Litterbug – A Software Approach to Littering

DAVID MEDNIKOV, Oregon State University
HITESH VARMA, Oregon State University
KEVIN ALLEN, Oregon State University
ALEXANDER YFRAIMOV, Oregon State University

1. DATAFLOW DIAGRAMS

1.1 Repository Architecture

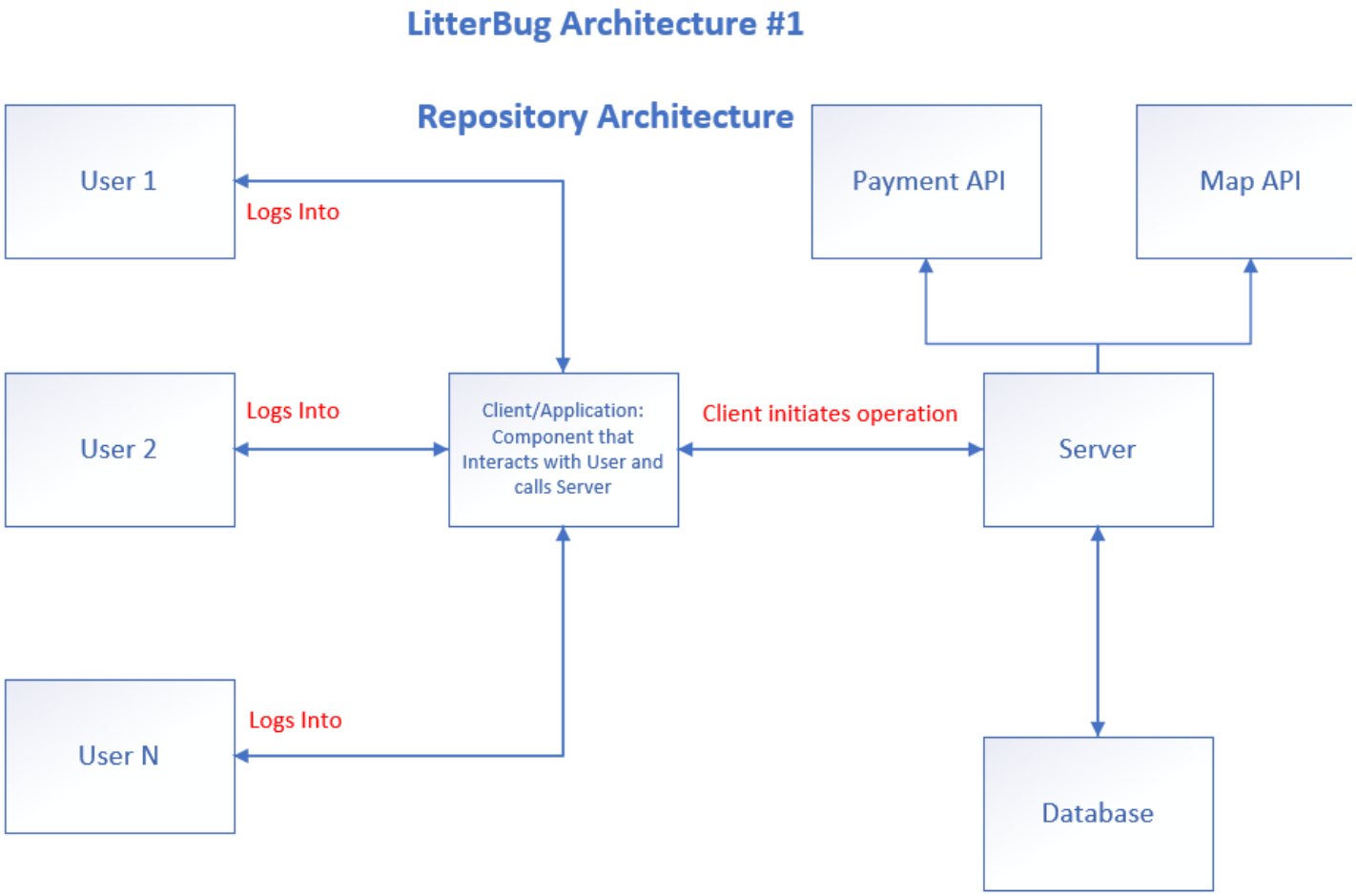

Fig. 1. Repository Architecture Diagram

This work is supported by Michael Johnson of Oregon State University.

