Rochester Institute of Technology

Department of Computer Science

Data Broker for Collaborative Sensor Platform

Salil Raghunath Rajadhyaksha

Advisor: Prof. Leon Reznik
Table of Contents

1. Abstract .......................................................................................................................... 3
2. Introduction ......................................................................................................................... 3
3. Background ........................................................................................................................ 4
4. Architectural Considerations .............................................................................................. 6
5. Implementation .................................................................................................................. 9
6. Experiment Results .......................................................................................................... 4
7. Analysis of Response time ............................................................................................... 25
8. Conclusion ........................................................................................................................ 27
9. Future Work ...................................................................................................................... 28
10. References ....................................................................................................................... 29
1. **Abstract:**

Smartphones could possibly be considered to be the invention of the century. There are currently roughly 2 billion smartphone [1] users in the world and the numbers are growing exponentially by each passing day. These smartphones have been mounted with a number of sensors like the GPS sensor, pedometer, accelerometer and barometer. Research is currently under way to include sensors to monitor air quality levels, heart rate monitor into our smartphone. Scientists from the United States Geological Survey have published a study stating how the smartphone sensor data could be used to create earthquake warning systems. Google is currently working on a modular phone that would primarily be a barebones motherboard which could be configured with sensors of the users choosing. All these resources currently available could be combined and be put to good use for the betterment of the society or by the research community.

2. **Introduction:**

The primary goal of the project was finding an effective way such that a large number of smartphone users can share their data effectively.

The primary hurdle in coming up with crowd-sourced data is convincing a large majority of the smartphone users to volunteer as data sources. To be able to do so the user’s primary concerns need to addressed. One of the primary concerns of the users would be the authenticity of the entity with which it is sharing its data. The user would particularly be concerned about how the data that it is transmitting is handled. This is where a centralized registry/database repository comes into the picture. The average smartphone user would be much more comfortable in sharing data with a central trusted point of contact.

The basis of the project is the development of a centralized data management and facilitating system for real-time sensor data called the data broker. The data broker would act as a centralized registry/repository for real-time sensor data for mobile devices.

Smartphone users willing to be sponsors of data would be registered in the data registry (data broker). Now, anybody needing access to data would send a request to the data broker. The data broker would first check its local repository to verify if the users request could be satisfied, in which case the data broker would return the result back to the requester. If the data broker fails to find the data of the requisite data quality in its local repository, it would query one of the registered devices in its registry and return the result.
Data Quality of the sensor Data:

One of the major factors that governs this data usage is the quality of the data that the user is expecting. The data received by the data broker could be assigned a number of data quality parameters. A few examples of Data Quality parameters are:

- Timestamp (How old the data is)
- The sensor quality (Measuring this is out of scope of this project)

While specifying the request to the data broker a data requester would also specify the data quality that it expects. For example the user could specify a condition that he would like to know the temperature at a given location (passing the latitude and longitude as parameters). The broker would then have to consider the request and try to deliver data as close as possible to the desired data quality.

3. **Background**

There is already some work that has been carried out in this field. There is project that has been developed by the name “sensibility testbed” \(^3\) that works at sharing of resources and data from the smartphones.

During the analysis of its communication model, we figured a few shortcomings that could possibly hamper larger participation in the registry for sharing of data. Also, the new proposed quality parameters had not been accounted for as a part of the said system. The communication workflow is also non-ideal in case of data sharing. It gives too much authority to the data requester and does not account for important factors like end user isolation, user anonymity etc.

All these factors could greatly hamper the large scale participation of users towards the database. While working with a system that primarily relies on crowd participation, the concerns of the participating parties need to be given the highest priority and need to be accounted for. This will ensure larger participation of the users which in turn would increase the availability of data in the broker.
As can be seen from the diagram if a data requester wishes to fetch data, he first contacts the server, the server looks up the registry for a suitable device and then passes the control of that device to the data requester for future communication and data exchange.

Although this architecture works perfectly fine for cases when resources like CPU cycles are to be shared, it is a bit of a risky preposition when it comes to sharing data. This is because the data sources (in this case cell phone users) might be a little skeptical about having its cell phone accessed by a third party program. Also the source in many cases would like its identity to be hidden from the data users.

The above system is the only possible alternative when resources are to be shared as the end user would need control of the donor’s phone. But, for the purpose of exchanging sensor data, the above system seems to be overkill. We could certainly do away with passing the control to the data requester. This would it turn help us gain the trust of more users and increase their participation toward the broker.

