Online Hex Client Documentation

This document contains a technical description of the Online Hex Tournament used in project 2.

If something is unclear or you can't get it to work, please reach out to us on the Blackboard forums. If you find a bug/exploit/vulnerability/etc, please send the course staff an email and we will try to patch it.

Getting started

In this section, we will walk you through the steps of getting access to the Online Hex Tournament gaming servers, integrating your code with the provided client, and playing games against remote players.

Setup and Structure

Make sure that you have python 3.X installed, that you have the requests library (`pip install requests`), that you are on the NTNU network (VPN if you're at home), and that you have downloaded the provided client files:

- `ActorClient.py` (python code for the client)
- `server.crt` (ssl certificate for authenticating the server)

For the following examples, suppose also that you are currently inside your project directory, which has the following structure.

```
|- project/
    |- my_code/
    |   |- __init__.py
    |   |- ...
    |- ActorClient.py
    |- server.crt
    |- ...
```

Where `my_code` is just a placeholder for the rest of your codebase with the Hex actors you've developed.

Note: It is important (by default at least) that the server certificate `server.crt` is stored in the same folder as `ActorClient.py`.

Getting Access

Most communication with our servers requires a personal API token for authentication. We have already created a user account for you, but if it is your first time connecting (or you lost your token), then it can be reset through your (blackboard-registered) email.

Resetting your token is a 2-step process. First, you must request a token reset, which can be done with the CLI provided in the `ActorClient.py` module.

```
python -m ActorClient request-reset-token --email <your-email>
```

This should give you a message confirming that an email has been sent to you with a **temporary** token. Check your inbox and copy the temporary token into the following command:

```
python -m ActorClient reset-token --temp-token <token-from-email>
```

This should give you a message confirming that the token reset was successful and your new token should be printed to the terminal. Save this token somewhere as you will need it later.

If you want to quickly verify that the new token works, try the following command:

```
python -m ActorClient get-profile --auth <your-token>
```

This should display information about your account, including your email, handle (nickname), qualification score, and remaining qualification attempts (more on this below).

Creating a Minimal Run Script

Now that you have credentials for accessing the gaming server, we will use the ActorClient to hook your code up with the Online Hex Tournament.

As an illustrative example, we will create a minimal script called `play_online.py`.

```python
# Import and initialize your own actor
from my_code import MyHexActor
actor = MyHexActor()

# Import and override the `handle_get_action` hook in ActorClient
from ActorClient import ActorClient
class MyClient(ActorClient):
    def handle_get_action(self, state):
        row, col = actor.get_action(state)  # Your logic
        return row, col

# Initialize and run your overridden client when the script is executed
if __name__ == '__main__':
    client = MyClient()
    client.run()
```

It imports and initializes your custom actor, overrides the `handle_get_action` method in the `ActorClient` so that it returns actions (vacant (row,col) coordinates on the board) according to your code, and runs the client whenever this script is executed. It is assumed that your Hex actor processes Hex boards of the same dimensions as the server is configured for (this will be announced).

Note: This is just a minimal example. `handle_get_action` is the most crucial method for you to override, but there are other hooks, such as for game start/end events. Please see the section "All Game Hooks" for details.

Note: Your system should be able to produce an action in a reasonably short time. To account for delays caused by network latency and server load, the configured timeout is somewhat forgiving, but to stay safe: **make sure that your system can produce a valid action in less than one second**. Failing to do so will result in a timeout and forced disconnection.

Note: Make sure that the action (row, col) returned by `handle_get_action` is json serializable. A common pitfall is to return a tuple of "numpy ints" instead of "python ints". Fixing this issue is as simple as casting to `int(row), int(col)` before returning.

Playing Against the Server

To start playing against the server, simply execute the script you just created.

```
python play_online.py
```

This should connect you to the server and prompt you for two things:

- *"Enter API token: "*
  In which case you can simply copy-paste the token you obtained earlier
- *"Do you want to attempt to qualify? (3 attempts left) [y/n]"*
  In which case you should **say NO** (for now, see below)

The client should now start a tournament where you play several games against remote actors. Whenever it is your turn to make a move, the `handle_get_action` hook will be called, which in turn will invoke the logic in `MyHexActor`. At the end of the tournament, you will see a score corresponding to the percentage of games you won.

We **strongly suggest that you answer no** to the qualification question until you know that your setup has been thoroughly debugged and that your actors are sufficiently strong. Your answer to the question determines whether you will play against random actors (no), or more competent actors (yes). Victories against the competent actors are what ultimately determine part of your grade for this project, but you have a limited number of allotted attempts, so be careful with how you budget your qualification games.

The initial dialogue with the server can be automated a bit. Instead of having to repeatedly copy-paste your API token every time you play, you can enter it into the client as a constructor argument:

```
client = MyClient(auth="<your-token>")
```

... or by setting an environment variable before launching the script.

```
export IT3105_AUTH=<your-token>
```

The same applies to the qualification question. A canned answer can be set in the constructor (`qualify=False`) or as an environment variable `IT3105_QUALIFY=0`.

