```
# Quarto Network
# Hosts functions to get and recieve moves over network as well as
# transferring data.
# Sean Straw & Ari Cohen

import threading
import SocketServer
import socket
from quarto_interface import *
from quarto_player import *
from quarto_state import *
from time import sleep
from copy import deepcopy

# A basic function to send data broken into chunks.
def data_send(file_full,connection):
    buffer_size = 1024
    file_size = len(file_full)
    file_sections = int(round(file_size/float(buffer_size) + .5))  # We're
        rounding up here.  A bit odd but this works best.
    buffer_length = file_sections * buffer_size
    precopy_info = str(file_size) + '$$$' + str(file_sections) + '$$$' + str
        (buffer_size)
    connection.sendall(precopy_info + '\n')
    check = connection.recv(1024)
    if check == '$ready$':
        for x in range(0,buffer_length, buffer_size):
            file_part = file_full[x:x+buffer_size]
            connection.sendall(file_part)

# A basic function to receive data broken into chunks
def data_receive(connection):
    precopy_info_dirty = connection.recv(1024)
    precopy_info = precopy_info_dirty.split('$$$')
    file_size = int(precopy_info[0])
    file_sections = int(precopy_info[1])
    buffer_size = int(precopy_info[2])
    connection.sendall('$ready$')
    file_full = ''
    for x in range(file_sections):
        file_part = connection.recv(buffer_size)
        file_full += file_part
    return file_full

def list_numbers_to_strings(array):
    new_array = []
    for index in range(len(array)):
        new_array.append(str(array[index]))
    return new_array

def list_strings_to_numbers(array):
    new_array = []
    for index in range(len(array)):
        new_array.append(int(array[index]))
    return new_array
```

```
    # Socket only sends strings.  These functions will format the data into
    # strings to be sent by socket.
    def format_game_data(game_state):
        available_pieces = game_state.get_available_pieces()
        available_pieces = list_numbers_to_strings(available_pieces)
        available_pieces_string = '$'.join(available_pieces)
        squares = game_state.get_squares()
        squares = list_numbers_to_strings(squares)
        squares_string = '$'.join(squares)
        current_piece = game_state.get_current_piece()
        current_piece_string = str(current_piece)
        game_state_string = (available_pieces_string + '$$' +
                             current_piece_string + '$$' +
                             squares_string)
        return game_state_string

    def format_move_data(move):
        move_row = move.get_row_placement()
        move_row_string = string(move_row)
        move_col = move.get_col_placement()
        move_col_string = string(move_col)
        move_piece = move.get_piece()
        move_piece_string = string(move_piece)
        move_string = (move_row_string + '$$' +
                       move_col_string + '$$' +
                       move_piece_string)
        return move_string

    # Reformats a string from format_game_data() as a GameState class.
    def rebuild_game_data(game_state_string):
        game_state = GameState()
        [available_pieces_string, current_piece_string, squares_string] =
            game_state_string.split('$$')
        available_pieces = available_pieces_string.split('$')
        game_state.available_pieces = list_strings_to_numbers(available_pieces)
        squares = squares_string.split('$')
        game_state.squares = list_strings_to_numbers(squares)
        game_state.current_piece = int(current_piece_string)
        return game_state

    def rebuild_move_data(move_string):
        move = GameMove()
        [row_string, col_string, piece_string] = move.split('$')
        row = int(row_string)
        col = int(col_string)
        piece = int(piece_string)
        move.set_move(row,col,piece)
        return move

    # Rebuilds the previous move based on the previous game state and the
    # new game state.
    def determine_move(pre_game_state, post_game_state):
        cheating_flag = False
        move = GameMove()
        move_pieces_changes = []
        move_squares_changes = []
```

```
        pre_game_state_pieces = pre_game_state.get_available_pieces()
        pre_game_state_squares = pre_game_state.get_squares()
        post_game_state_pieces = post_game_state.get_available_pieces()
        post_game_state_squares = post_game_state.get_squares()
        if (len(pre_game_state_pieces) != len(post_game_state_pieces) or
            len(pre_game_state_squares) != len(post_game_state_squares)):
            cheating_flag = True
        for piece in range(len(pre_game_state_pieces)):
            if (pre_game_state_pieces[piece] != post_game_state_pieces[piece] and
                pre_game_state_pieces[piece] == GameState.AVAILABLE):
                move_pieces_changes.append(piece)
            elif (pre_game_state_pieces[piece] != post_game_state_pieces[piece] and
                  pre_game_state_pieces[piece] != GameState.AVAILABLE):
                cheating_flag = True
        for square in range(len(pre_game_state_squares)):
            if (pre_game_state_squares[square] != post_game_state_squares[square] and
                pre_game_state_squares[square] == GameState.EMPTY):
                move_squares_changes.append(square)
            elif (pre_game_state_squares[square] != post_game_state_squares[square]
                and
                  pre_game_state_squares[square] != GameState.EMPTY):
                cheating_flag = True
        if len(move_squares_changes) == 0 and len(move_pieces_changes)==0:
            return [None, GameStatus.QUITTING]
        elif len(move_pieces_changes) == 0:
            new_piece = 0
        elif len(move_pieces_changes) > 1:
            cheating_flag = True
        elif len(move_pieces_changes) == 1:
            new_piece = move_pieces_changes[0]
        if len(move_squares_changes) > 1:
            cheating_flag = True
        if post_game_state_squares[move_squares_changes[0]] != pre_game_state.
