packet of the same type and the following data:

```
{
   "status": "logged_in",

   "username": "notna",
   "uuid": "cfde3788-e653-4ef3-8b19-f741e2194e0f",
}
```

`status` is the current authentication status. It should be one of `logged_in`, `wrong_credentials`, `user_exists` or `logged_out`.

`username` is the user name the user is logged in as. This field is only sent if `status` is `logged_in` or `user_exists`.

`uuid` is the *UUID* of the current account. It can be used to look up further information in the *user database*. It is only present if `status` is `logged_in`.

If the login attempt was successful, the server will already pre-send a *cg:status.user* packet with information on the user. It will also send a *cg:status.server.mainscreen* packet to update the client on the contents of the main screen. Also, the connection mode will change to `active`.

See also:

See the *cg:status.user* packet for more information on how to get User data.

See also:

See the *cg:auth.precheck* packet for more information on the authentication process.

`cg:auth.precheck` - Authentication Precheck Packet

`cg:auth.precheck`

This packet is used to pre-check a login attempt.

Internal Name	*cg:auth.precheck*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`auth` only

Purpose

Using this packet, the client can check if the account name actually exists and fetch an encryption key to be used when sending the password.

Structure

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

This is the data sent by the client to the server:

```
{
   "username": "notna",
}
```

`username` is just the user name entered by the user on the prompt displayed by the client.

The server responds with a packet like this:

```
{
   "username": "notna",
   "valid": true,
   "exists": true,

   "key": "...",
}
```

`username` is the same name as was sent by the client, but normalized according to the server. Usually this involves lower-casing the user name.

`valid` is a boolean flag that determines whether or not the username is valid on this server. This does not mean that it exists, just that it could exist.

`exists` is a boolean flag showing whether the account exists or not. This can be used by the client to ask the user if they want to create a new account.

`key` is a binary key to be used to encrypt the password before sending it to the server. It is specific to the connection, user name and will expire after some time. If the key is the empty string, no encryption should be applied.

See also:

See the *`cg:auth`* packet for further information on password exchange.

1.1.5 `status` - Status Packets

TODO

Packets

`cg:status.user` - User Status Update

`cg:status.user`

This packet is used to request and retrieve user information and status updates.

Internal Name	*`cg:status.user`*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	All States

Purpose

Using this packet, the client can request information about a specific user from the server. The server determines what information to send.

Additionally, the server may send this packet at any to preempt information requests or notify the client of changes to a users appearance.

Structure

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

This is the data sent by the client to the server to request information on a user:

```
{
   "username": "notna",

   "uuid": "61cf5d06-8d01-4fb3-a4a8-ea7a0633b0b8",
}
```

`username` is the name of the user that the client wants information on.

`uuid` is the *UUID* that the client wants more information on.

Note: `uuid` and `username` are not exclusive, but `uuid` will be used preferentially before `username`.

The server sends user status updates in the following format, either as a response to a request or as a notification:

```
{
   "username": "notna",
   "uuid": "cfde3788-e653-4ef3-8b19-f741e2194e0f",

   "status": "logged_in",
   ...
}
```

`status` is the current status of the user. This may be one of `online`, `away`, `busy`, `offline` or `notexist` if the user could not be found.

Note: If `status` is `notexist`, all other fields will not be populated.

`username` is the user name to be displayed for the given user.

`uuid` is the *UUID* of the given user.

Todo: Add more user attributes here.

`cg:status.message` - Status Messages for Clients

`cg:status.message`

This packet is used by the server to show different notices, warnings and errors on the client.

Internal Name	*cg:status.message*
Direction	Clientbound
Since Version	v0.1.0
Valid States	All States

Purpose

Using this packet, the server can cause the client to show warnings and other messages to the user.

Structure

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

This is an example that the server could send to the client:

```
{
   "type": "notice",

   "msg": "Hello World!",
}
```

`type` is the type of status message and determines the imagery used in the dialog on the client. Currently, there are the following types: `notice`, `warning` and `error`.

`msg` is the raw message. Currently, no formatting is supported, but this may change in the future.

Note: Long messages may be cut short by the client, depending on the window size and `type`.

`cg:status.server.mainscreen` - Status Updates for the main screen

`cg:status.server.mainscreen`

This packet is sent by the server to let the client know about the contents of the main screen.

Internal Name	*`cg:status.message`*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`active` only

Purpose

Using this packet, the server can update the client on information shown on the main screen.

Structure

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

This is an example that the server could send to the client:

```
{
    ...
}
```

Todo: Find something to transmit here…

See also:

See the *`cg:status.user`* packet for more information about how user profile data is sent to the client.

1.1.6 `party` - Party Management Packets

TODO

Packets

`cg:party.create` - Party creation

`cg:lobby.ready`

This packet is used to create a *party*.

Internal Name	`cg:party.create`
Direction	Serverbound
Since Version	v0.1.0
Valid States	`active` only

Purpose

This packet is used by a client to create a *party*. Afterwards, the client will be automatically joined the party using the *`cg:party.join`* packet.

Structure

The package ought not to contain any data.