Thus it can be concluded that the above architecture poses several questions and is a possible concern when it comes to fetching crowd funded data.
4. **Architectural Considerations:**

There are a couple of architectures that we considered for development of the sensibility data broker

- **Building Data Broker on top of the test-bed.**
  One architecture under consideration was building the data broker on top of the sensibility testbed. This would allow us to make use of the test bed server. The control of a suitable device would be passed on to our broker rather than being passed on to the data requester.

![Figure 2: Architecture of building the data broker on top of the sensibility test bed.](image)

The major advantage of the above architecture would have been the ability to reuse and leverage the existing architecture. However using the architecture would have added an extra redundant server in the middle. There by increasing the call chain. This in turn would affect the response time of the server Following would have been the communication chain if the above architecture was to be used.
1. Data source registers with the clearing house server (Refer figure 1 for clearing house server).

2. A data requester communicates its request to the broker.

3. The broker in-turn communicates its request to the clearing house server.

4. The clearing house server finds and appropriate device and passes it control to the broker.

5. The broker now fetches the required data and completes the request by returning the data to the requester.

Although this architecture, does provide a work around to the initial problem where the requester had direct access to the device it introduces a redundancy by having two servers.

If the broker was to be implemented using this architecture, the requester would need to make a request to the broker which in turn would route the request to the testbed’s server, this inter server communication would take up some precious time.

While working with API’s that need to process a thousands of requests in a day any sort of delay in terms of response time is highly undesirable. Another disadvantage of doing so would be that the broker would need to be at pace with the other server which is also an undesirable property.

Lastly, as a future step, we would like to replicate the broker on multiple servers so as to have to increase its availability and to avoid factors like single point of failure which are essential when it comes to dealing with servers. As I would discuss later, the architecture of the broker has been designed in such a way as to make it easy for the server’s to be replicated.

But if the above architecture was to be used, the availability of the server would still be hindered as the replication of the other server is beyond our control.

Therefore after careful considerations of the pros and cons and after due consultations with the professor, it was decided to do away with the architectural style and look for another alternative which did not involve server duplication and redundancy.
Independent broker and Data Source.

Execution flow in the above architecture:

1. A device willing to volunteer data registers itself with the data broker.

2. The device than runs in the background waiting for any possible communication from the broker.

3. A data requester passes on the request to the broker.

4. If the broker can satisfy the request with the help of locally available data, it responds to the request.

5. If the broker cannot find a suitable device in its repository, it contacts one of the devices from the repository and responds to the request.

The above architecture greatly reduces the call chain as there is just one server where all the communication is handled. This in turn would reduce the response time.
The only shortcoming of the above architecture was that it requires the implementation of the data source platform too and it also required developing the communication framework between the source and the broker.

However after the advantages that this architecture provided far outweighed its shortcomings. It would reduce the call chain drastically, thereby reducing the response time. Also it would not hinder the replication of servers. Thus this architecture would overcome all the shortcomings that the earlier architecture was causing. Hence, it was decided to develop the data broker using this architecture.

5. Implementation

- The Data Source

![Figure 4. Data Source Android application screenshot.](image)
The data source was developed as an Android application. People willing to donate their sensor data would install this application on their cellphone.

Following were the features of the application:

- Upon startup, it would register itself with the broker passing information like its sensor readings, its location (these have been randomized for the purpose of this project to give the effect of having phones located at multiple locations around the globe) and its data quality parameters (these have been set arbitrarily as quality evaluation was beyond the scope of the project)
- The application would then run in the background waiting for any further requests from the broker.
- If the broker communicates to a particular phone the need for data, it is duly passed on the broker.

There were a couple of options available for maintaining a background listening protocol between the source and the broker. One possible option was the use of a web server on the android phone. This would have drained the battery which was undesirable. The current protocol has been implemented using sockets. Sockets while listening for requests give all the system resources. This would in turn would reduce battery consumption at the source.

Steps to start the data source application:

- Click on the BrokerClient icon.
- Enter the ip address of the device and the broker

There are primarily two communications that are happening currently.

- One is upstream message passing from the source to the data broker. This has been handled using Rest Api calls.
- The other form of communication is the notification from the broker to the phone to resend its data. This takes place using sockets.

The application was developed for Android Kitkat and later versions. For the purpose of experiments the application was run on 1 Motorola X2 and 3 Motorola X devices.

One important thing to note here is that the locations of these devices were simulated so as to give us the effect of having devices located at multiple geographic locations.

