Timekeeper Personal Digital Assitant

Jason Tan, Leo Liu, Farnaz Zamiri, Ritwika Das

May 2022

1 Introduction

The Timekeeper is a day-planner PDA that assists the user with keeping track of when to leave for certain events.

During the day, we have many different obligations and places to be. It can be difficult to remember all of these things and get to our desired locations on time. In order to mitigate these challenges, we created a PDA that is able to send notifications to tell the user when they have to leave to arrive on time to different events during the day.

2 Related Works

The idea of a daily reminder is by no means a new concept. Danninger & Stiefelhagen, 2008 propose a similar "Virtual Secretary" that also works with schedules and current context of their user to let people know whether the user is busy or open for a chat [1]. This secretary can monitor whether the user is busy, or in a meeting, or on the phone with someone, and depending on who or why, provide information to anybody wanting to speak with the user. Perhaps it would let someone know when the meeting ends, or it would let someone know to simply "come back later" or even "knock on door" if they're on the phone. In any case, they set out with meeting and calendar information to provide information to other people. In our case, we wish to do more than simply convey information that the user already knows, so we must go beyond relying solely on the user's actions and inputs.

As Sarikaya, Ruhi mentioned, to represent a user using digital system, there are four axes [9]. One is the static information of user's profile, including name, age, gender, some preference etc. Second one is digital activity, which is the any digital information related to the person, including but not limited to calendar, email, social media. Third one is space, the physical location. Fourth one is the time. As in our project, we considered all four aspects. We considered the person's schedule. For each event in the schedule, we considered the user's current location and event location. Time is one of our main trigger in our system.

In a more recent work[4] the authors have presented a multi-function reminder for educational environments. In this study, they take into account two variables, time and location, as the contextual information. In the proposed platform, relevant reminder notifications pop-up on the user's smartphone based on the proximity of the user to specific locations. For example if the user approach the library building, they would be reminded by the system if they have a book that they need to return, or in the vicinity of a bus stop, the system notifies the user about upcoming bus leavings. For the purpose of our system, we aim to add to the context-awareness side of the system by incorporating more contextual information other than just location and time.

3 Architecture and Design

3.1 Overview

In order to implement a system that is able to perform the above mentioned activities, we separated the functionality into a few different modules/entities. Each of these modules has a different job and helps the user in different ways to get

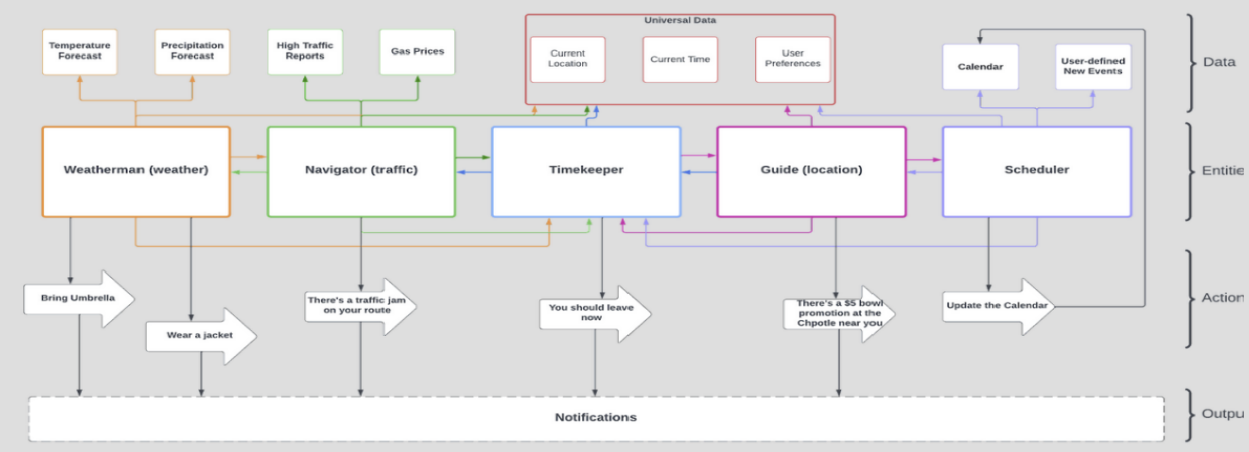

Figure 1: Original Architecture for our project

through the day. Another important part of this project is the concept of active entities.Active entities can be defined as entities/modules that do work other than just collect information and store it. In our project each module is an active entity and stores some portion of a single user's context.

3.2 Architecture

3.2.1 Weatherman

The weatherman provides the user with constant weather updates based on the user's current location as well as weather forecasts for where they will be during the day based on the events in their schedule. This also includes displaying helpful tips such as "Its raining today! Remember to bring an umbrella!" or "Be careful! It's really windy today!". This is achieved by polling OpenWeather, a weather API that allows developers to access weather forecasts based on latitude and longitude. The weatherman also communicates with the web server (to display notifications to the user) and the timekeeper.