1.2 Blackboard Architecture

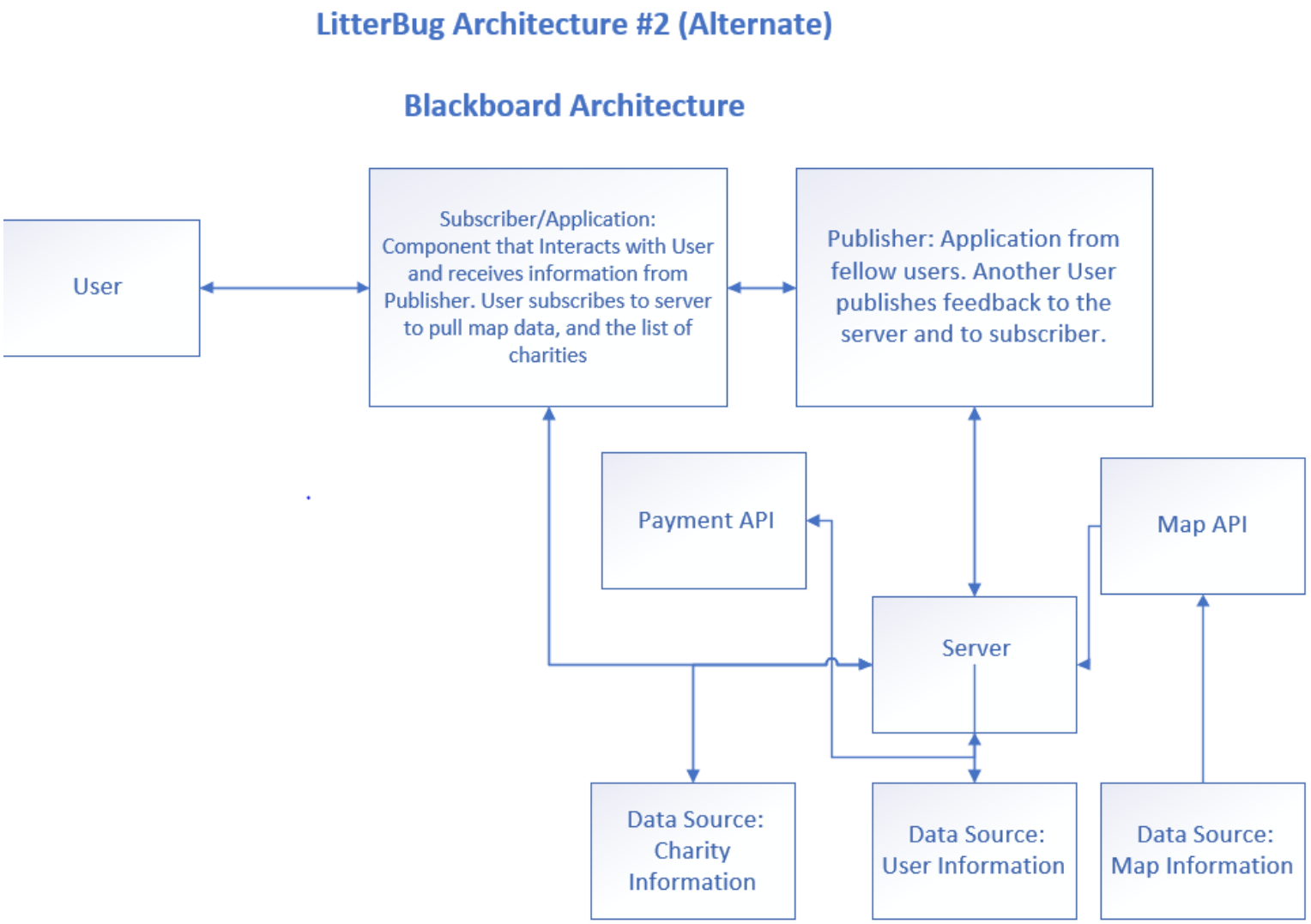

Fig. 2. Blackboard Architecture Diagram

2. KEY QUALITY ATTRIBUTES

2.1 Selected Attributes

- **Reliability** - It is essential that the map and payment servers are always available as they will be interacting with external APIs. If the API servers fail then the main uses of the app fail.
- **Efficiency** - This app will require a lot of refreshing maps. This can use a lot of data. Instead of constantly sending map imagery, send vector graphics data so that maps are rendered on the phone.
- **Usability** - The app should be easy to use and understand. We use a tabbed menu so that the layout is simple and intuitive. Each screen only serves one purpose.
- **Portability** - The app will need to run on both iOS and Android, but built so that porting to Alexa Fire, Windows Phone, or some other mobile OS is not extremely difficult.

2.2 Repository Architecture Support for Quality Attributes

- **Reliability** - As long as the server is running, it should be able to successfully pull att data from our database and ping the payment processing and maps APIs that we utilize for 3rd-party data. One server handles both directions of data transfer.
- **Efficiency** - Since one server handles all requests and responses, all data processing must be routed through one tunnel. This may slow down speed and make it less efficient. However, not retrieving actual maps data from the server helps.
- **Usability** - the architecture should not make much of an impact here, as long as the servers and APIs are reliable the app experience will not change.
- **Portability** - Similar to usability above, architecture should not have a large effect. Devices with poor data reception or location services may experience poorer performance.

2.3 Blackboard Architecture Support for Quality Attributes

- **Reliability** - As there are multiple parties involved in each data transfer (as opposed to each user having a direct path with only the server) if one of the users (publisher/subscriber) is having technical issues it will affect other users as well. For example, if the publisher is unable to save to the server, but data does get pushed to the subscriber, they will receive data that does not accurately reflect what's on the server.
- **Efficiency** - Since some data can be pushed directly to the subscriber from the publisher, this architecture is more efficient. It does not always rely on the server to process and route data transfer, so multiple things can happen simultaneously.
- **Usability** - the architecture should not make much of an impact here, as long as the servers and APIs are reliable the app experience will not change
- **Portability** - Similar to usability above, architecture should not have a large effect. Devices with poor data reception or location services may experience poorer performance.

3. FAILURE MODES

3.1 Failure Mode 1

Maps server pulls from Google Maps API. If Google Maps API is down, our map server is unable to pull map data and the map in the app will not load.