See also:

See the *`cg:party.join`* for further information on joining a party.

`cg:party.join` - Join party

`cg:party.join`

This packet is used to join a *party*.

Internal Name	*`cg:party.join`*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`active` only

Purpose

After the creation of a *party*, the creator will be joined automatically to the party. Additionally, if another client accepts an invitation to a party, he will be joined.

Upon joining, the server will send a *`cg:party.change`* packet to the other clients in the party containing the updated user list. The joining client will receive a similar packet which however will contain all the information on the party.

Structure

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

This is the data sent by the server to the client:

```
{
  "party":"397627fa-2aa3-4cef-b403-7658bb8b424d",
}
```

`party` is the party's *UUID*.

See also:

See the `cg:party.create` packet for further information on how a party is created.

`cg:party.invite` - Invite client to party

`cg:party.invite`

This packet is used to invite other clients to a *party*.

Internal Name	*cg:party.invite*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`active` and `lobby`

Purpose

This packet is used to invite other clients to a *party*. It transmits the username of the invited user to the server and afterwards tells the inviter whether the client exists. Additionally, it informs the invited client on the invitation.

Upon accepting the invitation, the server will receive a *cg:party.invite.accept* packet from the invited client.

Structure

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

This is the data sent by the server to the client:

```
{
   "username":"notna",
}
```

`username` is the name of the invited user.

The server will answer to the inviting client with this:

```
{
   "user_found":True,
}
```

`user_found` is a boolean informing the inviter whether the invited user has been found.

Additionally, it will send following data to the invited client:

```
{
   "inviter":"e2639d1f-a7b3-409f-87e4-595a85444d30 ",
}
```

`inviter` is the *UUID* of the inviting user.

See also:

See the *cg:party.invite.accept* packet for further information on accepting an invitation.

`cg:party.invite.accept` - Accept invitation to party

`cg:party.invite.accept`

This packet is used to accept the invitation to a *party*.

Internal Name	*cg:party.invite.accept*
Direction	Serverbound
Since Version	v0.1.0
Valid States	`active` only

Purpose

Upon being invited to a *party*, this packet is used to inform the server on whether the client has accepted or denied the invitation.

If it accepts the invitation, the client will receive a *cg:party.join* packet. Furthermore, the inviter will be informed via a *cg:status.message* packet, if the invited client accepted the invitation.

Structure

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

This is the data sent by the server to the client:

```
{
   "accepted":true,
}
```

`accepted` is a boolean declaring whether the invitation has been accepted.

See also:

See the *cg:party.invite* packet for further information on inviting to parties.

See also:

See the *cg:party.join* packet for further information on joining a party.

`cg:party.change` - Party data change

`cg:party.change`

This packet is used by the server to inform the client on any kind of change in a *party*.

Internal Name	*cg:party.change*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`active` and `lobby`

Purpose

This packet is used to inform all the clients in a *party* about any kind of change. This mostly will be a client joining or leaving the party.

Structure

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

This is the data sent by the server to the client:

```
{
   "userlist":["e2639d1f-a7b3-409f-87e4-595a85444d30", "e2639d1f-a7b3-409f-87e4-
↪595a85444d30"],
}
```

`userlist` is a list containing the *UUID*s of the users in the *party*.

`cg:party.leave` - Leave party

`cg:party.leave`

This packet is used to leave a *party*.

Internal Name	*cg:party.leave*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`active` and `lobby`

Purpose

This packet is used to leave a *party*, may it be by the clients own decision or for it being kicked. The server will also confirm the client that it has left the party.

Subsequently, the server will send a *cg:party.change* packet to all remaining clients in the lobby containing an updated user list.

Structure

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

The serverbound packet doesn't contain any data.

Note: If the client is kicked out of the party, the packet will only be clientbound since the client didn't choose itself to leave the party.

The server will send following to the client:

```
{
   "party":"397627fa-2aa3-4cef-b403-7658bb8b424d",
}
```

`party` is the *UUID* of the party that was left.

See also:

See the `cg:party.kick` for further information on kicking a user out of a party.

`cg:party.kick` - Kick user from party

`cg:party.invite`

This packet is used to kick another client from a *party*.

Internal Name	*`cg:lobby.kick`*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`active` and `lobby`

Purpose

This packet is used to kick a client from a *party*. It also allows the kicker to name a reason for why the other client has been kicked.

This client will receive a *`cg:status.message`* packet informing it on the reason. Subsequently, the server will send it a *`cg:party.leave`* packet.

Structure

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

This is the data sent by the server to the client:

```
{
   "username":"notna",
   "reason":"Pressed Alt-F4 to turn up the volume",
}
```

`username` is the name user that ought to be kicked.

`reason` is the justification for the kick.

See also:

See the *`cg:party.leave`* packet for further information on leaving a party.

1.1.7 `lobby` - Lobby Management Packets

TODO

Packets

`cg:lobby.create` - Create lobby

`cg:lobby.create`

This packet is used to create a *lobby*.