**Technologies used:** Java.
The Data Broker

The data broker application forms the major crux of this project. It was developed as a web service so that it would be globally accessible.

Following are the steps that the broker follows:

- **Step 1**: Volunteers willing to supply data to the broker register themselves with the data broker.
- **Step 2**: A data requester communicates its request to the broker specifying the location whose data it is interested in, the sensor information it needs and the quality of data it is expecting.
Step 3(a): The broker checks its local repository for suitable sensor data with the requested data quality. If data is present locally, it is promptly returned to the requester.

Step 3(b): If the broker fails to find suitable data in its repository but finds a source in its registry that could be accessed to satisfy this request, it contacts the source.

Step 4: After the source returns the necessary information the broker completes the request.

The design paradigm followed for the same was the layered architecture. This was done to decouple the functionality for easy further scaling that may be needed.

Following were the layers and its functionality:

a. The Service Layer

The service layer was built using Java’s Springboot\(^4\) technology. It exposed a set of api’s that could be accessed as http calls.

The web service had been deployed on a tomcat instance.

Following were the API calls and its parameters

- @RequestMapping(value="/registerDevice", method=RequestMethod.POST, consumes="text/plain")
  public String registerDevice( @RequestBody String userDetails) {

  This method is called by the source when it has to register with the broker.
  The field user details contained the details of the device that registered itself with the broker. The parameters included:
  - The device location
  - The data Quality
  - The sensor information that is present.
  - And information about contacting the data source.
This method is invoked by the data user when it has to request some data from the broker.

The field query parameters indicates the parameters that the user had to send along with the request. These parameters were:

- The sensor information needed
- The location from where the data is needed
- The quality of data that is expected
- The radius within which the broker should look for the data. This field was optional and if the user did not provide these details, the broker would default the search range to 140 miles.

This method is invoked by the data source when it has to post some data to the broker.

The data parameters would include the sensor information of that particular source.

The service layer has been decoupled in such a way that it can easily be replaced by some other protocol (like Soap) in the future if the need arises. There is no form of computation that takes place at the service layer.

Let us consider the getSensorInformation. The service layer is only responsible for fetching the request. After the request arrives here, it is passed on to the Business logic layer. It is here that the checks like malformed queries are done. The business layer than processes the request with the help of the underlying layer and returns the result back to the service layer which then passes it back as an HTTP response. As can be seen from the above illustration, the sole purpose of the service layer is to service the request. In the future, based on the change in requirements this layer could easily be replaced by other technologies.
b. The Business Layer

All requests from the service layer are passed onto the business layer where the processing of the request takes place.

Functionalities of the business layer include:

1. Parsing the request parameters.
2. Checking if the request has been well formed.
3. Preparing a Mongo Db document to be inserted into the database.
4. Parsing the result returned by the data layer and preparing the correct format to be returned to the requester.
5. Checking if the data passed by the data layer is of the desired data quality.
6. If the data is of the desired data quality, passing it back to the data requester.
7. If the data is of a lower data quality, contacting a suitable device.

Let us consider the example of the get sensor information method. After receiving the request, the service layer, it is passes on to the business layer. Here the query is checked and if it is malformed a suitable response is passed onto the requester. Otherwise, the business layer prepares a Mongo DB query and passes the query to the data layer. If the data layer passes a result based on the query, the result is formatted and passed to the service layer. If the data layer does not return a data, the business layer then requests the data layer to check the registry for a suitable device. If a device is found, the business layer contacts it, else a failure notice is passes onto the service layer.

Any change to policy of the broker or any additional checks that need to be done will need to be included in this layer. There are further features like user authentication that could be included. The logic for the same would need to be included in the business layer. Another possible implementation could be token based access where a authenticated user is given a token and it has to use this for all further communication. If the token expires, the user would need to re-authenticate and fetch a new token. The logic for that too could be included in this layer.
c. The Data Layer

The data layer handles all the interactions with the database. Requests are passed on to the data layer from the business layer.

Mongo labs, the cloud based service of Mongo DB has been used has the persistence layer. This provides for easy scaling up in the future. Since the nature of the data was largely unstructured, the idea of using a NO SQL database was finalized.

Functionalities of the data layer include:

1. Posting a new document for a new device to the database.
2. Updating information about a device.
3. Finding a suitable device from the database.
4. Posting data passed by the business layer to the database.

There are two methodologies that could be used to scale this application.

