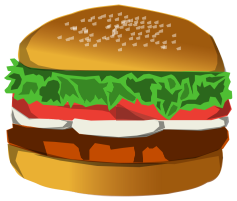
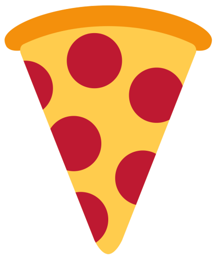


U Eat:

A Dining Concierge App

-Architecture



Group 28

Luke Carter

Ibrahim Mahmoud

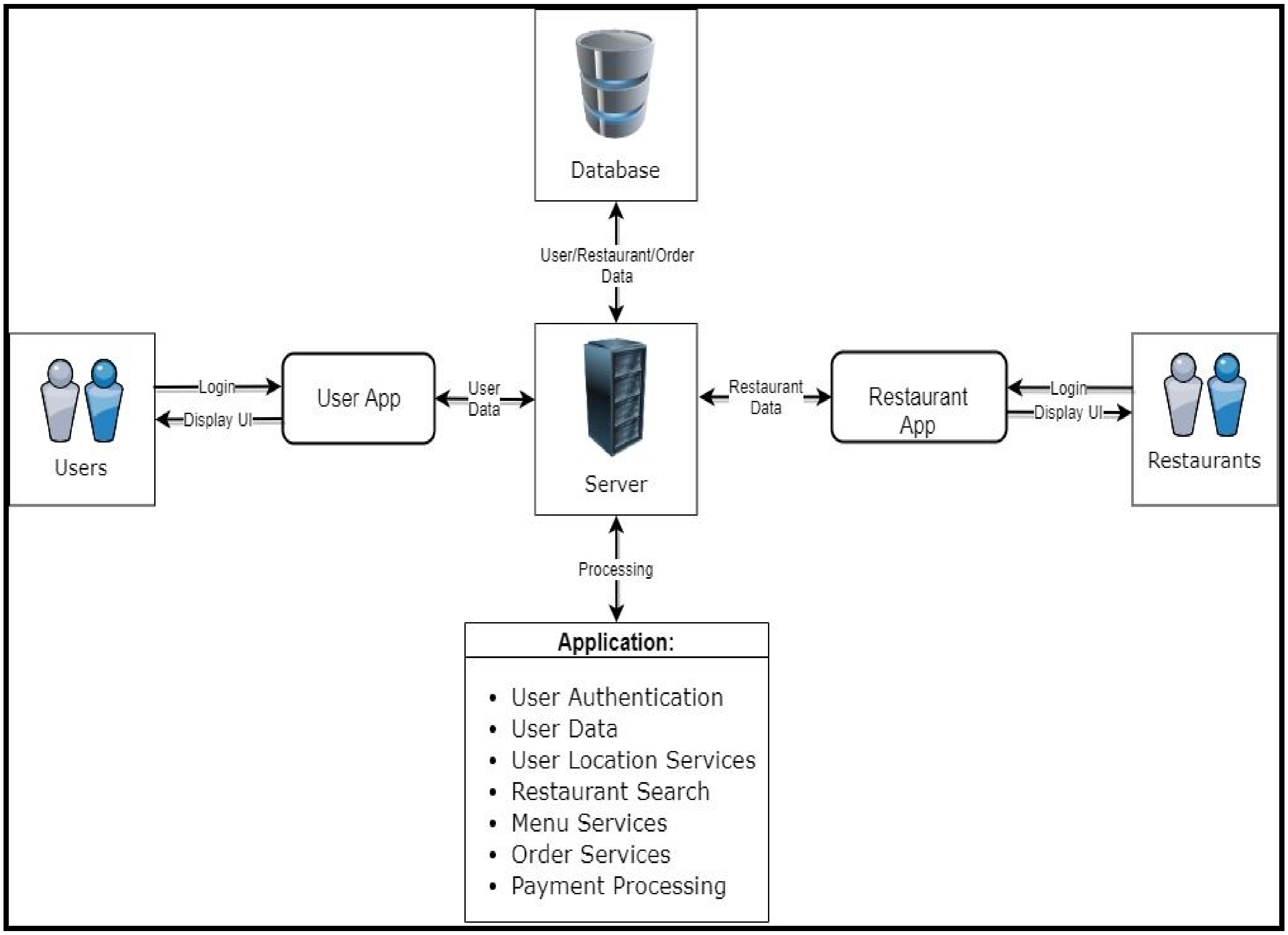