Internal Name	*cg:lobby.create*
Direction	Serverbound
Since Version	v0.1.0
Valid States	`active` only

Purpose

Using this packet, the server is notified of the creation of a *lobby*, either because a *custom game* was created by a client, or because the *matchmaking* matched enough players together to start a game.

Upon creating the lobby, the creator and his *party* members will be joined using a *cg:lobby.join* packet.

Structure

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

This is the data sent by the server to the client:

```
{
   "game": "doppelkopf",
   "variant": "c",
}
```

`game` may be a string declaring the type of game the lobby creator wants to play. This field is optional.

`variant` may be a string declaring the variant of `game` that the lobby creator wants to play. Available variants differ from game to game. This field is required if `game` is given.

See also:

See the *cg:lobby.join* packet for further information on the response of the server.

cg:lobby.join - Join lobby

cg:lobby.join

This packet is used to join a *lobby*.

Internal Name	*cg:lobby.join*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`active` only

Purpose

After the creation of a *lobby*, the creator and all his *party* members will be joined automatically. Additionally, any client accepting an invitation will receive this packet.

Upon joining, the server will send a *cg:lobby.change* packet to the other clients in the lobby containing the updated user list. The joining client will receive a similar packet which however will contain all the information on the lobby.

Structure

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

This is the data sent by the server to the client:

```
{
  "lobby":"397627fa-2aa3-4cef-b403-7658bb8b424d",
}
```

`lobby` is the lobby's *UUID*.

See also:

See the *cg:lobby.create* packet for further information on how a lobby is created.

`cg:lobby.invite` - Invite client to lobby

`cg:lobby.invite`

This packet is used to invite other clients to a *lobby*.

Internal Name	*cg:lobby.invite*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`active` and `lobby`

Purpose

This packet is used to invite other clients to a *lobby*. It transmits the username of the invited user to the server and afterwards tells the inviter whether the client exists. Additionally, it informs the invited client on the invitation.

Upon accepting the invitation, the server will receive a *cg:lobby.invite.accept* packet from the invited client.

Structure

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

This is the data sent by the client to the server:

```
{
   "username":"notna",
}
```

`username` is the name of the invited user.

The server will send following data to the invited client:

```
{
   "inviter":"e2639d1f-a7b3-409f-87e4-595a85444d30",
   "lobby_id":"g2639d1f-a7b3-409f-87e4-595a85444d30"
}
```

`inviter` is the *UUID* of the inviting user.

`lobby_id` is the *UUID* of the lobby the user was invited to.

See also:

See the *cg:lobby.invite.accept* packet for further information on accepting an invitation.

cg:lobby.invite.accept - Accept invitation to lobby

cg:lobby.invite.accept

This packet is used to accept the invitation to a *lobby*.

Internal Name	*cg:lobby.invite.accept*
Direction	Serverbound
Since Version	v0.1.0
Valid States	`active` only

Purpose

Upon being invited to a *lobby*, this packet is used to inform the server on whether the client has accepted or denied the invitation.

If it accepts the invitation, the client will receive a *cg:lobby.join* packet.

Structure

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

This is the data sent by the server to the client:

```
{
   "accepted":true,
   "inviter":"d2639d1f-a7b3-409f-87e4-595a85444d30"
   "lobby_id":"e2639d1f-a7b3-409f-87e4-595a85444d30",
}
```

`accepted` is a boolean declaring whether the invitation has been accepted.

`inviter` is the *UUID* of inviting user.

`lobby_id` is the *UUID* of the lobby the user was invited to.

See also:

See the *cg:lobby.invite* packet for further information on inviting to lobbies.

See also:

See the *cg:lobby.join* packet for further information on joining a lobby.

cg:lobby.change - Lobby data change

cg:lobby.change

This packet is used by the server to inform the client on any kind of change in a *lobby*.

Internal Name	`cg:lobby.change`
Direction	Clientbound
Since Version	v0.1.0
Valid States	`lobby` only

Purpose

This packet is used to inform all the clients in a *lobby* about any kind of change. This might be a client joining or leaving the lobby, the choice of game or its rules being changed, players signalising their readiness, and more.

Structure

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

This is the data sent by the server to the client:

```
{
   "users":{
      "e2639d1f-a7b3-409f-87e4-595a85444d30": {"ready": true, "role": 1},
      "e70d98cd-a33b-41f2-9cb4-8c6e3aeadbb7": {"ready": false, "role": 2},
      },
   "game":"doppelkopf",
   "gamerules":{
               "fuechse":true,
               "feigheit":true,
               "armut":false,
            },
   "gamerule_validators":{
               ...
   },
   "supported_bots": ["dk_dumb", "dk_smart"],
}
```

`userlist` is a dictionary mapping the *UUID*s of players to their metadata. This metadata currently contains the `ready` and `role` keys. All players must have their `ready` flag set to true to begin the game. `role` determines what the player can do. If the `role` is `-1`, the player should be removed.

`user_order` is the order of the users for being shown in the lobby and for determining the seat order in the game.

`game` is the name of the game that will be played.

`gamerules` are the rules by which the game will be played. Note that only updated rules will be sent.