FAULT TREE

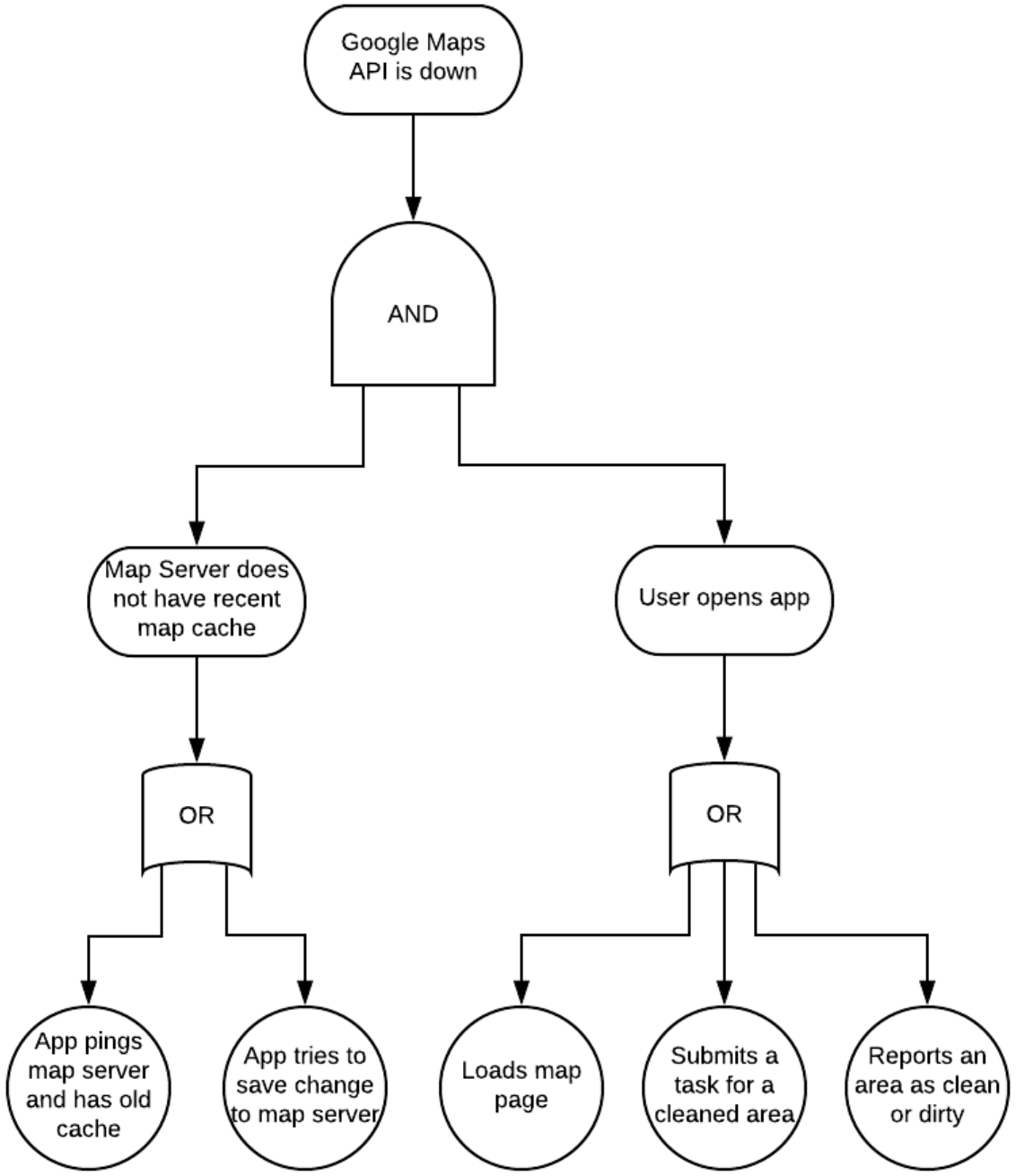

Fig. 3. Failure Mode 1 Diagram Fault Tree

3.2 Failure Mode 2

Payment Processing Server is down and cannot receive payments by Credit Card or send requests to PayPal and Google/Apple Pay.

