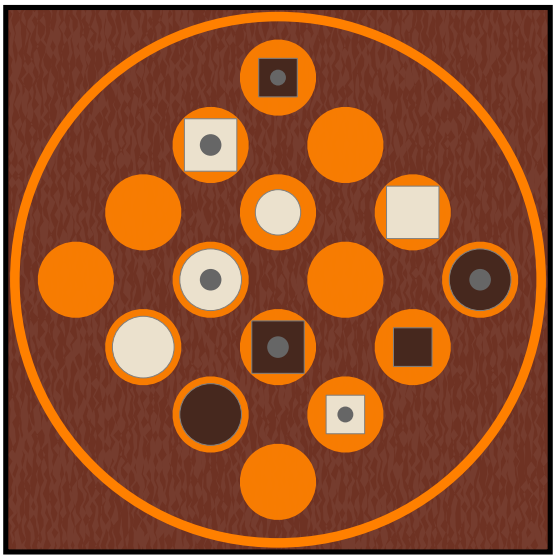
Quarto Game Protocol

TCS Module 2: Software Systems
Implementation Project 2025/2026



Version: 1.0
This document describes the communication protocol for clients and servers of the Programming Project game of 2025/2026.
If any part of the protocol is unclear or ambiguous, please send an email to Tom van Dijk (`t.vandijk@utwente.nl`) and Alexander Stekelenburg (`a.v.stekelenburg@utwente.nl`).

1 Communication Protocol

This section describes the text-based protocol used for communication between clients and the server.

1.1 Protocol format

Commands consist of a command name followed by arguments, separated by a tilde (~). A tilde is never allowed as part of an argument, unless explicitly stated otherwise. All other characters, including spaces, are allowed. Commands are terminated by a newline character (`\n`). Arguments are specified using the following notation:

- `<argument>` : required argument
- `[argument]` : optional argument
- `[argument]^n` : optional argument exactly n times
- `[argument]*` : optional argument, zero or more times
- `[argument]+` : optional argument, one or more times

1.2 List of commands

Server-side commands

- HELLO
- LOGIN
- ALREADYLOGGEDIN
- LIST
- NEWGAME
- MOVE
- GAMEOVER

Client-side commands

- HELLO
- LOGIN
- LIST
- QUEUE
- MOVE

1.3 Error handling

When a client or server receives an illegal or untimely command, it may respond with an error command. The command that triggered the error is otherwise discarded, that is, not handled by the client or server. For example, if a client sends a `MOVE` but the move is not a legal move for the current player, then an `ERROR` may be sent as it is illegal in the protocol to send an invalid `MOVE` command.

Syntax

```
ERROR[~description]
```

The description is optional and intended for debugging only. It must never be shown directly to the user, as its contents are not standardized. Receiving an `ERROR` message should always indicate a violation of the protocol and should never happen during normal operation. A server or client should never respond to an `ERROR` message with another `ERROR` message.

1.4 Initialization sequence (handshake)

Before any other commands (such as for playing games) may be exchanged, the client and server must complete a handshake. No other commands are allowed until this sequence has finished.

The handshake consists of exchanging `HELLO` messages, optionally establishing encryption, followed by the client claiming a username using `LOGIN`. The client always initiates the handshake.

HELLO (client)

Sent by the client immediately after establishing a connection.

Syntax

```
HELLO~<client description>[~extension]*
```

The client description is a short, human-readable string. Extensions may be listed in any order.

Examples

- `HELLO~Client by Alice`
  if no extensions are supported.
- `HELLO~Bob's client~CHAT~RANK`
  if `CHAT` and `RANK` are supported.
- `HELLO~Client~RANK~CHAT`
  is also legal, if the same extensions are supported (since order does not matter).

HELLO (server)

Sent by the server in response to the client's `HELLO`.

Syntax

```
HELLO~<server description>[~extension]*
```

The client description is a short, human-readable string. Extensions may be listed in any order.

Examples

- `HELLO~Server by Alice`
  if no extensions are supported.
- `HELLO~Welcome to the server made by Bob!~CHAT~RANK`
  if `CHAT` and `RANK` are supported.
- `HELLO~Server~RANK~CHAT`
  for the same list of extensions.