- We could have an instance each for each instance of the service that you deploy. We could then have a daemon process that runs in the background and synchronize the multiple instances of the database. The disadvantage of this is the overhead of the background thread that would keep running in the background. However this methodology would not include any monetary expenditure.
- The other methodology would include leveraging the features provided by Mongo Labs to have a fault tolerant database. However this would involve monetary expenditure.

```java
public MongoDatabase getDatabase()
{
    MongoURI uri = new MongoURI("mongodb://user:password@db01:27017/db02:30018"); //String that specifies the database to be connected to
    MongoClient client = new MongoClient(uri);
    MongoDatabase database = client.getDatabase("dataset");

    return database;
}
```

Figure 6: Code Snippet to connect to the database.
d) The Background Quality Manager Thread

The background thread runs in the background and checks the data quality of the data in the database. It particularly looks for the timestamp on the database and if the data is deemed to be of poor quality the manager communicates with the device and proactively fetches the data.

As we would see later from the experiment results, communicating with a device to fetch data is an expensive process and in the ideal scenario, the broker should satisfy each request based on data from its local repository. To do this, the data quality in the broker needs to be maintained. This is where the Quality manager thread comes into the picture. There are certain enhancements that can be done to this layer as a part of future work. Basically, the more sophisticated the daemon thread is, better would be the turnaround time of the broker.

Communication Model:

![Communication model of the Data Broker](image)

Figure 5: Communication model of the Data Broker
• Android Data Requesting Application

This application was developed as medium for testing the data broker.

![Android Data Requester Application](image)

**Figure 7. Data Requester Android Application.**

Following are the steps to use the data requester application:

1. Start the application
2. Specify the brokers address.
3. Specify the query parameters.

```java
protected Void doInBackground(Void... urls) {

    String response = "";
    URL url;
    try {
        String[] q = Query.split("/");
        String u = "http://" + DeviceIP + ":8080/getSensorInformation?requestParameters=" + Query;
        System.out.println(u);
        url = new URL(u);
        con = (HttpURLConnection) url.openConnection();
        con.setRequestMethod("GET");
        con.setUseCaches(false);
        con.connect();
        BufferedReader rd = new BufferedReader(new InputStreamReader(con.getInputStream()));
        String line;
        String result="";
        while ((line = rd.readLine()) != null) {
            result+=line;
        }
        rd.close();
        System.err.println(result);
        QueryResult.setString(result);
        result+=result;
    } catch (IOException e) {
        e.printStackTrace();
    }
```

**Figure 8: Code snippet from Data Requester. Module that makes request to broker.**

The primary purpose and target users of the broker would be other websites that could consume the web service. The broker architecture and response format make it easier for the broker to be consumed as web service. The Android application that was developed for testing the service was primarily for illustration purpose only.

Other means of communicating with the web service is the use of a browser. To do so, the requester would need to type the request URL in the URL bar and the request parameters would need to be comma separated (The request format would be clearer from the next section).

Since the response is a plain JSON. It would not be formatted as a webpage since that is not the primary purpose of the application.
In the following section I will go over some the request formats that are possible with the data broker and the response time that was reported for them while carrying out the experiments.

5. **Experimental Results**

The tests were carried out primarily using Postman[^5] which is a google chrome plugin that can be used to test RESTful web services.

**Following are a various types of requests that the broker handles**

a) A requester requesting single sensor information without passing any additional parameters.
   This is the scenario where the broker is able to find a match in its local repository. As a result of this the turnaround time of the request is pretty quick. Also since the user does not specify any radius the broker looks for a suitable device in the radius of 140 miles.  
   Sample Query:  
   The request parameters are: 
   <Sensor Info needed>,<DQ>,<LAT>,<LONG>  
   As it can be seen that the response time in this case is roughly 900 msec. In all the experiments carried out, these requests were processed with the least response time as the broker fetched the data locally rather than having it retrieved from the data source.
Figure 8: Query type 1 for data Broker.

b) A client requests for a single sensor data and specifies a given radius and the broker is able to satisfy this request using data from its own repository (No contact with source needed)

Sample Query:


The request parameters are: <Sensor Info needed>, <DQ>, <LAT>, <LONG>, <RadiusInMiles>

Response: JSON containing the requested information.

In the above scenario, the user has specified a radius within which range it expects the data to be fetched from. If this parameter is not specified the broker defaults it and searched in predefined range.