FAULT TREE

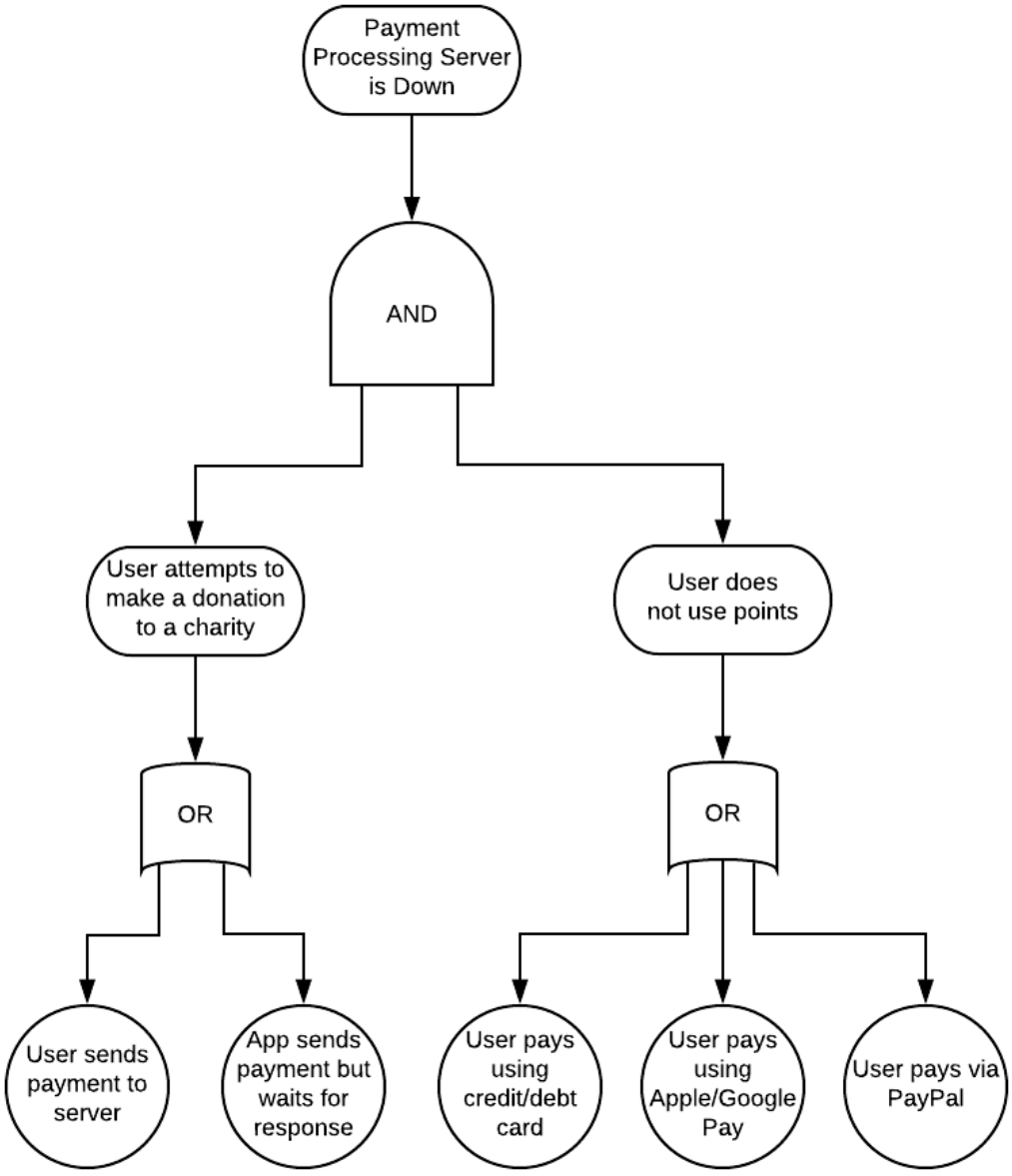

Fig. 4. Failure Mode 2 Diagram Fault Tree

3.3 Which Architecture is more prone to failure?

WHICH?

I believe that the blackboard architecture is more susceptible to failure. Because both of these fault trees involve contacting an API in some way, if that connection fails at any point, the entire transaction (whether that be an actual money transaction, or simply a map load request and response) will fail.

WHY?

The blackboard architecture is more susceptible to failure because there are more moving parts per transaction. The publisher may have an issue sending data to the server and/or the subscriber and the subscriber may have an issue receiving data from the server. Since these transactions can happen asynchronously, we create more possibility for there to be an error that occurs.

EXAMPLE OF AN ERROR SPECIFIC TO THIS ARCHITECTURE

An even worse scenario is one where the publisher is able to publish data to the subscriber but gets an error when publishing the same data to the server (example: first user (publisher) challenges another user (subscriber) to clean an area, but there is an error when the challenge is sent to the server. The subscriber has already been notified of the challenge, but when the user goes to view it, it does not appear in the app and the second user is not able to earn reward badges or points for the task. This causes confusion and data loss.

WHY REPOSITORY ARCHITECTURE IS BETTER

If the architecture were a repository architecture, this kind of scenario would be avoided, because the challenge would have to be routed through the server first, so it would fail before it ever reached the second user. While the first user would be notified of an error, it would not cause further errors down the line.

4. DECOMPOSITION OF REPOSITORY ARCHITECTURE

The two important elements we will be decomposing into lower-level diagrams are the server and the client app.