LOGIN (client)

Sent by the client to claim a username.

Syntax

```
LOGIN~<username>
```

If the username is already in use, the server replies with `ALREADYLOGGEDIN`. Otherwise, the server replies with `LOGIN`.

Examples

- `LOGIN~Alice`
- `LOGIN~Johnny Flodder`

LOGIN (server)

Sent by the server to confirm a successful login. This marks the end of the handshake.

Syntax

```
LOGIN
```

ALREADYLOGGEDIN (server)

Sent when the requested username is already in use. The client should try again with a different username.

Syntax

```
ALREADYLOGGEDIN
```

Note: If the server supports the `NOISE` extension and the username is already claimed using a (different) public key, then this command is also used as a reply. See the section on the `NOISE` extension for more details.

1.5 Extension handling

During the handshake, both client and server announce which protocol extensions they support. During the remainder of the connection, both parties only send commands supported by the other party. For example, chat messages are only sent to clients that support the `CHAT` extension. Similarly, a client will not send `NOISE` handshake messages if the server does not support the `NOISE` extension.

1.6 Commands after handshake

Once the handshake has completed, the commands below become available. Commands may still be untimely (e.g. starting a new game while already in one, or sending a move when it's not their turn). In such cases, the receiving party should reply with `ERROR`.

LIST (client)

Requests a list of all currently logged-in users.

Syntax

```
LIST
```

LIST (server)

Replies to `LIST` with the usernames of all connected clients.

Syntax

```
LIST[~username]*
```

The order of usernames is arbitrary.

Examples

- `LIST~Alice`
  when Alice is the only active client.
- `LIST~Charlie~Alice~Bob`
  when Alice, Bob and Charlie are all on the server.

QUEUE (client)

Indicates that the client wants to participate in a game. Sending this command again removes the client from the queue. No reply is sent by the server.

Syntax

```
QUEUE
```

NEWGAME (server)

Sent by the server to players that are placed into a new game. It is only allowed to place players into a new game that are queued. A player can only be in a single game at a time.

Syntax

```
NEWGAME~<player 1>~<player 2>
```

The first listed player makes the first move.

Examples

- `NEWGAME~Alice~Bob`
  for a new game between Alice and Bob, where Alice is the first player.

MOVE (client)

Sent by the client to indicate a move. Only allowed when it is the client's turn.

Syntax

`MOVE~<N>`	(first move only)
`MOVE~<N>~<M>`	(subsequent moves)

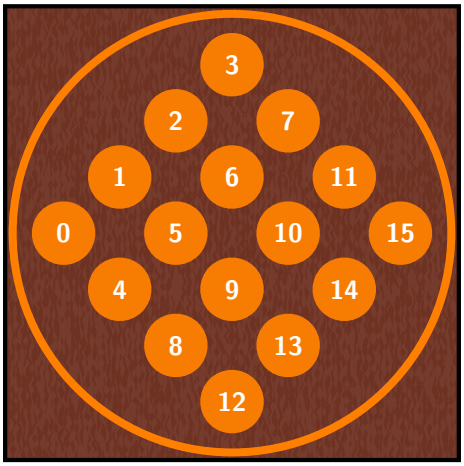
The first form of the `MOVE` command is only valid as the first move of the game.

- If `0 <= N <= 15` then `N` represents a piece given to the opponent.
- All other values of `N` are illegal.

The second form of the `MOVE` command is only valid after the first move of the game.

- If `0 <= N <= 15` then `N` denotes a square on which the given piece is placed.
- If the square is not empty, then the move is illegal.
- All other values of `N` are illegal.
- If `0 <= M <= 15` then `M` represents a piece given to the opponent.
- If the specified piece has already been placed, then the move is illegal.
- If `M == 16`, the player claims *Quarto*. Claiming *Quarto* incorrectly is not illegal, but results in an immediate loss (server will send `MOVE` followed immediately by `GAMEOVER`).
- If `M == 17`, the player places the last piece without claiming *Quarto*. This is only allowed as the final move of the game, when no pieces remain.
- All other values of `M` are illegal.

See the following image for a visualisation of the encoding of locations on the game board.