Figure 9. Query type 2 for Data broker

Note: If no source is present in the given radius, the broker informs the client about the same.
Figure 10. Query type 2(b) for Data Broker

c) The third type of request is one in which the client requests for a sensor information with the best data quality. In this case the broker scans through its registry and returns the data from the source that has the highest data Quality. DQ value greater than 10 is considered to be request for best data. As it can be noticed, the response time in this case is slightly higher. This is because the broker needs to find the best possible data.

The request parameters would be:
<Sensor Info needed>,<DQ >=10>,<LAT>,<LONG>,<RadiusInMiles(optional)>

The priority of the data requester while making such type of requests is that it needs access to data with the best data quality possible. Therefore when such a request is made it is obvious that the periphery parameter within which the broker would need to look for data needs to be relaxed a little. The broker, in this case doubles the search area within which to look for a suitable device.
d) The 4\textsuperscript{th} type of query is one in which the broker is unable to find data with suitable data quality in its local repository but there is a device in the said locality which can be contacted to fetch the needed data. The response time in this case is higher as the source needs to be contacted and this is an expensive operation.

Such requests are the most expensive and require the highest response time. As it can be seen from the screenshot below, the response time in this case is almost three times the response time for requests that are processed locally.

The broker does provide methodologies like the background thread that works towards minimizing the need to contact the source on demand.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure11.png}
\caption{Query type 3 for data broker.}
\end{figure}
The user could also request for multiple sensor information in the same request. More often than not the request would need information about multiple sensors. For example if there is a website that displays weather information and if it is consuming the broker web service, it would need access to all the weather related information that might be present with the broker.

The broker provides a mechanism to make request for multiple sensors. The query format in that case would be:

\(<\text{Sensor Info needed}>,<\text{DQ}>,<\text{LAT}>,<\text{LONG}>,<\text{RadiusInMiles}>\)

The sensor information parameter in this case would be colon separated where each value would indicate a sensor.

Eg if the user wants temperature, signal strength and AirQuality, this field would be denoted as:

Temperature:SignalStrength:AirQuality

The miles field is optional in these cases too.

Sample Query:

This is a type of request when the user requests for multiple sensor information but one of the sensor information is not present. The broker can handle these cases too. In the event that only partial data is present, the broker informs the client that information about one particular sensor is not present. If all the requested sensor information is present, the broker will return the same.

Let us consider the case where a data requester has requested for three sensor information and the broker is in a position to facilitate only two of the three sensor information. The broker in this case would return the two sensor information and pass a notification indicating the third sensor information wasn’t available.
g) If the query is not well formed, the requester is informed about the same.

![Response to a malformed Query](image)

**Figure 15:** Response to a malformed Query.

h) If the broker does not find local data that could match the user’s requirement and also fails to find any device that could satisfy this request, an informational message is conveyed to requester.

An important point to be noted here is that the broker tries to find some partial data if it can before informing the requester about its inability to respond to requests.

![Response to a failed request](image)

**Figure 16:** Response to a failed request.
7. Analysis of Response Time

As it can be seen from the above graphs, in both cases, i.e. when single sensor information is requested or when multiple sensor information is requested, the fastest response time is in cases when the data is to be fetched locally. The response time is the maximum in cases where the data source is contacted as this is an expensive process.
8. **Conclusion**

- The broker was deployed on a Tomcat instance and the data source Android application was installed on four Android phones for the purpose of these experiments.

- The experiment results proved that the turnaround time increased drastically in cases where the data had to be fetched from the server.

- It can thus be concluded that the efficiency of the broker depends greatly on its capability to maintain optimum data quality in its local repository which in turn would enable it to respond to a greater number of queries without the need to contact any data source.

- This functionality is currently being implemented with the Data Manager Daemon thread. This thread can be further sophisticated by adding features like machine learning wherein the broker would learn from previous requests and become more proactive than in the current scenario.
9. **Future Work**

- Developing the data source for IOS platform is one of the possible future works of the project. This would help increasing the participation of smartphone users in the registry.

- The fault tolerance of the broker can be increased by replicating the server. This would negate the single point of failure factor. The architecture of the broker has been developed using layered paradigm, this would make replicating the server pretty simple. Also the data layer is on the cloud. This would further facilitate the server replication.

- The fault tolerance of the data layer could also be improved by replicating the database either by using the facilities that Mongo DB provides internally or manually replicating the database servers.

- The replication of the database and the broker would both not need much effort because of the layered architecture that has been used to develop it.
10. References

1. Statistics of smartphone users courtesy -  


5. The broker was tested using-  
   [https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomp](https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomp)