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 or process that serves the connection that we call `agent`. Each agent has two jobs, read text requests from client and make corresponding library calls and returns results for the client, and when there is a notification in the system that client should be informed, write a message to the client. You can make each agent multithread 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 obj2231 Itemname "item long description" 10 2.5` that can be translated into library call as: `find('obj2231').additem('Itemname','item long description', 10,2.5)`

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.

After establishing a TCP connection, a client needs to authenticate in order to make privileged tasks. `authenticate(user,pass)` (or `login(user,pass)`) call will start the session. As the authentication database, you can use `sqlite3`. A typical implementation will be:

```
import hashlib
import sqlite3

def login(user,pass):
    with sqlite3.connect('project.sql3') as db:
        c = db.cursor()
        row = c.execute('select username,password from auth where username=?',(user,))
    if hashlib.sha256(pass.encode()).hexdigest() == row[1]:
        return True
    return False

def adduser(user,pass):
    encpass = hashlib.sha256(pass.encode()).hexdigest()
    with sqlite3.connect('project.sql3') as db:
        c = db.cursor()
        c.execute('insert into auth values (?,?)',(user,encpass)
```

For notifications, you can either create a condition variable per data structure (document, 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.

Alternatively, you can convert all callback interface into a condition variable and a message queue per receiver. The notification message will be inserted at the message queue and the condition variable will be notified. The agent waiting for notifications process the notification and show a message. All data structures should be working in critical region and accessed exclusively. You can implement basic objects as **monitors**.

A typical usage scenario will be: * connect the server * list available instances of edited objects * create a new instance * attach one of the instances (depends on topic) * edit instance or interact with instance * Change or create a view * Get notifications from another user and edit instance to update instance to generate notifications

Some project topics have query filters applied to notifications. You can choose to send notification to all interested users and user agents apply the query to update their view. Another approach will be applying the filter on the notifier site and send notification to interested (matching query) agents only. Notifications should be printed on the clients screen as reprinting the view or writing the details of the notification, item x added, updated etc.

In addition to notifications, this phase will add persistence. When a user enters `save` command, existing objects are written on a file or database. When program is started again, the existing state will be restored.

The extra implementation required in phase 2 are defined below depending on the project topic.

Schedule Manager

In this project, user can have multiple `View` objects where user can attach one at a time. Only notifications of the attached view will be received. Each `View` may refer to multiple `Schedule` objects. Notifications are based on Views. Each user may have one view active at a time. View objects belongs to users, they are not shared.

Users get notification about `Events` in the `View` objects filtering criteria. When an event matching criteria is created, updated, or deleted a notification is generated. In addition to `Event` changes in the `View`, users get notifications referring themselves. Adding, updating or deleting an event with user being a assignee will notify the user regardless of the `View`.

In this phase you will implement an authorization (access control) to `Schedule` and `Event` objects. Protection is a dictionary mapping users to a set of operations. For `Schedule`, the following operations are defined: * `LIST`: user can list the `Event` objects in the schedule. * `ADD`: user can add new events to the schedule. * `ACCESS`: user can access the events in the schedule by using their id. Note that this can be done without `LIST` permission. Operations on the `Event` is subject to event protection also. * `DELETE`: user can delete an `Event` in the schedule if s/he also has `WRITE` permission on it. Owner of the `Schedule` can delete the `Event` without `WRITE` permission.

The owner (creator) of the `Schedule` always has all permissions above.

For `Event`, the following operations are defined: * `READ` : user can read the `Event`. * `WRITE` : user can change the `Event`. * `ASSIGNED`: user can add/remove himself/herself as an attender of the event.

Queries only return events with `READ` permission. Users can only query schedules with `LIST` permission. Protection value matches a user with a set of values above. If no match, no permission is granted for non-matching users. Username `*` matches all users. When a username and `*` matches both, the username permissions are effective. Note that update of protection via CRUD methods also change the observing users view. Observing users should be notified so that they update their view.

Map Making Game

After authentication, user can list the existing `Map` objects in the system or create a new one. Instead of attaching, users execute `join mapid username teamname` command on a `Map` object. If `teamname` exists, users map will be existing team map, otherwise the team will be created.

For this phase, you can load configuration templates for global `Map` construction from a configuration file. Configuration file consists of configurations for different template names. When a map constructed with a template name, configuration from file will be used. User will run a command like: `newmap 1000x1000 arena`

The configuration information will be taken from the template name `arena` in the configuration file.

Players make a query on the global `Map` as they move and objects in their vision area will be visible. Players need to be notified if another player enters the vision of the player. Interaction with newly created game objects like mines will notify the player. i.e. a wide proximity `Mine` created in out of player vision may reduce health of the player. Also discoveries of team members will be updated on the team `Map` and other players are notified.

You can optionally add fighting rules among players of different teams. For example when player A moves to same pixel with player B, player B gets some damage. If you like, you can describe fight rules of your own.

Background image support will not be implemented in the second phase. You can choose any view of your game grid. You can print a table of objects in the viewing area and/or print a text grid to show positions.

Virtual Bulletin Board

In this project user create their `View` objects. User attaches a `Board` object to edit it if permissions are granted. User attaches a `View` object to get notifications. User notifications are subject to access control. Only readable `Message` objects generate notifications. Note that updating the permissions may also change the users view, so that related connections should be notified.

When printing the view on the client, you can print a table of messages matching the `View`. No spatial output is necessary for this phase.

Room Reservation System

You will have a detailed implementation of the access control this phase. For the `Organization`, there is a dictionary of permissions added to the constructor. It maps users to `Room` CRUD operations of the `Organization`: * `LIST`: user can list the `Room` objects in the organization. * `ADD`: user can add new rooms to the organization. * `ACCESS`: user can access the rooms and events in the organization.. Note that this can be done without `LIST` permission. Operations on
the `Room`, and `Event` are also subject to their own permissions. * `DELETE`: user can delete a `Room` in the organization if s/he also has `WRITE` permission on it. Owner of the `Organization` can delete the `Room` without `WRITE` permission. All `Events` in the `Room` are automatically deleted regardless of `Event` permissions.

For `Room` objects: * `LIST`: user can list and view the `Event` reservations for the room. * `RESERVE`: user can reserve the room. * `PERRESERVE`: user can reserve the room for periodic events. Implies `RESERVE`. * `DELETE`: user can delete the reservations for the room. It requires `WRITE` permission on the `Event`. Owner of the `Room` can delete any events `WRITE` permission on event.

for `Event` objects: * `READ`: User can see the title and details of the events. If not granted room will be displayed as `BUSY` without any other detail. * `WRITE`: User can update and delete (if `Room` has `DELETE` too) the `Event`.

All necessary constructor and update methods should be added.

A user will be notified based on `View` object. There is only one `View` object per user. User will be notified for all events in the matching `query` definitions for the `View`. Note that permissions changes may also trigger notifications.

`roomView` and `dayView` commands result on a list of tables, a table per matching `Room` and a table per matching `Day` respectively.