Qualifying

Once you are sufficiently confident in your actors, you can start a qualification session. To do this, answer yes when prompted about whether you want to qualify (alt: set `qualify=True` or export `IT3105_QUALIFY=1`).

Note: Disconnecting (or timing out) will cause you to lose **all** the remaining games in the qualification session.

Note: Unless it happens during the first couple of games, timing out (or disconnecting in any way) will cost you a full qualification attempt.

Your grade will be determined by your best score across all qualification attempts, so there is no need to not use them all before the deadline. Do keep in mind, however, that the servers can get a bit sluggish if too many people try to connect concurrently, so don't postpone your qualification games until the very end.

To see your best qualification score, execute the `get-profile` CLI command mentioned in the "Getting Access" section above. Alternatively, you can navigate to http://it3105.idi.ntnu.no/qualifiers. Your score should be displayed next to your handle (nickname). See the "API Methods" section below for information on how to change your handle.

Ranked League Play

It is also possible to play against other students in an open league. You should not have to make any changes to your agent or client code to make this work (as long as `handle_get_action` correctly returns an action depending on whether you are player 1 or player 2). All you have to do is set the option `mode='league'` when running the client:

```
if __name__ == '__main__':
    client = MyClient()
    client.run(mode='league')
```

This will throw you into the matchmaking pool and the server will attempt to find an opponent for you. If found, you will play a single series of games against the opponent before the connection is closed.

Note: Failing to produce a valid action will cause you to lose the current game (but you can still win the remaining ones).

Note: Disconnecting (or timing out) will cause you to automatically lose all remaining games in the matchup.

Note: Repeatedly disconnecting or forfeiting too many games due to bad action responses will soft-ban you from the matchmaking system for a while. This is not intended as a form of punishment, but rather a safeguard that prevents you from losing too many games if you have a bug, and conversely, a way to assure that you will be paired up with worthy opponents.

Your performance in this tournament will **not affect your grade**, it is just supposed to be a bit of fun. However, for those interested in bragging rights, the outcome of each game will be used to update your standing in an Elo-based rating system. Your current rating will be visible next to your handle at http://it3105.idi.ntnu.no/league.

There is no limit to the number of times you can play in the league, so don't hesitate to try it. Also, as further explained below, performance early on doesn't matter that much, so feel free to use it as a testbed for different strategies or to gauge the power level of your Hex agents.

At the start of every day (00:00), all ratings will be decayed by a factor of 0.1 towards 1300 (the starting rating). For instance, a 1400 rated player will have their rating decayed by 10 to 1390, a 1700 rated player will have their rating decayed by 40 to 1660, and a 1200 rated player will gain 10 ratings and end up at 1210. We have introduced this system to incentivize continuous participation and to make recent performance count more. We might change the decay factor if it proves to be too high or too low, but we don't think it will matter much since it is easy to play a lot of games (you can farm Elo while making dinner).

Feel free to be creative with the agents you submit to the league. The Hex players you send to the qualification games (and demo) should comply with the provided spec, but the league is more of a free-for-all. Of course, we don't condone any form of obvious cheating (exploits/server attacks/etc), but as long as your Hex agent produces a valid action, it doesn't matter how that action was generated.

All Game Hooks

In addition to the crucial `handle_get_action` method, there are other hooks in the `ActorClient` that allows you to inject your own code at various points in time. In addition to a few administrative handlers for logging and error handling, there are six callbacks that will be invoked at various points throughout a nominal game session.

By default, the hooks will just log information to the terminal (except for `handle_get_action` which also generates a random valid action), but you may override any of them with your own logic if you want to, for instance, reset some state in your agent at the beginning of each game.

Below follows an example of how the various hooks get called throughout a gaming session where you play M games against N opponents.

- *Repeat N times (once for each opponent)*
  - `handle_series_start`
  - *Repeat M times (once for each game)*
    - `handle_game_start`
    - *Repeat until the game is over (once for each time it is your turn):*
      - `handle_get_action`
    - `handle_game_over`
  - `handle_series_over`
- `handle_tournament_over`

The provided source code in `ActorClient.py` should be documented reasonably well, but as a reference, we include the default code for each of the six hooks in the sections below.

