CEng 445 Spring 2023 Projects Phase 2 Description

In phase 2, you are going to implement a TCP service as your application. The service is going to listen to a TCP port taken as a command line option as: `python3 yourapp.py  --port 1423`

For each connection request (`accept()` returns) you start a thread (`threading`) that serves the connection that we will call the `agent`. Each agent has two responsibilities:

- read requests from the client, make corresponding library calls, and returns results for the client,
- when there is a notification in the system that client should be informed, write a message to the client.

You can start multiple threads per agent for that purpose. (see (chatsrv.py)[https://github.com/onursehitoglu/python-445/blob/master/examples/chatsrv.py] in the repository)

The client and your server speaks in a textual protocol that all calls are converted into commands. For simple data, you can use space separated list of words as:

```
additem 2231 Itemname "item long description" 10 2.5
```

that can be translated into library call as:

```
obj['2231'].additem('Itemname','item long description', 10,2.5)
```

Each component has a unique identifier in the namespace of the editted model. Operations on components use this identifier to address the component.

If commands require larger data to be transferred (like an image), you had better use JSON like objects. For large data, the size can be issue. You can read an integer for command size, than read the command structure. i.e. if client needs to send a command of size 1532, it first writes a binary representation of 1532 (see `struct` module), then writes the 1532 bytes of JSON. The other end (client and server) first reads an integer as size of communication, then call `sock.recv(size)` to get the `bytes` string containing the JSON.

After establishing a TCP connection, a client needs to set its username with `USER`*username* command or a similar mechanism.

For notifications, you can either create a condition variable per data structure (grid, page, graph etc.) and/or a condition variable per user. When a notification is required, the `notify/notify_all` method can be called selectively on the threads of interest.

The receiver of the notification can reload the content and update the view. In the following example, observers registers with a set of values they are interested in and only observers with registered values are notified:

```
class Observer:
    '''multithread observer example
       observer thread can observe a selection
       of values in the set of integers'''
    def __init__(self,N):
        self.values = [0 for i in range(N)]
        self.mut = RLock()
        self.observers = {}

    def register(self, obj, vset):
        ''' A new condition is created per observer and it is
            notified when interest set of the observer changes'''
        with self.mut:
            cond = Condition(self.mut)
            self.observers[obj] = (vset, cond )
            return cond
    def unregister(self, obj):
        with self.mut:
```

```
            del self.observers[obj]

    def wait(self, obj):
        with self.mut:
            if obj in self.observers:
                self.observers[obj][1].wait()

    def __getitem__(self, idx):
        with self.mut:
            return self.values[idx]
    def __setitem__(self, idx, v):
        with self.mut:
            self.values[idx] = v
            # notify interested observers
            for obs in self.observers.values():
                if idx in obs[0]:
                    obs[1].notify()
```

All data structures should be working in critical region and accessed exclusively. You can implement your editable objects as **monitors**.

A typical usage scenario will be: * connect the server * list available instances of edited objects * create a new instance, and attach it or, attach one of the instances * interact with instance * Get notifications from another user and edit/update instance to generate notifications

The notifications received by `agents` sends an informative message to client socket.

In addition to notifications, this phase will add persistence. When a user enters `save` command, currently edited object is written on a file or database. When program is started again, the existing state will be restored. You can either auto-save when last user detaches an object or do nothing and loose the changes. Another alternative is to use an ORM (Object Relational Model) like SqlAlchemy and store your model in a database so that all updates will be persistent. This will need extra work from phase 1 and slow down some of your updates. Please note that the objective of this phase is to practice concurrent programming. Even if you use a database model, you should define your critical regions properly.

For simple operations, use a tool like `netcat` to simply send/receive from the socket. For more sophisticated operations, prepare a TCP client for your demo. Your TA will inform you about the demo requirements.

Please note that you will use a `websocket` wrapper to this phase implementation in your fourth phase so that your server will interact with a browser. Please keep this in mind when you are making implementation choices.

The topic specific descriptions of Phase 2 are given below.

Race Map

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 addition to concurrent implementation, second phase has two challenges:

- Adding checkpoint components.
- Game mode.

Checkpoint components record time of interaction for each car. They share per `Map` data so that there is a legal ordering of checkpoints that a car should visit. Each car has a next checkpoint and only that checkpoint will record the interaction. When a car interacted with a checkpoint, the next checkpoint in the track will

be active for that car. In this way, cars have to visit a legal path on the track. For example, if I place 4 checkpoints in 4 corners, NW, NE, SE, SW, a car should start from NW corner and visit all checkpoints clockwise. During the game, `Map.draw()` report will order cars based on laps and last visited checkpoint scores. Feel free to make changes in the checkpoint class design if you have a better idea.