3.2.2 Navigator

The Navigator provides accurate traffic and travel time information based on the user's preferred mode of travel. This is achieved by polling the DistanceMatrix API which allows developers to get estimates for travel time based on preferred mode of transportation.

3.2.3 Scheduler

The scheduler handles the addition of new events into the system. When a user wants to add a new event the scheduler checks if there are any conflicts and then adds an event. This action also sends out querying messages to the navigator and the weatherman for updated information relating to the event. The scheduler also handles deleting events

3.2.4 Timekeeper

The timekeeper consolidates all the information from the weatherman, navigator and scheduler to send notifications to the user such that they leave at an appropriate time to reach their destination. The timekeeper uses the weather information and the travel time/traffic information to create an estimate of the travel time. The timekeeper polls the weatherman and navigator for more current information closer to the original estimated departure time. Then it notifies the user 5 minutes prior to the time they should leave (this time can be parametrized by the user).

3.2.5 Webserver

The web server displays notifications to the user and also provides the user with an interface to

modify their daily schedule.

3.2.6 Overview of adding an event

When an event gets added to program, a massage is generated from web server and send to the scheduler. When scheduler gets the message, it will convert the event location from human readable address into latitude and longitude. Then the scheduler will check against the database for any time conflicts. If there is none, scheduler will add this event into the database. After that, scheduler will reply to web server saying everything is good, then it will broadcast a new event message to all other entities except web server. Upon receiving this new event message, timekeeper will create this event in it's memory, and then wait for other information. Weatherman will request weather information from API, and send this update to the web server and timekeeper. When web server gets the weather update message, it will update the weather information of the corresponding event. Timekeeper will also do the same thing by setting the weather field inside event object to this updated weather. Navigator will query from API how long is the traveling time, then it will send it to the scheduler and timekeeper. Scheduler will add this estimate time into the database. Timekeeper will schedule a function to run at a reasonable time. When the time comes, the scheduler will either update weather and estimate information if it is far away from event time, or it will send notification message to web server so the web server can notify the user to depart.

A little trick here is MPI guarantees the order of message arrive is in the order of message sent. Therefore, it is guaranteed that timekeeper will add an event into its memory before an estimate message send to it.

3.3 Technologies

Through the development of this project, out choice of programming language must have the following propriety: support multi-process or multi-thread, easy access to online APIs, support object oriented programming, and relatively easy to use. We decided to use python as our implementation language with MPI to support multi-process programming. It fits our requirements and there are many libraries written in python makes it easier to use. Many of the API suppliers have provided documentations and examples in python. Thus, we chose python with the following libraries to implement our idea.

3.3.1 Python

We used object oriented programming in python. Each entity is an object. Along with those we have Location object which contains latitude, longitude, and human readable address. Weather object that contains weather related information. Event object that contains event related information like start and end time, event name, etc. The event object also contains a location object and a weather object, which are important to an event. There is also a location object shared among all entities for user location. It is updated by a message broadcasted from the web server process.

Each entity would have to define a class method called run(mpi rank, mpi communicator). This function is called from main program entrance and they are the start point of each process.

3.3.2 MPI

We are using mpi4py library which allows us to run python using mpi. [5] Right now we are only using the original communicator of MPI, but a different communicator can be created in main function and pass to each different entity.

3.3.3 Flask

We chose Flask because it is easier to use and it provides debug support. We have also considered other choices including NodeJS, Java Spring. However, it will make things easier if all the modules are implemented using the same language.[2]

3.3.4 Weather

We used the OpenWeather API to gather current weather information as well weather forecasts. This API only requires the latitude and longitude of the location and returns the current forecast, a 48 hour hourly forecast, a 7 day daily forecast and also weather alerts.

3.3.5 Traffic

We used Google's distance matrix API to calculate the time it takes for the user to travel from current location to the event location by the transportation method selected by the user (driving, walking, bicycling or transit). The calculated travel time takes into account the traffic condition at the time of the API call, if "driving" has been selected as the mode of transportation. Those information will be passed around and sent to the user.

3.3.6 Geocoding

As for geocoding, we used Geocoder[3], which is a python library for translating between address and longitude, latitude. Internally, we used OSM[6] for geocoding and decoding.

3.3.7 APS

One of the major job for Timekeeper is to schedule a job to run at a specific time. The solution we used is called Advanced Python Scheduler(APS)[**aps**]. It allows us to schedule a job at any given time efficiently. The job object within this library is stored in the event object which allows us to change it easily.

3.3.8 Database

For the database of our system, we used PostgreSQL[7] which is a relational database management system. We store events' information in the database. The attributes of our database schema include event id, preferred transport mode, event location, user location, event start time, event end time, and etc. The back-end communicates with the database using the psycopg2[8] library.

4 Experimental Results

Our completed prototype implements the underlying active entity system, successfully dividing the separate domain work amongst each entity and managing communications with MPI as discussed in the Architecture section. Our front-end consisted of a week view of the calendar as shown in Figure 1, through which the client interacts with the rest of the system.