Miles McCoy

Mathew McDade

Timothy Tseng

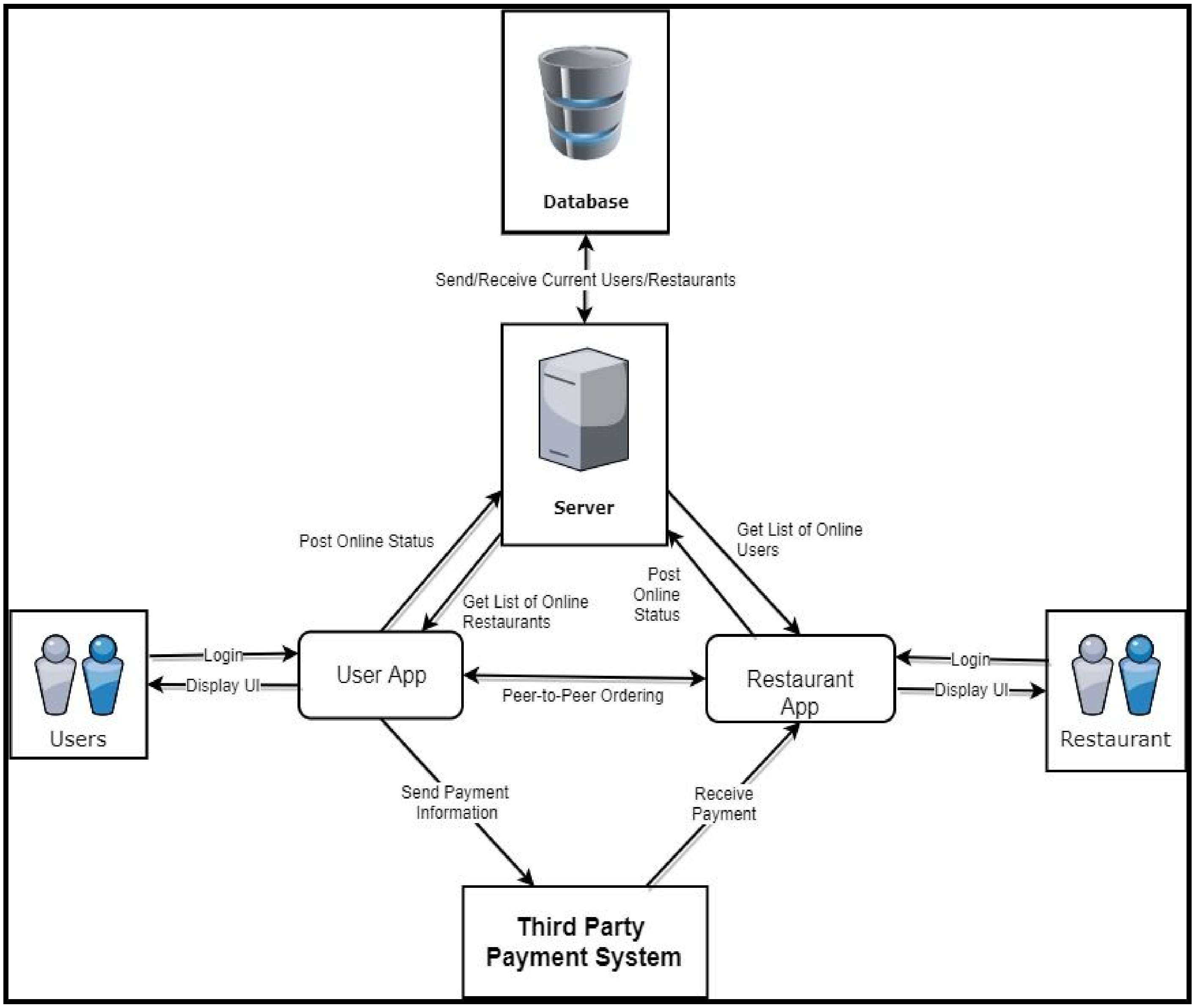
Architecture A:

Client - Server (Server heavy)



Architecture B:

Hybrid Peer-to-Peer Architecture



Quality Attribute Comparison:

System Quality Attributes:

1. **Reliability**
    a. **Architecture A:** A server side heavy architecture will require an active network connection for the user. Additionally, a connection must be maintained with the DataBase (maybe an AWS implementation) and the payment services. Also, the freshness of data relies on updates from the restaurant user. The restaurant user may leave stale data in the DB. That could cause an order by a user that cannot be fulfilled in the requisite time. The application may not function if connection to the DB, or the server, are not operational.
    b. **Architecture B:** A hybrid peer-to-peer architecture requires both the user and restaurant to be able to access a server in order to post and get connection availability. The user and restaurant client apps must then maintain a connection for communicating order information. Errors on either end of the peer-to-peer connection could leave the ordering system in an undefined state. This would be a serious reliability concern.
2. **Efficiency**
    a. **Architecture A:** This architecture requires a user request to the client, that goes to the server, pulls information from the DB, if that info is stale, a request is pushed to the restaurant user, and then the info is sent back to the user. Since, it is a real time system, it will require data refresh on a regular basis. That could be done on as a direct request by the user device. This architecture is more scalable, allowing multiple users to view the current restaurant data in a location without needing a connection to the restaurant user. There are ways to make small use cases more efficient, but large cases will be more efficient on this architecture.
    b. **Architecture B:** The peer-to-peer architecture would have very poor efficiency, primarily limited by the restaurant client’s ability to maintain many active connections and order states with current user clients. This would limit the architecture’s scalability as increasing concurrent users would quickly impact system performance.

3. **Integrity**
   a. **Architecture A:** There is a lot of data being sent to multiple users. At any point in the data flow it could be intercepted or corrupted. The system requiring the most protection is being outsourced via an apple/amazon pay api. The other systems are merely pieces of data in the database. The front end requires authentication and the network communication can use 256 SSL if necessary, 128 SSL may be enough security for the exchange of food orders and menu items.
   b. **Architecture B:** The system would not be able to ensure the data security or integrity of data passed between the peer clients. As pointed out in the reliability discussion, the system could also conceivably be put into a bad state if there is an error in data transmission or a lost connection.
4. **Usability**
   a. **Architecture A:** The biggest drawback to usability is the need for a network connection and the need for data updates by the third party users. Without a connection the system will not work. Given that the system is supposed to be as close to real time as possible, having a network connection is going to be a requirement for all architectures. The third party issue can be solved with a filter on the search results, ie not providing stale data. Overall this architecture allows for ease of use and balances additional needs.
   b. **Architecture B:** Usability in the peer-to-peer architecture would potentially be more difficult than an alternative architecture. The peer-to-peer architecture puts the responsibility of connection integrity and security on the client side applications. This could be addressed with client side code, but the number of recoverable failure modes would likely increase the user complexity on both ends greatly.
5. **Maintainability**
   a. **Architecture A:** This architecture allows for easy maintenance. As opposed to pushing a downloaded update out to the users this system allows for backend changes and then a single change from one process to a different process. This allows for changes to be made incrementally on each part of the system as time passes. It also allows for roll back to a stable release if the current release

causes issues. This system is more modular. Disparate teams could work on discrete tasks without fear of killing the whole system

b. **Architecture B:** Maintainability will be more complex for a peer-to-peer architecture as it will require simultaneous deployment of changes to the two peer clients, as opposed to making mutually compatible changes to a restful API. This also means that there is increased difficulty establishing compatibility between two clients using different versions of the application.

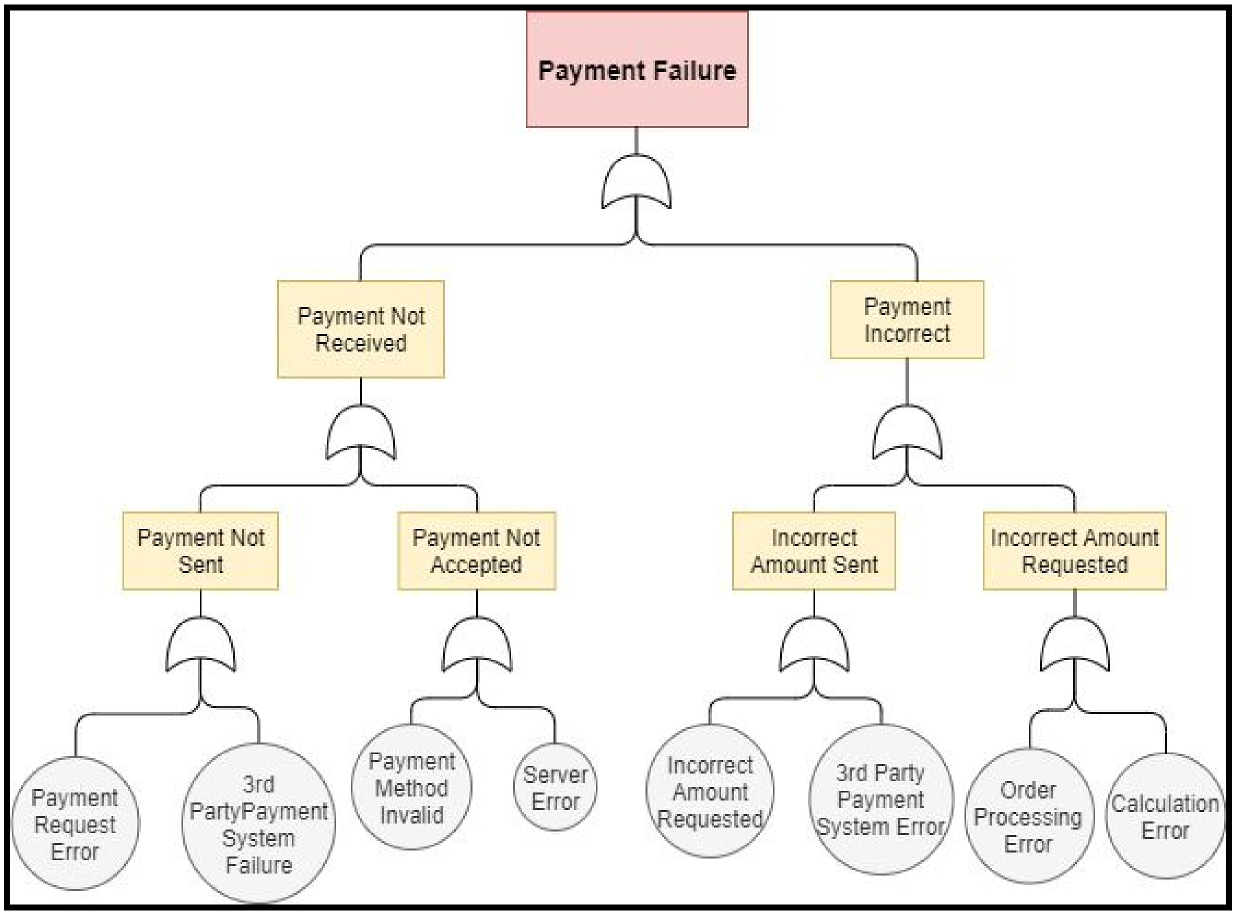
6. Flexibility

a. **Architecture A:** Breaking the system into multiple pieces allows changes and growth to the system without a complete redeployment. The client can be developed in a multitude of languages because the server just supplied data via an api. The DB is handled with MySQL calls. Meaning the server could be switched from node to python by maintaining the front end api and the backend severe calls. The whole look of the front end could be redone without touching the server or the DB. This architecture allows for maximum flexibility.

b. **Architecture B:** The peer-to-peer architecture has a particular type of flexibility related to its relative freedom from reliance on a server. The only thing that the peer clients require from the server is a reliable listing of available peers. This means that the system is flexible in terms of being free from most server processing load demands, but this demand is transferred to the restaurant side client and each client’s host hardware processing capacity.

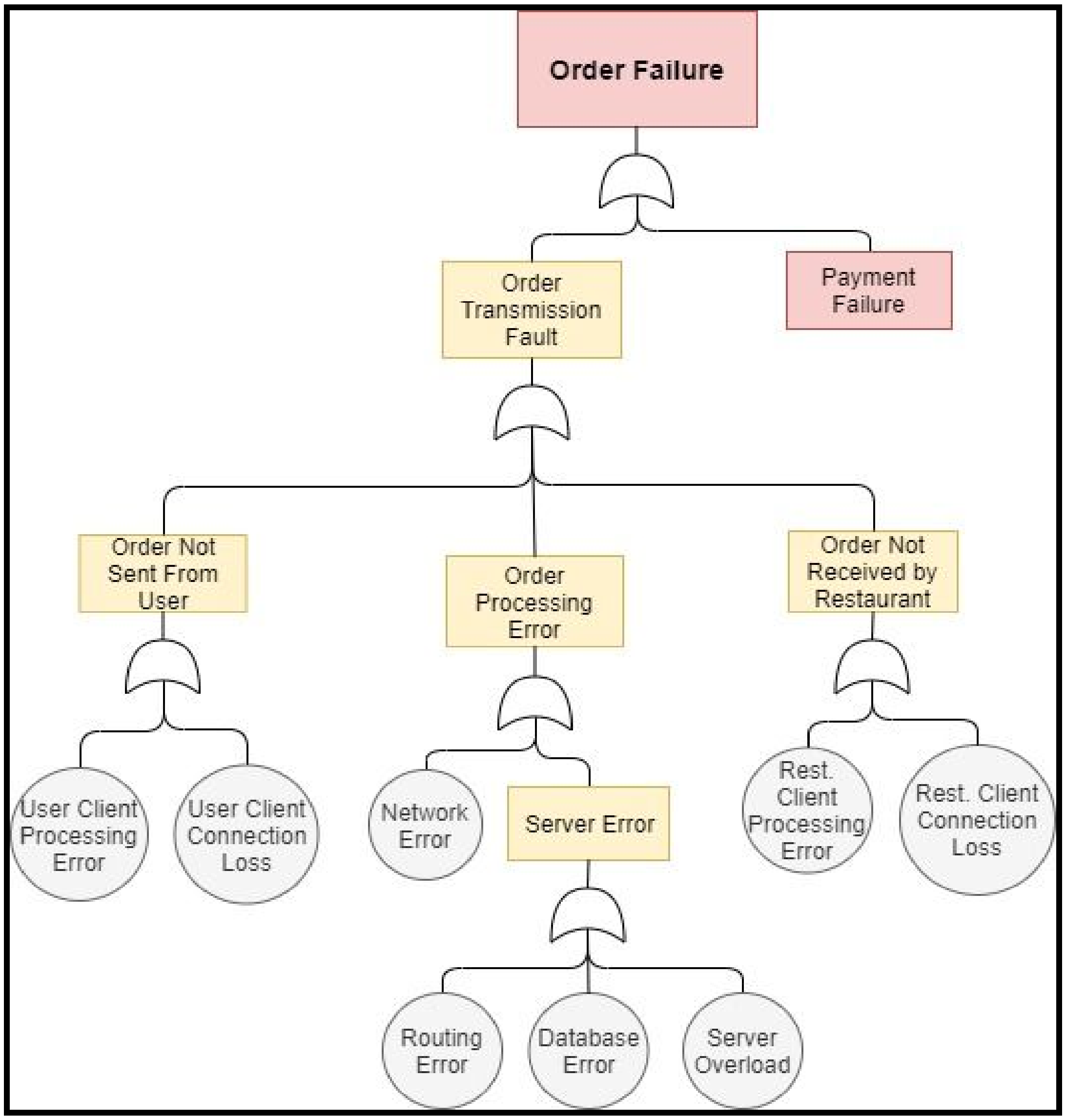
Failure Modes Analysis:

❖Failure Mode 1: Payment Failure



➔ Analysis of the failure mode shows that a peer-to-peer architecture would likely be more vulnerable to payment failure. While the peer-to-peer and client-server architectures share great similarity due to the use of third-party payment systems, the failure mode factors for the peer-to-peer architecture are essentially twofold that of the client-server architecture as responsibility for the payment process is distributed to the user clients instead of being centralized in the server-side application. Both architectures share a similar vulnerability to failure of the third-party payment system, but a server-focused architecture will likely be more recoverable from such faults than the distributed client system, and thus the superior choice for payment failure mode attenuation and avoidance.