            get_current_piece():
            cheating_flag = True
        if cheating_flag == True:
            notify("Cheating may have been detected.") # reserves functionality to
                break or whatever
        move_row = move_squares_changes[0] / 4
        move_column = move_squares_changes[0] % 4
        move.set_move(move_row, move_column, new_piece)
        return [move, GameStatus.PLAYING]

    class HostData():
        HALT_SERVER = -1
        KEEP_SERVER = 0
        HOST_MOVE = 0
        CLIENT_MOVE = 1
        SIGNAL_GAME = 2
        NO_CLIENT = 0
        CLIENT_CONNECTED = 1

        def __init__(self):
            self.server_action = HostData.KEEP_SERVER
            self.player_turn = self.HOST_MOVE
            self.client_status = self.NO_CLIENT
```

```
        def get_game_state(self):
            return self.game_state

        def set_game_state(self, game_state):
            self.game_state = game_state

    server_host_database = HostData()

    class Server(SocketServer.BaseRequestHandler):
        def handle(self):
            global server_host_database
            server_host_database.client_status = HostData.CLIENT_CONNECTED
            data_receive(self.request)
            data_receive(self.request)
            while server_host_database.server_action == HostData.KEEP_SERVER:
                while server_host_database.player_turn == HostData.HOST_MOVE:
                    sleep(1)
                if server_host_database.player_turn == HostData.SIGNAL_GAME:
                    data_send(server_host_database.game_state_string,
                              self.request)
                    break
                data_send(server_host_database.game_state_string,
                          self.request)
                server_host_database.game_state_string = data_receive(self.request)
                server_host_database.player_turn = HostData.HOST_MOVE
                sleep(1)


    def start_host_server(player):
        global server_host_database
        notify("Server starting!")
        host_server = SocketServer.TCPServer((player.HOST_IP, player.HOST_PORT),
            Server)
        host_ip, host_port = host_server.server_address
        host_server.host_database = HostData()
        threaded_server = threading.Thread(target=host_server.handle_request, args=()
            )
        threaded_server.start()
        notify('Server running on port %i' % host_port)
        while server_host_database.client_status != HostData.CLIENT_CONNECTED:
            sleep(1)
        notify('Client connected!')
        return host_port

    def connect_to_host(player):
        connection_to_host = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        connection_to_host.connect((player.HOST_IP, player.HOST_PORT))
        data_send('$ready$', connection_to_host)
        notify('Connected to host!')
        return connection_to_host

    def signal_host_game_over(player, game_state):
        game_state_string = format_game_data(game_state)
        data_send(game_state_string, player.connection)
        player.connection.close()
```

```
def signal_client_game_over(player, game_state):
    global server_host_database
    game_state_string = format_game_data(game_state)
    server_host_database.game_state_string = game_state_string
    server_host_database.player_turn = HostData.SIGNAL_GAME

def get_network_host_move(game_state, connection):
    game_string = format_game_data(game_state)
    data_send(game_string, connection)
    notify("Waiting for host to move...")
    new_game_state_string = data_receive(connection)
    new_game_state = rebuild_game_data(new_game_state_string)
    [new_move, game_state] = determine_move(game_state, new_game_state)
    return [new_move, game_state]

def get_network_client_move(game_state):
    global server_host_database
    server_host_database.game_state_string = format_game_data(game_state)
    server_host_database.player_turn = HostData.CLIENT_MOVE
    notify("Waiting for Client to move...")
    while server_host_database.player_turn == HostData.CLIENT_MOVE:
        sleep(1)
    new_game_state = rebuild_game_data(server_host_database.game_state_string)
    [move, game_state] = determine_move(game_state, new_game_state)
    return [move, game_state]
```