`gamerule_validators` is a dictionary containing the validators for the current game.

`supported_bots` is a list of supported *bots* names.

Todo: Document the validator concept

Note: All the parameters are optional. However, they should be all sent upon joining so the client knows what information to show.

Note: The keywords for the different `gamerules` will change depending on the `game`. Also, multiple of the games being of german origin, many rules will have german names. All gamerule names should be ASCII only for maximum compatibility. This does not however apply to the displayed translated names.

`cg:lobby.leave` - Leave lobby

`cg:lobby.leave`

This packet is used to leave a *lobby*.

Internal Name	*`cg:lobby.leave`*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`lobby` only

Purpose

This packet is used to leave a *lobby*, may it be by the clients own decision or for it being kicked. The server will also confirm the client that it has left the lobby.

Subsequently, the server will send a *`cg:lobby.change`* packet to all remaining clients in the lobby containing an updated user list.

Structure

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

The serverbound packet doesn't contain any data.

Note: If the client is kicked out of the lobby, the packet will only be clientbound since the client didn't choose itself to leave the lobby.

The server will send following to the client:

```
{
   "lobby":"397627fa-2aa3-4cef-b403-7658bb8b424d",
}
```

`lobby` is the *UUID* of the lobby that was left.

See also:

See the *`cg:lobby.kick`* for further information on kicking a user out of a lobby.

`cg:lobby.kick` - Kick user from lobby

`cg:lobby.kick`

This packet is used to kick another client from a *lobby*.

Internal Name	*cg:lobby.kick*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`lobby` only

Purpose

This packet is used to kick a client from a *lobby*. It also allows the kicker to name a reason for why the other client has been kicked.

This client will receive a *cg:status.message* packet informing it on the reason. Subsequently, the server will send it a *cg:lobby.leave* packet.

Structure

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

This is the data sent by the server to the client:

```
{
   "uuid":"dc71e5dd-5d4b-4809-8546-068e2628f115",
   "reason":"Pressed Alt-F4 to turn up the volume",
}
```

`uuid` is the *UUID* of the user that ought to be kicked.

`reason` is the justification for the kick.

See also:

See the *cg:lobby.leave* packet for further information on leaving a lobby.

`cg:lobby.ready` - Lobby readiness conveyance

`cg:lobby.ready`

This packet is used by a client to signalise it is ready to begin the game.

Internal Name	*cg:lobby.ready*
Direction	Serverbound
Since Version	v0.1.0
Valid States	`lobby` only

Purpose

This packet is used by a client to signalise it is ready to begin the game. When all clients in a *lobby* conveyed their readiness, the game begins.

When the server receives this packet, it will send a *cg:lobby.change* packet to all clients in the lobby containing the updated list of ready players.

Structure

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

This is the data sent by the client to the server:

```
{
   "ready": true,
}
```

`ready` defines whether or not the client is ready to play.

1.1.8 `game` - Main Game Packets

TODO

Packets

`cg:game.start` - Start game

`cg:game.start`

This packet is used to start the game.

Internal Name	*cg:game.start*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`lobby` and `active`

Purpose

This packet is used to start the game, either when all clients in a lobby conveyed their readiness or when a client reconnects to the server after exiting from a running game. Upon receiving this packet, as well as all the card creation packets, the client will send this packet back to the server so that it knows, when all the players are ready and the cards can be dealt

Structure

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

This is the data sent by the server:

```
{
   "game_type": "doppelkopf",
   "game_id": 'e613d0cc-1021-46fb-8403-c2b66663cfb6',
   "player_list": [
      'd5b445bf-8836-4fec-a4a8-a219f6df073e',
      '08e6b252-6f24-4d0f-9d77-be926461874a',
      '9267bb0e-619c-41c6-a3d1-ef7d574ccbdd',
      '9765882f-5763-4373-93a5-f8fd0c643018',
   ],
}
```

`game_type` is the type of the game (`skat`, `doppelkopf`, `rummy` or `canasta`).

`game_id` is the *UUID* of the game.

`player_list` is a list of the *UUID*s of the players in the game, in the same order as in the server's game object

The client will send and empty packet to the server.

`cg:game.end` - End game

`cg:game.end`

This packet is used to end the game.

Internal Name	*`cg:game.end`*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`game_*`

Purpose

This packet is used to end the game, either when the predefined amount of rounds has been reached or when all players decide to exit the game early.

Structure

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

This is the data sent by the server:

```
{
   "next_state": "results"
}
```

`next_state` can either be `results` if the game has been ended properly, or `lobby` if it was ended abruptly.

`cg:game.load` - Load game

`cg:game.load`

This packet is used to load a game upon continuing an old game.