4.1 Lower-Level Diagram for the Server

Data flow diagram for server in Repository architecture

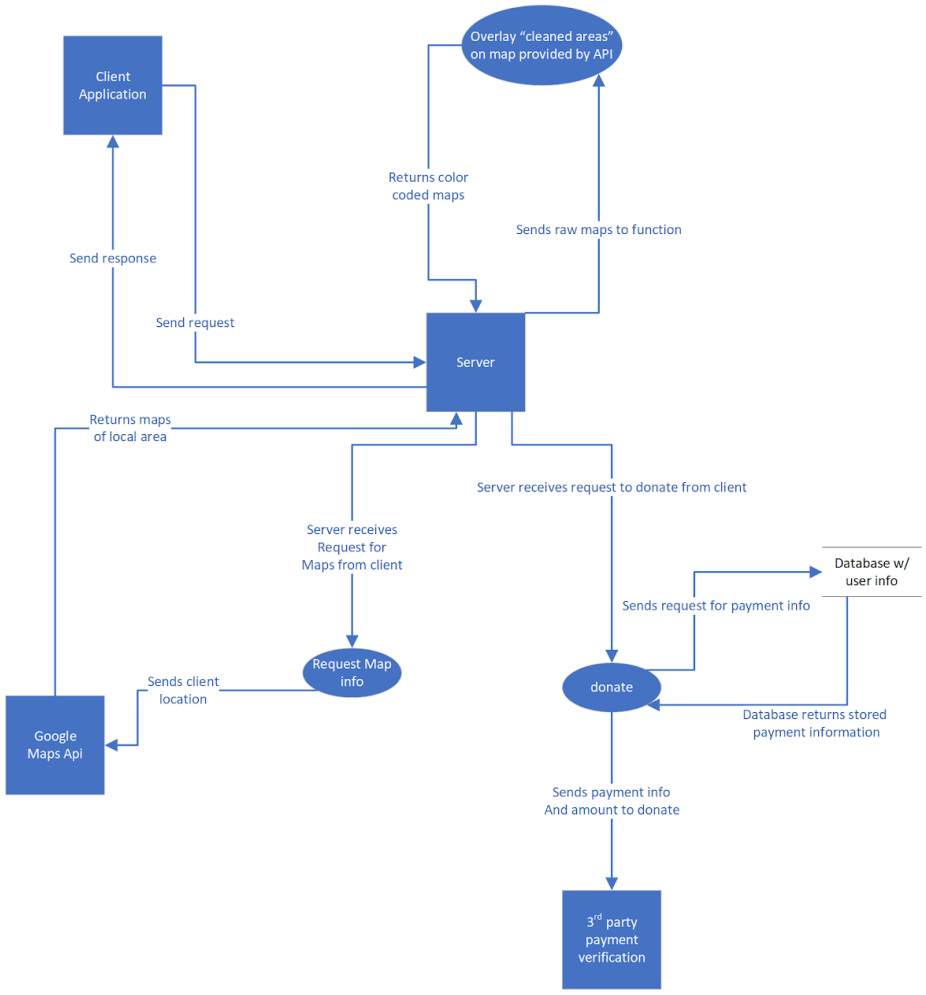

Fig. 5. Server Lower-Level Diagram

4.2 Lower-Level Diagram for the Client Application

Data flow diagram for client application in Repository architecture

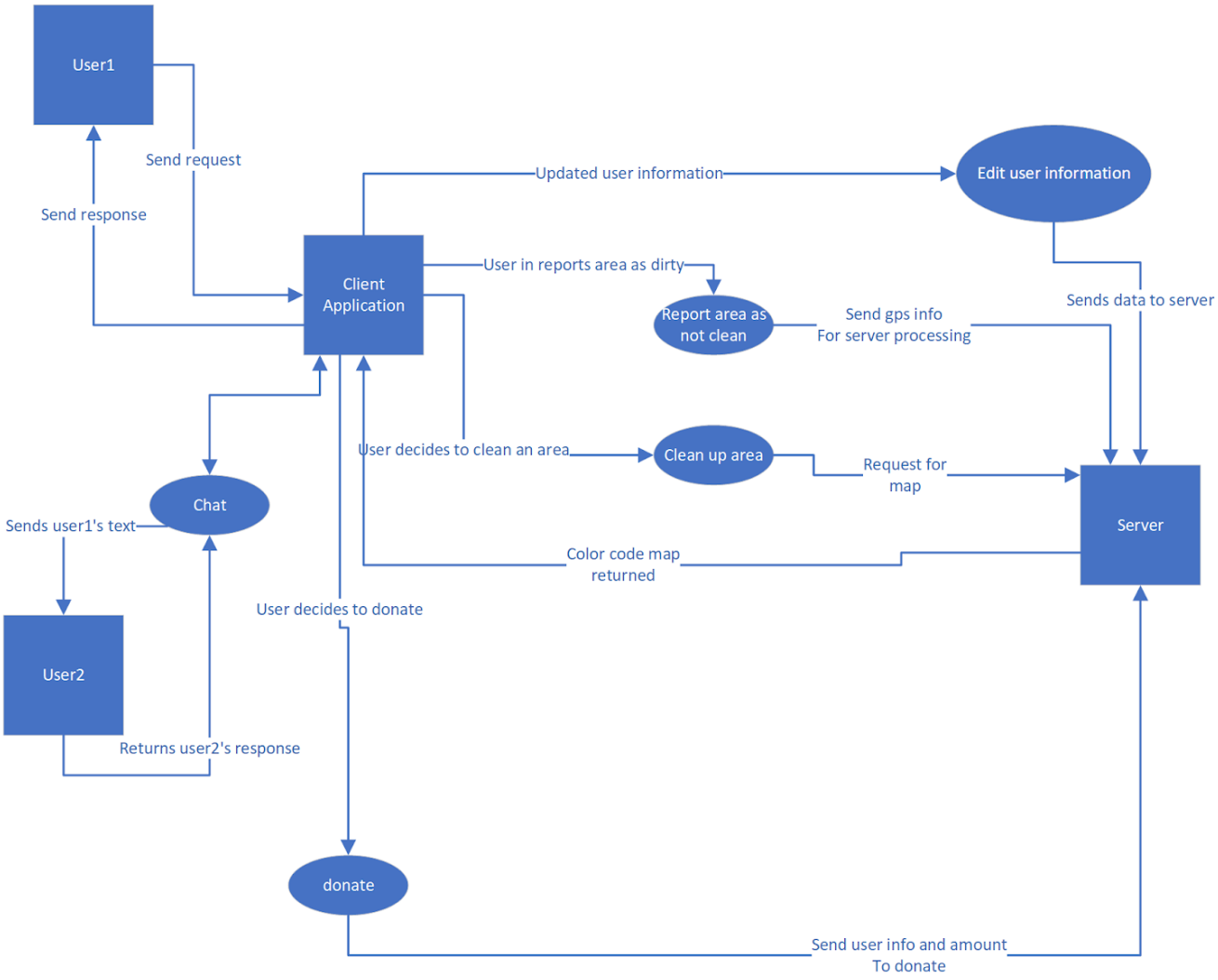

Fig. 6. Client Lower-Level Diagram

5. VALIDATE SELECTED ARCHITECTURE

5.1 Use Case 1

USE CASE 1 SUMMARY

User can use map to see areas near them and (on mobile only) mark them as complete before/during/after they have picked up litter in their area. User can view their points balance and badges once they log in.

VALIDATING USE CASE 1

The first part of this use case consists of the user logging in while outdoors at which point the client sends a request for an updated map to the server. The server upon receiving this request will request map info from the Google Maps API which it then sends to an overlay function which returns a color coded map reflecting the most recent cleanliness status of the area. It will mark the areas which are clean and the areas which have yet to be cleaned.

It then sends this updated color coded map as a response to the client application for the user to view. The user then decides which area to clean, marks this on the map at which point the client application sends a request to the server to update the map. The server upon receipt of this information will wait until another user's client application sends confirmation that the area submitted as cleaned by the previous user is in fact cleaned (More on this in use case 3).

Once that is confirmed the server will return an a map overlaid with color coding which reflects that the area is now clean. Since all clients request updated map data from the same server, the change in the state of the map will be reflected on all client's mobile devices when they travel to that same area which makes the repository architecture a sound choice.

5.2 Use Case 2

USE CASE 2 SUMMARY

User can donate to charities, not only at sign up, but at any point during the life of the app. They will get bonus points and medals for doing so.

VALIDATING USE CASE 2

The user must choose a charity after logging into the app. After completing this, the user chooses a payment method and amount and clicks a button to donate to that particular charity. All of the prior activity happens completely within the client application up to the point right after the donate button is clicked.

At this point the client application sends the user's info and amount to donate to the server. The server in turn sends a request for payment information to a database which returns the stored user's payment information to a third party payment verification system. Upon verification, the server is sent confirmation that payment was successful.