When a map enters the game mode (`Map.start()`), a game controller thread is started, which will periodically call `car.tick()` for all of the cars and push notifications to the agent views. Edit notifications are immediate, but game loop notifications will be sent once in `P` tick completions (determine `P` based on your observations). No editing will be allowed in the game mode. Game mode can be started by any client and can be ended by any client (add new methods to `Map` if you need).

Optionally, you can put your TCP clients in a game mode too. Use `blessed` library to enable a terminal mode, get key press events and draw the grid on a fixed location for a game like experience.

Web Dash

In the second phase, all components will be constructed with a refresh period in seconds (integer). User can pass a specific value or default value will be set. You need to implement a `TimerThread` thread which queues all refresh timers and trigger when timer expires. You can implement it as a sorted list of expiration timestamps. The timer thread sleeps for a second and checks the expiration of first item in an infinite loop. It triggers a refresh event (possible condition variable notification) on the components and insert their next expiration in the queue. You can use `heapq` or a similar library. The `TimerThread` triggering a component refresh causes the `Tab` containing it to be notified and all users attached to the `Dash` containing the `Tab`. You can make `Timer` component implementation compatible with this new thread.

A user notification should have the component(s) information as 'component … is refreshed' so that only the relevant components need to be loaded by the client end of the socket.

In addition to `TimerThread`, two new components will be implemented:

- `DBUpdate`: The component will have the `query` variable in the environment and a set of `param` names are defined. User sets the parameter variables and trigers `submit` event. Similar to `DBQuery`, it is executed on an SQLite3 database. However parameter values will be substituted in the `query` which is a `str.format()` specifier.

  For example if the parameters are `isbn`, `title`, and `year`, and `query` contains:

  ```
  REPLACE INTO book(isbn,title,year)
  VALUES ({isbn},{title},{year});
  ```

  The result of `env['query'].format(param)` will be executed on the database.

- `FileShare`: Component will be used as a simple shared directory. It will have a `path` variable in the environment which is realized as a relative directory in the filesystem of the server. The view will contain the directory listing, list of files, their sizes and the update timestamp. Whenever user sets `param['filename']` and `param['content']` and triggers `upload` event, the file is created or updated in the directory. When user sets `param['filename']` and triggers `download` event, the file content will be put into `param['content']`. Similarly `delete` event deletes a file from the directory. This component will be refreshed when one of the attached users make a change in the shared content.

As a clarification of a missing point in the first phase, components store `param` variables in a per user basis. When a user sets a `param` value, it is local to that user. Only that user can inspect its value. This way, multiple users can set parameters and trigger the actions concurrently. In order to implement this, we need to decouple parameter values from the `Component` objects. Per component `param` dictionary is stored and maintaned by the `agent`, the thread handling the communication. The `trigger` method of the `Component` is modified as:

`trigger(event, param)`: calls `event` method of the object with `param` argument as in `event(param)`. `param` is the parameter dictionary ie. `upload({'filename':'test.txt','content':'hello'})`.

Graph Data

In this phase, the `Graph` object will be in *auto-compute* mode. In auto-compute mode, each modification or explicit `compute()` call cause a node to compute. This will trigger computation of the adjacent (outgoing) nodes and the computation will propagate to all dependent nodes. You should not call `Graph.compute` for all updates, only relevant nodes should be computed.

All attached users should receive a notification for all updated nodes. The notifications on the client end should contain id information of the updated component.

You need to implement the following additional components:

- `AddColumn`: Gets a function as a lambda expression like `Filter`. The function return one or more values. Each value is added as a column to the table. The component has one input one output port.

  Assuming each row is passed as a dictionary to function:

  ```
  ac = Repo.components.create('AddColumn')
  ac.expr = lambda (self) : {'total': (self['mt'] + self['fin'])/2}
  ```

- `RegExImporter`: It is an importer component reading a text file and parsing each line with a regular expression. For each match of the regular expression, the group dictionary (`re.match(...).groups()`) values will be set as a row.

  ```
  ri = Repo.components.create('RegExImporter')
  ri.regex = '([^ ]+) [^ ]+ [^ ]+ \[([^]]+)\] "([^"]+)" ([0-9]+) ([0-9]+)'
  ri.columns = ('remote','time','method','result','size')
  ri.file = '/var/log/apache2/access.log'
  ```

  This importer will import a table with columns `remote, time,method, result, size` where each grouping expression `(..)` in regexp defines the value of a column.

  A sample line from such a file is:

  ```
  144.122.254.254 - - [20/Nov/2024:12:48:26 +0300] "GET /lab/?path=index.ipynb HTTP/1.1" 200 1022
  ```

`View` component notifications will dump the view to client socket.

You can use `pandas` library in this phase. You will be drawing graphs as an exporter component in the following phases.