Software/Systems Design

2.1 Design a Data Refresh System

Overview

The system must receive from the user a location in the form of a mobile GPS fix (latitude, longitude, accuracy) that represents a place the user has visited. Using the Foursquare search venues API call, the system looks up information about the place and outputs it to the user. The results are cached so that if the place is visited again, the information can be retrieved without having to repeat the API call. Cached items are deleted every thirty days, requiring results to be refreshed.

Classes

The following are some class definitions in a UML-like format.

User

Data
Sensor
WebClient
Cache

Public Methods
- DisplayInfo()

Private Methods
- ParseJSON(JSON)

WebClient

Data
Client_ID : String
Client_Secret : String
Version : String
Database : String

Public Methods
- CallAPI(URL)
- GenerateURL(Latitude, Longitude, Accuracy)

Cache

Data
cachedData : Array of <URL, JSON>
ExpireAfter : Int

Public Methods
- DeleteCache()
- UpdateCache(URL, JSON)
- SearchCache(URL)

Sensor

Data
Accuracy : Float

Public Methods
- GetLocation()

Sensor

This class defines the sensor. The accuracy of the sensor in meters is a constant data member.

The only method is as below.

- GetLocation: Returns the current location in latitude and longitude.

Cache

This class defines the cache. The cache stores each URL input-JSON output for the amount of days given by ExpireAfter (e.g., 30 days).

The methods are described below.

- DeleteCache: Deletes items older than ExpireAfter days (must be run every day);
- UpdateCache: Updates the cache with a new URL and JSON combination;
- SearchCache: Searches the cache for a given URL, returning the JSON response if found and NULL otherwise.

WebClient

This class defines the web client. As the Venue data in the Foursquare database are public, it is possible to do userless Venue searches. Authentication, therefore, based on user credentials is not required and client ID, client secret, and version parameters can be used instead to access the data. Five thousand userless requests per hour are allowed for the whole application. This class encapsulates Client_ID, Client_Secret, Version, and Database parameters common among all users.

The methods are described below.

- CallAPI: Makes an HTTP request using the given URL;
- GenerateURL: Returns the string encoding the URL for the venue data at a particular latitude, longitude, and accuracy.

User

This class defines the user. The user has Sensor, WebClient, and Cache objects.

The methods are described below:

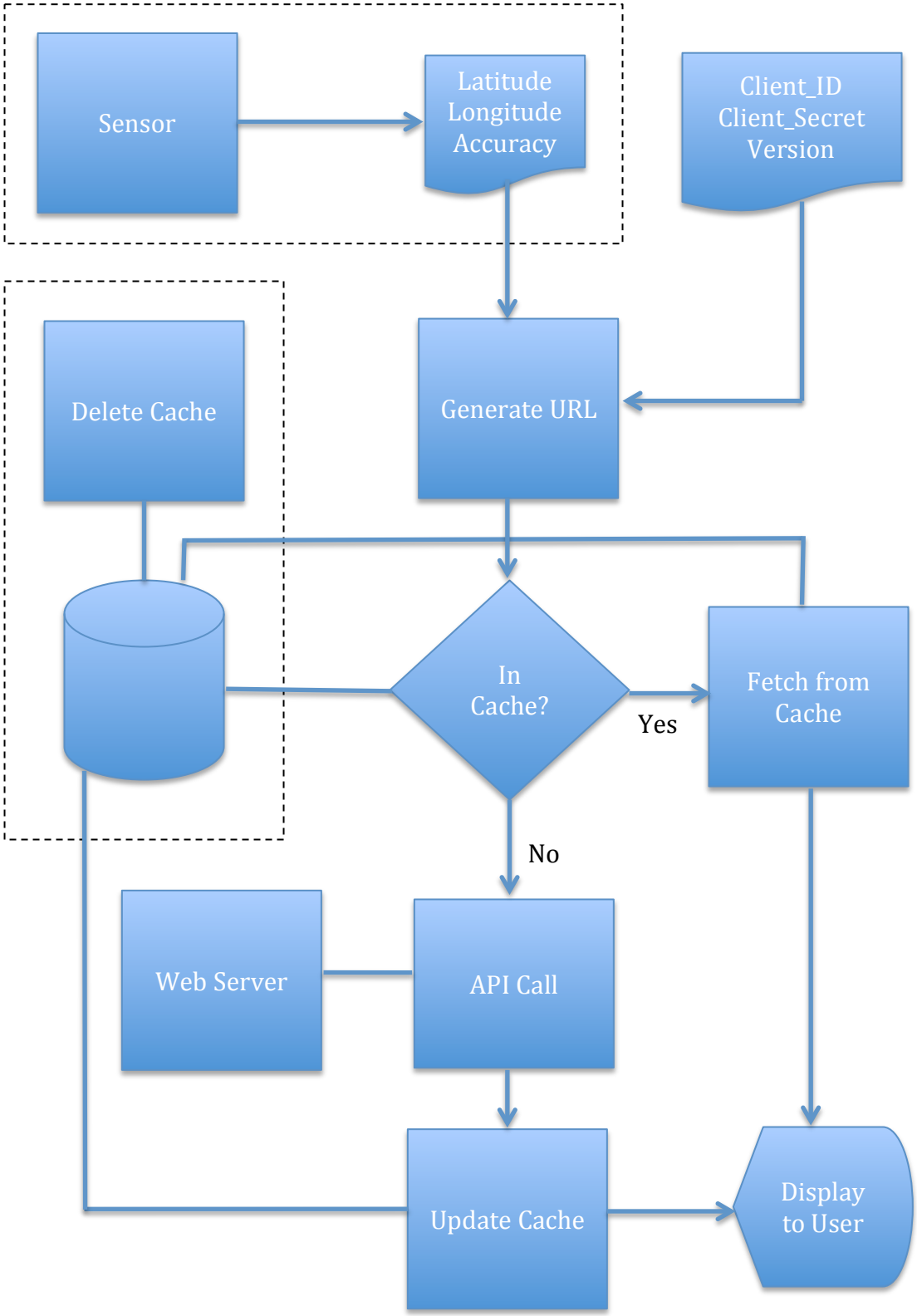
- DisplayInfo: Display the Venue information to the screen;
- ParseJSON: Parses the JSON response returned from the client.

Basic Execution

The application has already requested the Client_ID, Client_Secret, and Version parameters from the server. Using the latitude, longitude, and accuracy fix it receives from the GPS sensor (e.g., 40.66 N, 73.99 W, e.g., 5 m), it generates the appropriate URL to access the Venues data. Then a check is made to see if the URL is already stored in cache. If so, the JSON response for that URL is retrieved from cache. If not, an HTTP request on the URL is performed to search the Venues database to generate the JSON response and the URL and response combination are cached. Finally, the JSON string is parsed and the results are displayed to the user.

Workflow

The workflow is shown below.



2.2 Large-scale Data Handling

Think back on the implementations you made in the first section.

Would these change once the data size gets larger? If so, how?

Yes, increasing the data size would require changes in the implementations. This is because as the data get bigger, the computational complexity of the various algorithms involved increases, thereby increasing execution time. For example, generating the data for each file at once, writing the files to disk, and copying files and performing comparisons between them would all take longer. Additionally, the memory usage of the program would increase linearly with the data size.

Assuming the data size does not become so large as to hit the memory and storage capacities of a single machine, various techniques can be used to handle bigger data sizes. To reduce memory usage, the data for each file can be generated in blocks, instead of all at once. Additionally, multiple processes can be launched across the various cores of a machine, reducing runtime. Then the results can be combined at the very end.

How would you handle a data size that becomes too large to hold on a single machine on local disks?

When the data size becomes too large to hold on a single machine, a system consisting of multiple machines like a cloud, a cluster, or a grid may be used to generate, process, and store the data. As the size of each file to generate can be computed in advance based on the parameters of the run, each file can be generated and stored on each machine, or broken up and stored among multiple machines if the file size gets too large. Updating and backing up the dataset would also now occur across multiple machines. Because the tasks the machines would perform are independent, they should be parallelizable.

Would there be libraries or tools that you could use to help out? If so, which ones and how could they help?

Frameworks like Hadoop and Spark can be used to perform large-scale data processing. When one does not have the compute resources already on-hand, cloud-based services like Amazon Web Services can be used to process and store the data using these frameworks. Hadoop is built on the Hadoop Distributed File System (HDFS) in which big files are broken up and spread across multiple nodes, replicating the data three times and providing a name node to properly reference the data. The MapReduce algorithm performs parallel computation on files stored in HDFS and aggregates the results. Spark is similar, but uses in-memory computing, which reduces file read and writes.

In my own work at the University of Chicago, I rely heavily on the Swift parallel programming language and the Midway compute cluster at the Research Computing Center. Midway uses a General Parallel File System in which files are broken up into blocks and distributed across multiple filesystem nodes. To generate a master dataset, I would launch a Swift process that launched worker nodes to generate each file or block of each file in parallel.