Our prototype is capable of adding, modifying, and deleting events from the calendar, with the expected addition or deletion of monitoring and calculations in interested active entities. These entities, namely Weatherman and Navigator, will update their own internal state upon receiving new information in the form of event addition or updates, as well as upon receiving new contextual information in the form of certain time, place, or, for Weatherman, forecast changes.

Timekeeper too will monitor the estimates that other active entities give it to punctually present notifications to the user with relevant information such as when they have to leave for an event or to update them on what weather to expect in the day.

Though there is much more in both breadth and depth that our prototype could grow in to, our first prototype possesses at its base an extensible framework to which more entities can be added, and in such a feature-diverse domain, such flexibility is most crucial.

5 Conclusions and Future Work

This project can be extended in many ways. The framework of the project makes it very easy to add other components that may be useful to the user. For example, an entity could be added that only looks at the current events surrounding a user's location or an entity that manages a user's notes such that they can be pulled up or displayed based on the context the user is in. The main change with adding any other functionality would go into the Timekeeper since it

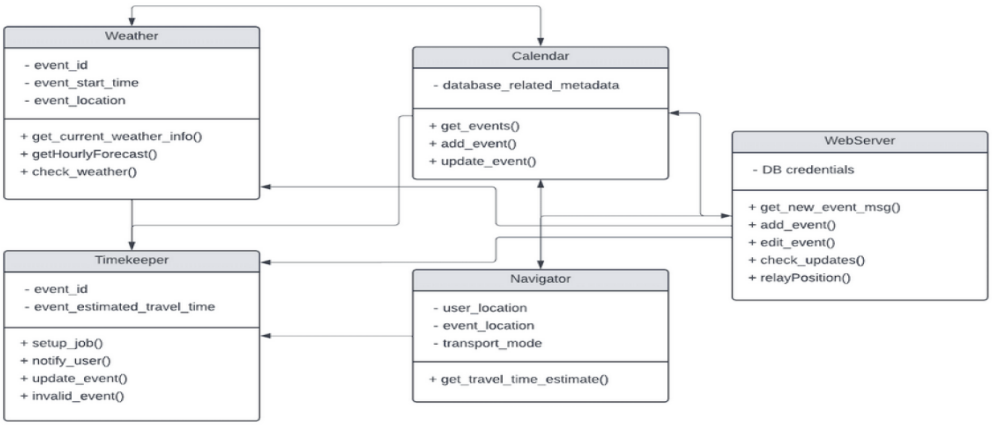

Figure 2: Partial UML for the Timekeeper

is responsible for consolidating information and creating an accurate estimated trip duration.

Following from the previous statement, not only is the project extensible within one instance of itself, but it is also extensible in the sense where we can have a Timekeeper interacting with another Timekeeper. This change would introduce a multitude of different features, where trip duration can be modified based on another user. For example, if two people are planning to meet but it is known that one of them is running late, the other person's estimated departure time can be modified to accommodate for that.

Another aspect of our project that we would like to address in the future is the use of resources. The current set-up can be slow and cause lag since we are running multiple processes. A possible solution to this could be changing the project base to c and running in a multi-threaded environment instead. However this could introduce new challenges related to multi-threaded programming such as data races.

Lastly, our original vision of the project included learning user preferences dynamically. If we could do this then the application would be able to provide the user with much more personalized information.The project could learn the preference of the driver. For example highway over small country roads or shortest time over shortest distance, no toll routes etc. Then the navigator would only suggest routes and provide estimated trip duration taking these factors into account. We could apply learning to almost all our existing entities and any new addition of an entity could also learn the user's habits.

Overall, the project we have presented is a good basis for many different extensions that would create an app that is very useful to people in their daily lives.

References

[1] Maria Danninger and Rainer Stiefelhagen. "A context-aware virtual secretary in a smart office environment". In: *Proceedings of the 16th ACM international conference on Multimedia*. 2008, pp. 529–538.

[2] *Flask*. URL: `https://palletsprojects.com/en/2.1.x/`.

[3] *Geocoder*. URL: `https://geocoder.readthedocs.io/`.

[4] Abdulaziz Guerrouat. "A Multifunction Reminder Platform for an Educational Environment". In: *2018 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT)*. 2018, pp. 1–5. DOI: 10.1109/3ICT.2018.8855757.

[5] *MPI for python*. URL: https://mpi4py.readthedocs.io/en/stable/.

[6] *OnStreetMap*. URL: https://www.openstreetmap.org/.

[7] *Potgres*. URL: https://www.postgresql.org.

[8] *Psycopg2*. URL: https://pypi.org/project/psycopg2/.

[9] Ruhi Sarikaya. "The Technology Behind Personal Digital Assistants: An overview of the system architecture and key components". In: *IEEE Signal Processing Magazine* 34.1 (2017), pp. 67–81. DOI: 10.1109/MSP.2016.2617341.