CEng 445 Spring 2023 Projects Phase 2 Description

In phase 4, you are going to convert your phase 2 server to a Websockets server. You can also use Django channels or other frameworks but `websocket` library is the best way to minimize the work. You need at least version 11 for `threading` compatible library.

You can find Websocket versions of the chat service example at:
https://github.com/onursehitoglu/python-445/blob/master/examples/chatsrv-ws.py

You need to:

```
from websockets.sync import server   # this is threading compatible server
from websockets.exceptions import ConnectionClosedError,ConnectionClosedOK
```

Then, instead of `socket` library calls `socket(), bind(), listen(), accept()` and starting a TCP thread per accepted connection, start a websockets agent using:

```
def agent(wsock):
    peer = sock.remote_address

    try:
        while True:
            inp = wsock.recv()
            ...
            # you can reply by wsock.send(str)
    except ConnectionClosedOK:
        # peacefull termination
    except ConnectionClosedError:
        # client generated an error


# this will create a thread calling `Agent(wsock)` per connection
srv =  server.serve(Agent, host=HOST, port=PORT)
srv.serve_forever()
```

Your notification thread can `send()` to socket as in phase 2. So you need to change the way you create the threads.

In phase 4, you had better use JSON as your command and data exchange. A typical way is to convert all client calls in form:
`obj(id).command(a=aval,b=bval)` into
`{"obj": "id", "method":"command", "a": "aval", "b":"bval"}`

You can format your results as:
`{"status": "success", "value": { "obj": "id",...}` or in case of errors:
`{"status": "fail", "reason": "object cannot be found"}`

The notifications should be pushed by the server side and browser should update the model. Do partial update whenever appropriate. Use global reload of the model only when it makes sense.

The main idea of phase 4 is to have a rich web application with all user interaction takes place in the browser and server side only provides collaborative update of the model and enforcement of some semantics. You can use local files to serve landing page, scripts and images. If you like, you can use Django as well. You can use the following lines to attached to a Django application environment. For example you can include them to make your phase 2 server use Django models for persistency.

```
import django
from django.contrib.session.models import Session

def setupDjango(projectpath, projectname):
```

```
    '''call this once to setup django environment'''
    sys.path.append(projectpath)
    os.environ.setdefault('DJANGO_SETTINGS_MODULE',projectname + '.settings')
    django.setup()

# call setupDjango("/home/user/","ceng445") for ceng445 project at /home/user
```

You can use Javascript libraries that are related to the nicer views of your project.

Race Map

You can use Canvas API or higher level libraries like Konva or Fabric.js to edit the game grid. Your editor should include a toolbox of components and drag drop based editing support. When game is on, editing will stop and cars will be controlled by the keyboard and/or mouse on the browser side.

The notifications in the edit mode are immediate where they are periodical in the game mode. That means the positions of cars are sent from server in a predetermined frequency. Feel free to fine tune its value by experimenting. A value less than 5 updates per second will be too slow, more than 25 will be useless. During the game, browser sends user input to interact with the grid cell that will determine the cars new position and reply back. Browser gets positions of other cars via periodical notifications.

You can use the square tiles from:
https://opengameart.org/content/racing-pack

or better one you can find. Do not send full images from server. Just respond in image id and browser loads that id. The cells and cars can be rotated using `rotate(angle)` in Canvas API.

All changes in the map during the editing will send notification to other attached clients. The notifications are `View` aware, so that they only agents with the containing view are notified. When agents take notifications, they update their view calling `draw` command on the view object. Note that each connection will have only one view, a newly created view should override the previous one.

In the demo a graphical view of grid, edit mode with instant modifications, and a multicar game mode should be demonstrated.

Web Dash

In this phase, editing will be interactive using drag and drop whenever appropriate. The components will be on the toolbox and dragging on a `Dash` view, an instance will be created. Environment variables will be edited by a pop up area. You can render the form for parameters on a pop up area or render as the component body. The landing page should correspond to a `Dash` which is a tabbed view. You can use *jquery UI Tabs*. Make your browser model updated by notification even if it is not in the active `Tab`.

The periodic *refresh* of the component will be made by the browser side where server triggered events like `Timer` or `DBUpdate` submissions are pushed by the server. Similarly all edit events are instant.

In the demo, you need to show all supported components with instant notifications and updates.

Graph Data

You can use *Joint.js* in this phase, which draws graphs and lets you edit them. A typical demo is:
https://www.jointjs.com/demos/roi-calculator

The `ForeignObjectElement` class implements HTML text to be rendered inside graph nodes.

After adding components from a toolbox with drag and drop, you can edit the component by linking ports and updating parameters as forms. Each component will be a small HTML element with a form of parameters.

In this phase, you will add a component `Chart` which will display a chart from tabular data. You can use *plotly*, *chart.js* or simply export `SVG` on `pandas` and show it.

In the demo, you need to show all supported components with instant notifications and updates.