Internal Name	*`cg:game.load`*
Direction	Serverbound
Since Version	v0.1.0
Valid States	`lobby`

Purpose

This packet is used, when a player in a lobby loads an old game. It conveys the game data to the server so that the server can load this game. The other clients in the lobby will only receive the game data with the *cg:game.start* packet

Structure

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

This is the data sent to the server:

```
{
   "game_id": "e8d1e1e2-75c8-4225-ab1a-16dabcc260d1",
   "data": {
      "id": "e8d1e1e2-75c8-4225-ab1a-16dabcc260d1"
      "type": "dk",
      "creation_time": 1591004154.1594243,
      "players": [
         "acb8fa68-5c22-42cc-a4fa-1ba600dcdb9e", "c4db1dfe-9d6c-41c2-9a88-ea7c641738a6
↪",
         "d940a7e4-c19a-4904-abcf-71aab689da11", "ac5085ad-148d-4838-b800-dba3c6a5c91c
↪"
      ],
      "gamerules": {
         "dk.etc": ["and", "so", "on"]
      },
      "round_num": 3,
      "buckrounds": [],
      "scores": [[-3, 3, 3, -3], [2, 2, 2, -6], [5, -5, 5, -5]],
      "current_points": [4, 0, 10, -14],
      "game_summaries": [
         ["re_win", "re"],
         ["kontra_win", "no90"],
         ["kontra_win", "kontra", "no90", "against_cqs"]
      ]
   }
}
```

`game_id` is the game's *UUID*.

`data` is a dictionary containing the data of the saved game. It should contain following keys:

`id`: see `game_id`

`type` The game type. It can be `dk` (Doppelkopf), `sk` (Skat), `cn` (Canasta) and `rm` (Rummy).

`creation_time` is the system time at which the game was created.

`players` is a list of the *UUID*s of the players.

`gamerules` is a dictionary containing the game's gamerules.

`round_num` is the amount of rounds, that have already been played.

`buckrounds` is a list of the buckrounds, that still have to be played. Its exact structure depends on the buckround gamerules.

`scores` is a list containing lists for each round. In these lists, the scores for the round are saved.

`current_points` is a list containing the current scores for the players.

`game_summaries` is a list containing the game summaries for all rounds.

`cg:game.save` - Save game

`cg:game.save`

This packet is used to save a game for the purpose of continuing it later.

Internal Name	*cg:game.save*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`game_*`

Purpose

This packet is used, when all players decided to adjourn the game. The server will send the clients in the game this packet containing the game data, that should be saved locally for being able to load it when continuing the game.

Structure

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

This is the data sent by the server:

```
{
   "game_id": "e8d1e1e2-75c8-4225-ab1a-16dabcc260d1",
   "data": {
      "id": "e8d1e1e2-75c8-4225-ab1a-16dabcc260d1"
      "type": "dk",
      "creation_time": 1591004154.1594243,
      "players": [
         "acb8fa68-5c22-42cc-a4fa-1ba600dcdb9e", "c4db1dfe-9d6c-41c2-9a88-ea7c641738a6
↪",
         "d940a7e4-c19a-4904-abcf-71aab689da11", "ac5085ad-148d-4838-b800-dba3c6a5c91c
↪"
      ],
      "gamerules": {
         "dk.etc": ["and", "so", "on"]
      },
      "round_num": 3,
      "buckrounds": [],
      "scores": [[-3, 3, 3, -3], [2, 2, 2, -6], [5, -5, 5, -5]],
      "current_points": [4, 0, 10, -14],
      "game_summaries": [
         ["re_win", "re"],
         ["kontra_win", "no90"],
         ["kontra_win", "kontra", "no90", "against_cqs"]
      ]
   }
}
```

`game_id` is the game's *UUID*.

`data` is a dictionary containing the data of the saved game. It should contain following keys:

`id`: see `game_id`

`type` The game type. It can be `dk` (Doppelkopf), `sk` (Skat), `cn` (Canasta) and `rm` (Rummy).

`creation_time` is the system time at which the game was created.

`players` is a list of the *UUID*s of the players.

`gamerules` is a dictionary containing the game's gamerules.

`round_num` is the amount of rounds, that have already been played.

`buckrounds` is a list of the buckrounds, that still have to be played. Its exact structure depends on the buckround gamerules.

`scores` is a list containing lists for each round. In these lists, the scores for the round are saved.

`curront_points` is a list containing the current scores for the players.

`game_summaries` is a list containing the game summaries for all rounds.

`cg:game.dk.question` - Request an answer from a client

`cg:game.dk.question`

This packet is used to request an answer from a player. It is only used for the game *Doppelkopf*.

Internal Name	*cg:game.dk.question*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`game_dk` only

Purpose

Using this packet, the server can ask the client on its "opinion" on something. A question packet will be answered by the client with a *cg:game.dk.announce* packet. This packet is only available for the game *Doppelkopf*.

It will be used to ask all players about a *reservation* at the begin of each round. In the course of this, the concerned players will be inquired after *solos*, *throwing*, *pigs*, *superpigs*, *poverty* and *wedding*. In the cases of a *wedding* or a *poverty*, the choice of the trick or of the cards to exchange are requested by this packet.

See also:

See *Doppelkopf: Rules* for further information on special rules.

In case of an accusation concerning an external misconduct, e.g. originating from a chat, this packet will be used to ask all the players if they support the accusation.

See also:

See the *cg:game.dk.complaint* packet for further information on accusations.

Structure

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

This is the data sent by the server to the client:

```
{
   "type":"reservation",
   "target":"296f8f9f-40dc-4ef7-b9b5-851d58c9c966",
}
```

`type` is the type of request sent.

Note: Following types are available: `reservation`, `solo`, `throw`, `pigs`, `superpigs`, `poverty`, `poverty_accept`, `poverty_trump_choice`, `poverty_return_trumps`, `poverty_return_choice`, `wedding`, `wedding_clarification_trick`, `black_sow_solo` and `accusation_vote`.

`target` is the *UUID* of the player to whom the question is directed. This is necessary because sometimes all players are supposed to hear a question, though it might not be directed at all of them.

See also:

See the *`cg:game.dk.announce`* packet for further information on announcements.

`cg:game.dk.announce` - Make an announcement

`cg:game.dk.announce`

This packet is used to announce something. It is only used for the game *Doppelkopf*.

Internal Name	*`cg:game.dk.announce`*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`game_dk` only

Purpose

Using this packet, a player can make an announcement. This announcement will be sent to all the players. This packet is only available for the game *Doppelkopf*.

This packet will be used to answer to a *reservation*, *solo*, *throwing*, *pigs*, *superpigs*, *poverty* and *wedding*. In case of a *wedding*, it will transfer the clarification trick and in case of a *poverty*, it will be used to tell the amount of returned trumps. During the course of the game, it will be used to announce *Re* and *Kontra* as well as denials like *No 90* etc. Furthermore, it will be used to announce a *pig*. In case of an accusation with external misconduct it will be used to transmit the votes of the players.

See also:

See *Doppelkopf: Rules* for further information on special rules.

Structure

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

This is the data sent by the client to the server:

```
{
   "type":"poverty_return",
   "data":{"amount":3},
}
```

`type` is the context in which the announcement is made.

`data` is an optional argument transmitting further information if required.

Note: Following types are available: `reservation_yes`, `reservation_no`, `solo_yes`, `solo_no`, `throw_yes`, `throw_no`, `pigs_yes`, `pigs_no`, `superpigs_yes`, `superpigs_no`, `poverty_yes`, `poverty_no`, `poverty_accept`, `poverty_decline`, `poverty_return`, `wedding_yes`, `wedding_no`, `wedding_clarification_trick`, `re`, `kontra`, `no90`, `no60`, `no30`, `black`, `pig`, `superpig`, `black_sow_solo`, `ready`, `throw`.

Note: Following types require data: `solo_yes`, `black_sow_solo`: `type` (the type of the solo), `poverty_return`: `amount` (the amount of trumps returned to the poverty player, `wedding_clarification_trick`: `trick` (the trick the bride wishes to determine the re party), `no90`, `no60`, `no30` and `black`: `party` (Optional, the party of the announcing player, but only, if it wasn't known yet.)

The server conveys following data to all the clients:

```
{
   "announcer":"453b1c0c-4742-4ba7-9d42-6f4acec1856a",
   "type":"pig",
}
```

`announcer` is the *UUID* of the player who made the announcement.

`type` and `data` are similar to arguments the server received.

`cg:game.dk.card.intent` - Do something with a card

`cg:game.dk.card.intent`

This packet is used to do something with a card. It is only used for the game *Doppelkopf*.

Internal Name	*cg:game.dk.card.intent*
Direction	Serverbound
Since Version	v0.1.0
Valid States	`game_dk` only

Purpose

Using this packet, a player can perform an action with a card. Usually this is playing the card. Subsequently, the server will send a `game.dk.card.transfer` packet to all clients. This packet is only available for the game

Doppelkopf.

In case of a *poverty*, this packet will be used to choose the cards that should be exchanged. Otherwise, it's used to play a card over the course of the game.

See also:

See *Doppelkopf: Rules* for further information on special rules.

Structure

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

This is the data sent by the client to the server:

```
{
   "intent":"play",
   "card":"91eb5e2c-b7e8-4d8a-b865-7e9eaf2e6469",
}
```

`intent` is the action that the player wants performed. It can be `play`, `pass_card` or `return_card`.

`card` is the *UUID* of the card the player wants to use for the given intent. If an intent requires multiple cards, this field may be a list.

See also:

See the `game.dk.card.transfer` for further information on how a card is moved from one slot to another.

`cg:game.dk.card.transfer` - Transfer a card

`cg:game.dk.card.transfer`

This packet is used to transfer a card from one *slot* to another one. It is only used for the game *Doppelkopf*.

Internal Name	*`cg:game.dk.card.transfer`*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`game_dk` only

Purpose

Using this packet, the server can signalise the client that a card was transferred to another *slot*. This packet is only available for the game *Doppelkopf*.

This may be used for dealing the cards, where the cards will be moved from the shuffled deck to the hands of the players. It will also be used when a player plays a card; the card will be transferred from the player's hand to the table. Furthermore, after all the players played their card, the four cards on the table will be moved to the trick stack of the player who won the trick. Moreover, if the rule *Armut* is active, upon declaring an *Armut*, this packet will be used for exchanging three cards from the concerned players.

See also:

See *Doppelkopf: Rules* for further information on special rules.

Note: To minimise the possibilities to cheat, the packet will only transmit the value of the card if the client is intended to know about it. Otherwise, the client will only be informed on the transfer of an unknown card.

Structure

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

This is the data sent by the server to the client:

```
{
   "card_id":"91eb5e2c-b7e8-4d8a-b865-7e9eaf2e6469",
   "card_value":"cq",
   "from_slot":"hand2",
   "to_slot":"table",
}
```

`card_id` is the *UUID* of the transferred card.

`card_value` is the value of the card. If the client should not know about the card value, an empty string will be transmitted.

`from_slot` is the *slot* in which the card was before the transfer. If this is `None`, the card is to be created. If this field is not `None` and the `card_id` does not exist, the client should crash with an appropriate error message.

`to_slot` is the slot to which the card will be transferred.

Note: Following slots are available: `stack`, `hand0` to `hand3`, `poverty`, `table`, `tricks0` to `tricks3`

`cg:game.dk.complaint` - Point out a wrong move

`cg:game.dk.complaint`

This packet is used to point out a mistake another player has made. It is only used for the game *Doppelkopf*.

Internal Name	*`cg:game.dk.complaint`*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`game_dk` only

Purpose

Using this packet, a player can denounce a mistake by another player. This packet is only available for the game *Doppelkopf*.

This packet is used when a player makes a mistake by accident or deliberately and another player denounces this mistake. First, the denouncing player has to accuse, which player made the mistake and choose the type of the misconduct. In case of an accusation with *wrong card* or *wrong announcement*, he will receive a list of all the cards the accused player played and all the announcements he made. The accusing player must choose from this list, which move was illegal. In case of an accusation with *played early*, the server will check whether the last card of the accused player was played before it was his turn. If the accusation proves to be wrong or if the accusing player decides to

cancel the accusation, he will receive a penalty himself. Otherwise, the accused player will be punished and the game might be aborted, depending on the penalty settings.

The mistake can also emanate from a chat or voice chat. Since the server cannot automatically arbitrate such a complaint, the two other players have to confirm it using a *cg:game.dk.question* and a *cg:game.dk.announce* packet. If 3 of the 4 players back the accusation, the punishment will be undergone by the accused, otherwise by the accuser.

Note: If the punished player ought to receive demerit points, the *cg:game.dk.scoreboard* will be used.

See also:

See *Doppelkopf: Penalties* for further information on penalty settings.

Structure

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

This is the data sent by the client to the server:

```
{
   "accused":"e421c337-70f6-409a-bdcf-acf1b3c3c6e0",
   "type":"wrong_announcement",
}
```

`accused` is the *UUID* of the accused player.

`type` is the misconduct the accused is charged with.

Note: Type can have following arguments: `wrong_card`, `wrong_announcement`, `played_early`, `external`

In case of an accusation with `wrong_card` or `wrong_announcement`, the server will reply like this:

```
{
   "moves":{
      0:{
         "type":"announcement",
         "data":"reservation_no",
      },
      4:{
         "type":"announcement",
         "data":"kontra",
      },
      5:{
         "type":"card",
         "data":"cq",
      },
      ...
   },
   "accused": "e421c337-70f6-409a-bdcf-acf1b3c3c6e0",
   "type": "wrong_announcement",
}
```

`moves` is a dictionary containing all the moves the player has done so far. Each move is represented by its move-ID, beginning in each round with 0 and counting up for each announcement made and each card played. The ID is followed by a dictionary declaring its `type` (`announcement`, `card` or `accusation`) and `data` specifying the kind of the announcement or the value of the card.

Note: Only the accuser will receive the `moves` field. All other clients will still get all other fields, however.

The client will respond with the following data:

```
{
   "accused":"e421c337-70f6-409a-bdcf-acf1b3c3c6e0",
   "type":"wrong_announcement",
   "move":{
      "98fd442d-4ee0-4d96-bf51-12917e36a001":{"type":"announcement", "data":"kontra"},
   },
}
```

`accused` and `type` remain the same as in the first packet.

`move` is the move representing the misconduct, stored as described above.

`cg:game.dk.turn` - Turn Update

`cg:game.dk.turn`

This packet is used to inform all players about the next turn. It is only used for the game *Doppelkopf*.

Internal Name	*cg:game.dk.turn*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`game_dk` only

Purpose

Using this packet, the server informs all clients, whose turn it is to play a card. This packet is only available for the game *Doppelkopf*.

Structure

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

This is the data sent by the server to the client:

```
{
   "current_trick":1,
   "total_tricks":12,
   "current_player":"7eb1c06d-2f66-46c7-8eef-6aa5a2ff85aa",
}
```

`current_trick` is the trick that is currently being played. The first trick is `1`, not `0`!

`total_tricks` is the amount of tricks in one game.

`current_player` is the *UUID* of the player that should play next.

`cg:game.dk.round.change` - Data update on the round

`cg:game.dk.round.change`

This packet is used to update the client's data on a round of *Doppelkopf*.