Each of the 16 pieces is uniquely encoded as an integer from 0 to 15, corresponding to the following attributes:

Code	Color	Size	Shape	Fill
0	light	small	round	solid
1	dark	small	round	solid
2	light	large	round	solid
3	dark	large	round	solid
4	light	small	square	solid
5	dark	small	square	solid
6	light	large	square	solid
7	dark	large	square	solid

Continued on next page

Code	Color	Size	Shape	Fill
8	light	small	round	hollow
9	dark	small	round	hollow
10	light	large	round	hollow
11	dark	large	round	hollow
12	light	small	square	hollow
13	dark	small	square	hollow
14	light	large	square	hollow
15	dark	large	square	hollow

While implementations may use any internal representation for pieces, i.e., using other attributes than color, size, shape, or fill, the protocol assumes that each piece is uniquely identified by a combination of four binary attributes, encoded as an integer from 0 to 15 as specified above.

MOVE (server)

Broadcast by the server to all players in the game to indicate a move. This is also sent to the player who performed the move. If the move ended the game, this command should be followed by a `GAMEOVER` command.

Syntax

`MOVE~<N>` (first move only)
`MOVE~<N>~<M>` (subsequent moves)

GAMEOVER (server)

Indicates that a game has ended. The server provides the winning player and the reason for the end of the game.

Syntax

```
GAMEOVER~<reason>[~winner]
```

Possible reasons are:

- `VICTORY`
  if the game ended with one of the players winning.
- `DRAW`
  if the game ended without a winner.
- `DISCONNECT`
  if the game ended because one player disconnected, letting the other player win.

In case of `VICTORY` or `DRAW`, the server should always first send the `MOVE` command to both players, and then the `GAMEOVER` command to both players. After receiving a `GAMEOVER` command, a client should be able to use `QUEUE` to queue for a new game.

Examples

- `GAMEOVER~VICTORY~Alice`
  for a game won by Alice.
- `GAMEOVER~DISCONNECT~Bob`
  for a game won by Bob due to a disconnect between Alice and the server.
- `GAMEOVER~DRAW`
  for a game that ended in a draw.

2 Extensions

2.1 NAMEDQUEUES extension

The `NAMEDQUEUES` extension allows a client to enter a specific queue identified by a name. Clients are only matched against other clients in the same named queue.

QUEUE (client) [modified]

Sent by the client to (un)queue for a game. With this extension, the client may optionally specify the queue name.

Syntax

```
QUEUE[~name]
```

After receiving this message, the server proceeds as follows:

- If the client is already in a game, the command is ignored.
- If no name is given, the empty string is used as the queue name. This is also the *default* queue that clients without the `NAMEDQUEUES` extension enter.
- If the client is already in the queue with the given name (possibly the empty string), then the client is removed from that queue.
- Otherwise, the client is placed in the queue with the given name and removed from any other queue. A client may be in at most one queue at a time.
- Clients are only matched for a game with other clients in the same named queue.

Furthermore:

- Queueing is only allowed after the handshake has completed (i.e., after `LOGIN`).
- To avoid a race condition with `NEWGAME`, queueing while already in a game is not allowed; therefore, if the client is already in a game, `QUEUE` is ignored.

Examples

`QUEUE`	queue in the unnamed (default) queue
`QUEUE~`	also queue in the unnamed (default) queue
`QUEUE~Group 4`	queue in the queue named `Group 4`

2.2 NOISE extension

The `NOISE` extension ensures confidentiality and authenticity of all messages exchanged between client and server. When this extension is used, an encrypted channel is established during the initialization sequence, after which all protocol messages are encrypted. Both static and ephemeral keys are exchanged. Ephemeral keys are generated on the spot and guarantee *forward secrecy*, while static keys remain the same and are used to *authenticate*.

Support for this extension is indicated by including `NOISE` in the `HELLO` messages exchanged during initialization.

Overview The `NOISE` extension is based on the Noise protocol, a framework for cryptographic protocols. See also `https://www.youtube.com/watch?v=ceGTgqypwnQ`. We use the `Noise_XX_25519_AESGCM_SHA256` instantiation, i.e., the XX handshake pattern with X25519 keys, AES-GCM ciphers, and SHA-256 hashing. The XX handshake pattern consists of three messages:

1. Initiator → Responder: the initiator's ephemeral public key (32 bytes)
2. Responder → Initiator: the responder's ephemeral public key and encrypted static public key (96 bytes)
3. Initiator → Responder: the initiator's encrypted static public key (64 bytes)

These messages also include MAC tags to prevent tampering. The optional payload field defined by the Noise specification is not used. In the `NOISE` extension, the client acts as the *initiator* and the server as the *responder*. Each handshake message is sent as a single Base64-encoded line over the network.

Implementation You are not required to implement the cryptographic primitives yourself. Instead, you should use a reference implementation of the Noise protocol. For Java, this means using the `noise-java` library, available from the GitHub repositories of `rweather` or `signalapp`, for example `https://github.com/signalapp/noise-java`. A pre-built JAR of the `signalapp` fork is available via Maven Central.

The central class in this library is `HandshakeState`. It must be initialized in `INITIATOR` mode on the client and in `RESPONDER` mode on the server.

Ephemeral keys are generated automatically by the framework. Each party must additionally provide a local static X25519 key pair. The private key is a 32-byte value that should be generated using a secure random number generator (e.g., `SecureRandom` in Java); the corresponding public key is derived automatically and is also 32 bytes long. The private key should be stored in a secure location, while the public key may be shared publicly.

To supply the static key to the `HandshakeState`, call `getLocalKeyPair()` to obtain a `DHState` object and set the private key using `setPrivateKey(...)`. After the private key has been supplied, the handshake is initiated by calling `start`. The three handshake messages are then exchanged by repeatedly invoking `writeMessage` and `readMessage`, with each message being Base64-encoded before transmission and Base64-decoded upon reception.

Once all three handshake messages have been exchanged, the `split` method is used to derive two `CipherState` instances: one for encrypting outgoing messages and one for decrypting incoming messages. Associated data is not used.

For further details, consult the Javadoc of the noise-java library.

Adapted handshake After both `HELLO` messages have been exchanged and both parties support the `NOISE` extension, the client **must** initiate encryption by sending the first Noise handshake message.

The full sequence is as follows:

1. Client sends `HELLO` advertising `NOISE` extension support
2. Server replies with `HELLO` also advertising `NOISE` extension support
3. Client sends Noise message 1
4. Server replies with Noise message 2
5. Client sends with Noise message 3
6. Client continues the handshake with the `LOGIN` command, now encrypted

Encrypted messages after the key exchange After the handshake completes and the cipher states have been derived, all further communication must be encrypted. Each plaintext protocol command is encrypted using the sender cipher, then Base64-encoded, and finally sent as a single line over the network. On reception, the message is Base64-decoded and decrypted using the receiver cipher. The decrypted message must be a valid plaintext protocol command.

Authentication When a client logs in using the `NOISE` extension, the server must associate the username with the client's public static key. Subsequent login attempts for that username must satisfy the following:

- If the client uses `NOISE` but presents a different public static key, the server responds with `WRONGKEY`.
- If the client does not use `NOISE`, the server responds with `ALREADYLOGGEDIN`.

The public static key of the remote party can be obtained from the `HandshakeState` using the `getRemotePublicKey` method.

Example *Start of initialization sequence*

- Client → Server: `HELLO~Crispy Clean Client by Charlie~NOISE~RANK`
- Server → Client: `HELLO~Sanne's Secret Server~NOISE`
- Client → Server (Noise message 1): `8Bam4p6FNMWMS1dDHhHClh3ciH5fb4DnwJhfoTbucUs=`
- Server → Client (Noise message 2): `8BmbrPdaoMr0TuuQaT1iEEXgo4IZ8saGzo9K5rqjQVnutRJFoAkvKdItXqf4LI3XGoAWTJAi++TwKmiUlYjgXIKML1NeJKN20hPWEZyaYurZsGBpZy8QkJIbnPx2/PWZ`
- Client → Server (Noise message 3): `Ev5QxO/G2I3VACi7zBkM+KA5QOfJKdUILl56b5CkRw9icWnjGOeZraMRLuFCM/rAXawcSMRyRPt2mSo/7+NMLQ==`
- Client → Server (encrypted `LOGIN`): `wDKg5tCoIvwU4FALmY2IPN9cTicromKlMFOO`
- Server → Client (encrypted `LOGIN`): `ppUxGGwWo3CyPZPjwS4CHsUzpi48`
- Client → Server (encrypted `QUEUE`): `bujMfAZ7xDpUxzj4JH0iCltuQ8+8`

WRONGKEY (server)

Sent as a reply to a `LOGIN` command when the client attempts to log in using a username that is already associated with a different public static key.

Syntax

```
WRONGKEY
```

2.3 RANK extension

The `RANK` extension allows the server to communicate the current player rankings to clients. Support for this extension is indicated by including `RANK` in the `HELLO` messages during initialization.

RANK (client)

Sent by the client to request the current player ranking on the server.

Syntax

```
RANK
```

RANK (server)

Sent by the server to communicate the current rankings to the requesting client.

Syntax

```
RANK[~username~score]*
```

All players who have played at least one game on the server should be included. The score must be a nonnegative integer (floating-point values are not allowed). Usernames must be listed in order of descending score.

Examples

```
RANK~Alice~100~Bob~0
RANK~Bob~3~Alice~0~Charlie~0
```

In the first example, the score could be a win percentage, where Alice has won all games and Bob has lost all games. In the second example, the score could simply be the number of wins, where Bob has won three games so far.

2.4 CHAT extension

The `CHAT` extension allows clients to communicate using text messages, both during and outside games. Support for this extension is indicated by including `CHAT` in the `HELLO` messages during initialization.

Important Chat messages may contain the protocol delimiter character (~). To support this, the protocol is extended with the notion of *escape characters*, which is mandatory to implement correctly for the `CHAT` extension.

Escape characters Apart from the new commands described below, the protocol is extended to support escape characters. Escape characters allow special characters to be used inside command parameters.

The escape character is the backslash (\). Within parameters:

- the delimiter character ~ is encoded as \~,
- the backslash character \ is encoded as \\.

For example, a `CHAT` command with the message

```
this is a tilde:  ~, and this is a backslash:  \
```

is encoded as:

```
CHAT~this is a tilde:  \~, and this is a backslash:  \\
```

To implement this correctly, decoding must be performed by reading received message lines character by character. Whenever a backslash is encountered, the following character is taken literally. Standard delimiter-based parsing (e.g., using `String.split`) is insufficient.

Encoding is performed by escaping parameters before transmission: every backslash is replaced by `\\` and every tilde by `\~`. Only these two characters must be escaped. You can use `String.replace` to accomplish this.

CHAT (client)

Sent by the client to broadcast a chat message to all connected clients. The server delivers the message to all clients that support the `CHAT` extension, except for the sending client.

Syntax

```
CHAT~<message>
```

Examples

```
CHAT~Hello from Enschede!
CHAT~Obviously everyone has seen a tilde before!  It's not
as if \~ is that special.
CHAT~\~
```

The first example sends a normal chat message. The second example sends a message containing a tilde. The third example sends a message that consists only of a single tilde.

CHAT (server)

Sent by the server to notify a client that it has received a message from the global chat. This command is sent to all clients that support the `CHAT` extension, except for the client that sent the original message.

Syntax

```
CHAT~<sender>~<message>
```

Examples

```
CHAT~Charlie~Is it raining for everyone else as well?
CHAT~Chuck~\~\~
```

Here, the second example is a message by Chuck containing two tildes.

WHISPER (client)

Sent by the client to deliver a private chat message to a specific client (possibly itself). If the recipient is not known to the server or does not support the `CHAT` extension, the server replies with `CANNOTWHISPER`.

Syntax

```
WHISPER~<recipient>~<message>
```

Examples

```
WHISPER~Bob~I have a feeling that Chuck will cheat...
WHISPER~Johnny Flodder~Alice is winning.  Execute plan B wi
thin \~10 seconds.
```

WHISPER (server)

Sent by the server to notify a client that it has received a private chat message.

Syntax

```
WHISPER~<sender>~<message>
```

Examples

```
WHISPER~Alice~I have a feeling that Chuck will cheat...
WHISPER~Chuck~Alice is winning.  Execute plan B within \~10
seconds.
```

CANNOTWHISPER (server)

Sent by the server to notify the client that a private message could not be delivered to the intended recipient.

Syntax

```
CANNOTWHISPER~<recipient>
```