Before each set of games

```python
def handle_series_start(
    self, unique_id, series_id, player_map, num_games, game_params
):
    """Called at the start of each set of games against an opponent

    Args:
        unique_id (int): your unique id within the tournament
        series_id (int): whether you are player 1 or player 2
        player_map (list): (inique_id, series_id) touples for both players
        num_games (int): number of games that will be played
        game_params (list): game-specific parameters.

    Note:
        > For the qualifiers, your player_id should always be "-200",
          but this can change later
        > For Hex, game params will be a 1-length list containing
          the size of the game board ([board_size])
    """
    self.logger.info(
        'Series start: unique_id=%s series_id=%s player_map=%s num_games=%s'
        ', game_params=%s',
        unique_id, series_id, player_map, num_games, game_params,
    )
```

Before each game starts

```python
def handle_game_start(self, start_player):
    """Called at the beginning of of each game

    Args:
        start_player (int): the series_id of the starting player (1 or 2)
    """
    self.logger.info('Game start: start_player=%s', start_player)
```

When the server needs you to make a move

```python
def handle_get_action(self, state):
    """Called whenever it's your turn to pick an action

    Args:
        state (list): board configuration as a list of board_size^2 + 1 ints

    Returns:
        tuple: action with board coordinates (row, col) (a list is ok too)

    Note:
        > Given the following state for a 5x5 Hex game
            state = [
                1,              # Current player (you) is 1
                0, 0, 0, 0, 0,  # First row
                0, 2, 1, 0, 0,  # Second row
                0, 0, 1, 0, 0,  # ...
```

```python
                    2, 0, 0, 0, 0,
                    0, 0, 0, 0, 0
                ]
            > Player 1 goes "top-down" and player 2 goes "left-right"
            > Returning (3, 2) would put a "1" at the free (0) position
              below the two vertically aligned ones.
            > The neighborhood around a cell is connected like
                  |/
                --0--
                 /|
        """
        self.logger.info('Get action: state=%s', state)
        row, col = self.get_random_action(state)
        self.logger.info('Picked random: row=%s col=%s', row, col)
        return row, col
```

After each game

```python
    def handle_game_over(self, winner, end_state):
        """Called after each game

        Args:
            winner (int): the winning player (1 or 2)
            end_stats (tuple): final board configuration

        Note:
            > Given the following end state for a 5x5 Hex game
            state = [
                2,              # Current player is 2 (doesn't matter)
                0, 2, 0, 1, 2,  # First row
                0, 2, 1, 0, 0,  # Second row
                0, 0, 1, 0, 0,  # ...
                2, 2, 1, 0, 0,
                0, 1, 0, 0, 0
            ]
            > Player 1 has won here since there is a continuous
              path of ones from the top to the bottom following the
              neighborhood description given in `handle_get_action`
        """
        self.logger.info('Game over: winner=%s end_state=%s', winner, end_state)
```

After each set of games

```python
    def handle_series_over(self, stats):
        """Called after each set of games against an opponent is finished

        Args:
            stats (list): a list of lists with stats for the series players

        Example stats (suppose you have ID=-200, and playing against ID=999):
            [
                [-200, 1, 7, 3],  # id=-200 is player 1 with 7 wins and 3 losses
                [ 999, 2, 3, 7],  # id=+999 is player 2 with 3 wins and 7 losses
            ]
        """
        self.logger.info('Series over: stats=%s', stats)
```

After all sets of games

```python
    def handle_tournament_over(self, score):
        """Called after all series have finished

        Args:
            score (float): Your score (your win %) for the tournament
        """
        self.logger.info('Tournament over: score=%s', score)

```

API Commands

`ActorClient.py` comes with utilities for accessing an API server for operations such as resetting your authentication token, querying information about your user, and updating your player handle (nickname). The API methods are accessible through a command-line interface that is documented below.

Note that some of the API methods require login. API authentication is done with the same token as you use to access the gaming servers. Here too, you can preconfigure the API token with `export IT3105_AUTH=<your-token>` and omit the (`--auth`) argument in the sections below.

Request Reset Token

Resetting your API token is a two-step process. First, you need to request a reset:

```
python -m ActorClient request-reset-token --email <your-email>
```

You should now receive an email containing a **temporary** token that can be used to reset your main token. Contact the course staff if you run into any issues.

Note: To prevent abuse and spam, you can only request a token reset once per hour.

Reset Token

Having received an email from it3105 in your inbox, copy the temporary token from the body and paste it into the following command.

```
python -m ActorClient reset-token --temp-token <token-from-email>
```

You should now get a message confirming that your token has been reset, along with the new token.

Get Profile (login required)

Info about your user, including the number of qualification game attempts you have left, your registered email, your player handle (see below), and your best qualification score, can be queried with the following command.

```
python -m ActorClient get-profile --auth <your-token>
```

And should return something like:

```
{
    "attempts": <number-of-remaining-qualification-attempts>,
    "email": "<your-email>",
    "handle": "<your-handle>",
    "score": <highest-qualification-score>
}
```

Update Handle (login required)

Your player handle, or nickname, will be used when displaying your score on the public leaderboards. By default, you should have a random one, but you can change it to something more flashy if you want.

```
# To set a specific handle
python -m ActorClient update-handle --auth <your-token> --handle '<handle>'

# Use empty string to reroll a random handle
python -m ActorClient update-handle --handle ''
```

Note: We haven't implemented sophisticated input validation logic, but please don't pick profanities or slurs.