Internal Name	*cg:game.dk.round.change*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`game_dk` only

Purpose

Using this packet, the server informs the client on change in the round. This packet is only available for the game *Doppelkopf*.

It will be used to signalise the begin or the end of a round. Furthermore, it tells the client after the end of the reservations about the game type.

Structure

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

This is the data sent by the server to the client:

```
{
   "phase": "reservations",
   "player_list": [
      'd5b445bf-8836-4fec-a4a8-a219f6df073e',
      '08e6b252-6f24-4d0f-9d77-be926461874a',
      '9267bb0e-619c-41c6-a3d1-ef7d574ccbdd',
      '9765882f-5763-4373-93a5-f8fd0c643018',
   ],
   "game_type": "solo_hearts",
}
```

`phase` is the current phase of the game.

Note: `phase` can have following values: `loading`, `dealing`, `reservations`, `tricks`, `counting` and `end`

`player_list` is a list of the *UUID*s of the players in the game, in the same order as in the server's game object

`game_type` is the type of the game.

`modifiers` are modifiers like a buckround that influence the weight of the game.

`cg:game.dk.scoreboard` - Update the scoreboard

`cg:game.dk.scoreboard`

This packet is used to update points and pips. It is only used for the game *Doppelkopf*.

Internal Name	*cg:game.dk.scoreboard*
Direction	Clientbound
Since Version	v0.1.0
Valid States	`game_dk` only

Purpose

Using this packet, the server updates points and pips that players have. This packet is only available for the game *Doppelkopf*.

After each trick, the packet will convey the pips all players received. At the end of each game and in case of a penalty, the packet will convey the points all the players received.

Structure

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

This is the data sent by the server to the client:

```
{
   "player":"dabb43c0-2854-4cb8-aee0-3c3db3a54244",
   "pips":25
   "pip_change":15
   "points":-5
   "point_change":0
}
```

`player` is the *UUID* of the concerned player.

`pips` is the amount of pips the player has accumulated in this round.

`pip_change` is the amount of pips the player gained with the last trick.

`points` is the amount of points the player has accumulated in the play.

`point_change` is the amount of points the player gained with the last game.

Note: Both `point_change` and `pip_change` may be zero if nothing has changed.

1.1.9 `ping` Connection Mode

TODO

1.1.10 Packets

`cg:version.check` - Version compatibility check

`cg:version.check`

This packet is used to check client and server compatibility

Internal Name	*cg:version.check*
Direction	Bidirectional
Since Version	v0.1.0
Valid States	`versioncheck` only

Purpose

Using this packet, the compatibility between server and client is ascertained.

Structure

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

This is the data sent by the client to the server:

```
{
   "protoversion": 1,
   "semver": "0.1.0-dev",

   "flavor": "vanilla",
}
```

`protoversion` is a positive integer number that has to match exactly between all parties.

`semver` is used for display to the user and may be used in the future for more granular compatibility checks.

`flavor` is the "edition" of the client. `vanilla` indicates a standard and unmodified client. Modded versions and special versions should use different flavors. The flavor must match exactly and is case sensitive.

The server will respond with a packet of the same type and the following data:

```
{
   "compatible": true,

   "protoversion": 1,
   "semver": "0.1.0-dev",
   "flavor": "vanilla",
}
```

`compatible` indicates whether or not the client and this server are compatible with each other. If `compatible` is false, the server will end the connection immediately after sending the packet.

`protoversion`, `semver` and `flavor` are the corresponding version information from the server.

Note: Note that `protoversion` and `semver` may not appear to match to the client. This can happen if the server supports a compatibility mode for older/newer clients. The server should always report its actual version, not the emulated one.

CHAPTER 2

Glossary

Custom game A *custom game* is a game that is created by a player and not by the *matchmaking* system. After the creation, the game will be prepared in a *lobby*. In a custom game, the game as well as the rules can be chosen individually. Furthermore, users can be manually added and removed as players and as spectators.

Doppelkopf TODO

Lobby A *lobby* is a submenu used for the creation of *custom game*. In it, settings for the game can be specified and other players can be invited to join the game.

Matchmaking The *matchmaking* system is an algorithm on the server conceived to matching multiple users that want to play, together, preferably users with the same level of experience.

Party A *party* is a group of players that joined to play together, either a *custom game* or a game created by the *matchmaking* system.

Slot A *slot* is a single slot that zero or more cards can occupy. Common slots are the hands of the players, the table and the "bank".

user database The *user database* stores common information about players. It is usually accessed by the *UUID* of a user, but can also be searched by username.

UUID A *UUID* is a Universally Unique Identifier. Usually represented as a hexadecimal string, like `fe033447-68ac-41f8-a654-6fd84071ae6a`. It is used to uniquely identify users and other objects.

round A *round* is a single round of a *game*. It commonly consists of *tricks* and cannot be paused and restarted later

trick A *trick* usually consists of four cards that are played.

game A *game* is made up of *rounds*. It is fully self-contained.

bot A *bot* is a computer-controlled player that can play one or more supported games. Bots usually act autonomously and have access to the same information as a normal player.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Packet Index

C

Index