The server calculates the proper reward the user should receive and sends a response to the client application stating that the donation was successful and lists the badges/medals the user received. For security reasons, having the server and a 3rd party payment processor handle payment processing activity as opposed to the client seemed like the correct choice.

5.3 Use Case 3

USE CASE 3 SUMMARY

User verifies that cleaned areas are actually cleaned or not and the app leaves the area green or updates it to red if not clean.

VALIDATING USE CASE 3

Use case 3 deals specifically with limiting the threat of users abusing the system as a means to gain medals which they are not entitled to. An example of this is reporting areas as cleaned which the user has not in fact cleaned. We think the repository architecture will give us a means to combat this possible undermining of the system.

When a user logs into the app and reports they cleaned an area the client application sends this information to the server. The server in turn finds the last user to have cleaned that area and compares the time elapsed between cleanings to see if a reasonable amount of time has passed for the area to become dirty again.

If the timing is too short then the user is flagged for a potential false report. If enough flags are generated for that user they will be issued a warning or removed from the system. Since all submissions from the client application are time stamped and the server receives these data transmissions from all clients, this kind of calculation can be easily done.

6. IMPLICATIONS

6.1 Potential Revisions

One revision we could make to our architecture is to more clearly illustrate how it would handle verification that the areas submitted as cleaned by the users have truly been cleaned. As of right now we have two means of verifying this. The first is through confirmation and/or disconfirmation by other users. The other is purely based on the time elapsed between an area being marked as cleaned by one user and the time it is marked by a subsequent user as being cleaned.

It currently does not take into account if multiple users are working together and cleaning the same area in which case the server would receive two submissions that the same area was cleaned within short proximity of one another. This could result in a user being unjustly flagged for a false submission. This could affect the usability of the app as a false flagging could deter a user from continuing to clean feeling their honesty is being challenged on insufficient grounds.

The server could check the gps coordinates of both users at the time of submission and verify that they were in the same area when they marked an area as clean and use that to ensure a flag would not be issued as one possible solution.

User can use map to see areas near them and (on mobile only) mark them as complete before/during/after they have picked up litter in their area. User can view their points balance and badges once they log in.

6.2 Impact on Quality Attributes

Both of our failure modes are triggered by services we use which are external to the software we are engineering. Both the Google Maps API in failure mode 1 and the 3rd party payment processor in failure mode 2 are external services that our software works in tandem with to deliver a complete functioning system. However, because the proper functioning of the system is in some sense at the mercy of these external systems we are left with the situation that if they fail so do we.

Given this, we could revise our architecture to be able, should the need arise, to fall back on additional third party payment processors in the event of our primary one failing. In the same vein we could consider using Apple Maps API as a fallback if Google Maps API is non responsive although this would be a tremendous undertaking and may not be feasible given time constraints.

The potential for failure however in both failure modes would be greatly reduced. This would result in the reliability of our system increasing substantially because currently if either of these two components fails then the users cannot carry on using the system until it gets resolved.

6.3 Impact on Potential for Failure

Lastly, one choke point in our architecture's efficiency is that all data processing is currently going to a single server. If we revised our architecture to utilize multiple servers that could be brought up on demand it could increase the efficiency of our system.

Utilizing a platform like Amazon Web Services to bring more virtual servers online when demand is high to distribute the processing load (on the weekends during daylight hours when activity will presumably on its peak) and taking them offline as necessary could greatly increase efficiency. Since all servers would be referencing the same database this should not present too much of a hurdle to possibly implement as an architectural revision.

This at the same time would introduce another point of failure relying on external virtual servers however we would run these in tandem with our original server only in the event that our own struggled to handle the load being placed on it.

Even if Amazon's failed, we would not be left in a worse position than had we chose not to use them to begin with so we don't see this as an issue.

7. SUMMARY OF GROUP MEMBER CONTRIBUTIONS

- David Mednikov
  - Identify Key Attributes & Failure Modes, compiling the google doc into the ACM format
- Hitesh Varma
  - Create and share google doc, Data flow diagrams for repository architecture and alternative architecture (pipe & filter)
- Alexander Yfraimov
  - Validate selected architecture and explain the implications
- Kevin Allen
  - Architecture decomposition and lower-level diagrams