❖Failure Mode 2: Order Completion Failure

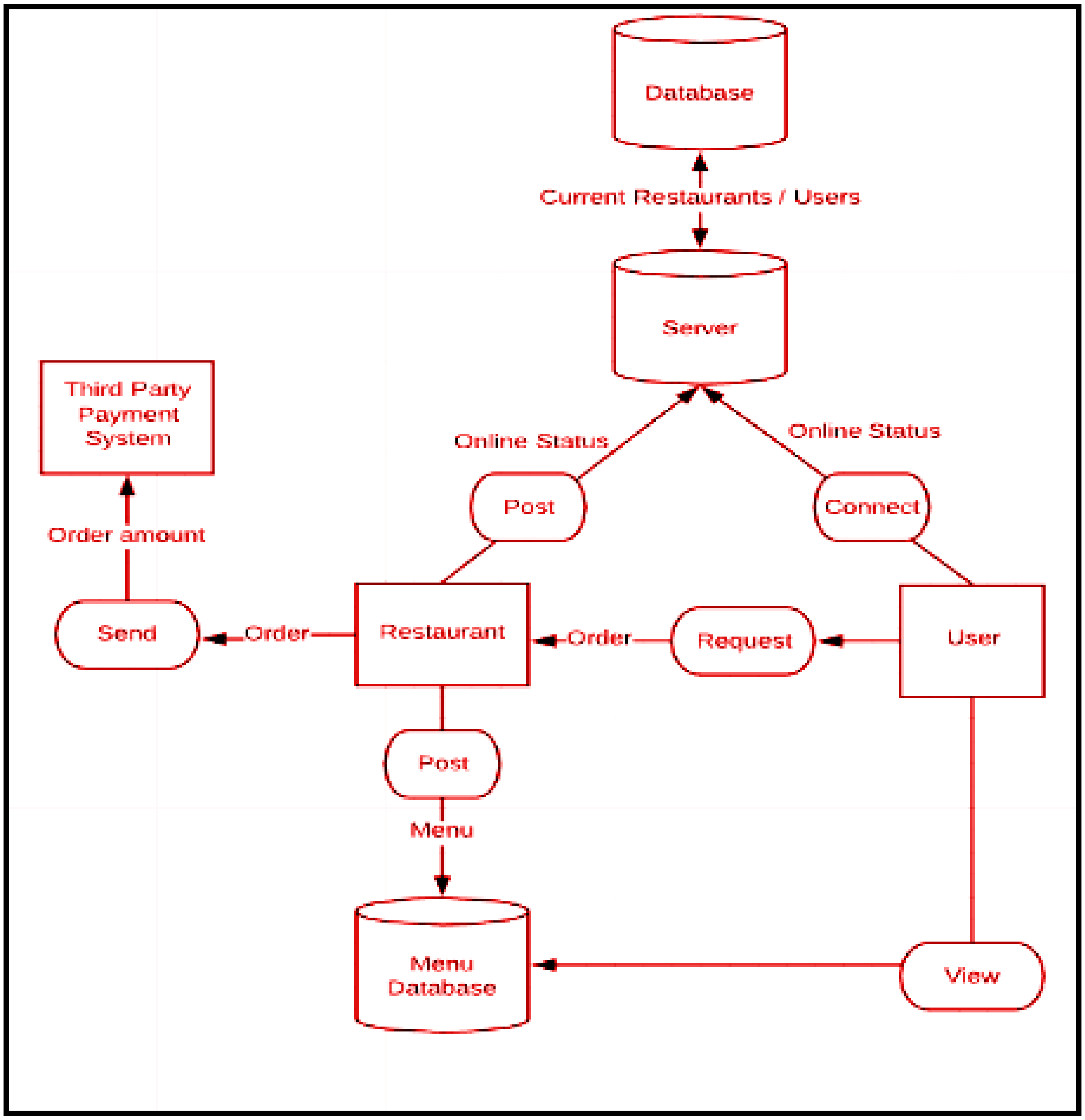


➔ Similarly to the previous failure mode--and in fact including the previous failure mode as a factor--the order completion failure mode analysis shows the client-server architecture to be superior to the peer-to-peer architecture. This is due in part to the server's ability to maintain, secure, and backup a shared state between the user and restaurant apps. In the peer-to-peer architecture, the order state must be maintained between the two client application and any data fault or connection loss will likely result in an unrecoverable state. While server error is a potential factor leading to order completion failure. A robust server system will be resistant to errors via error handling and resilient to server overload via scalable server load distribution.

Architecture A Client-Server Decomposition

❖Key element 1

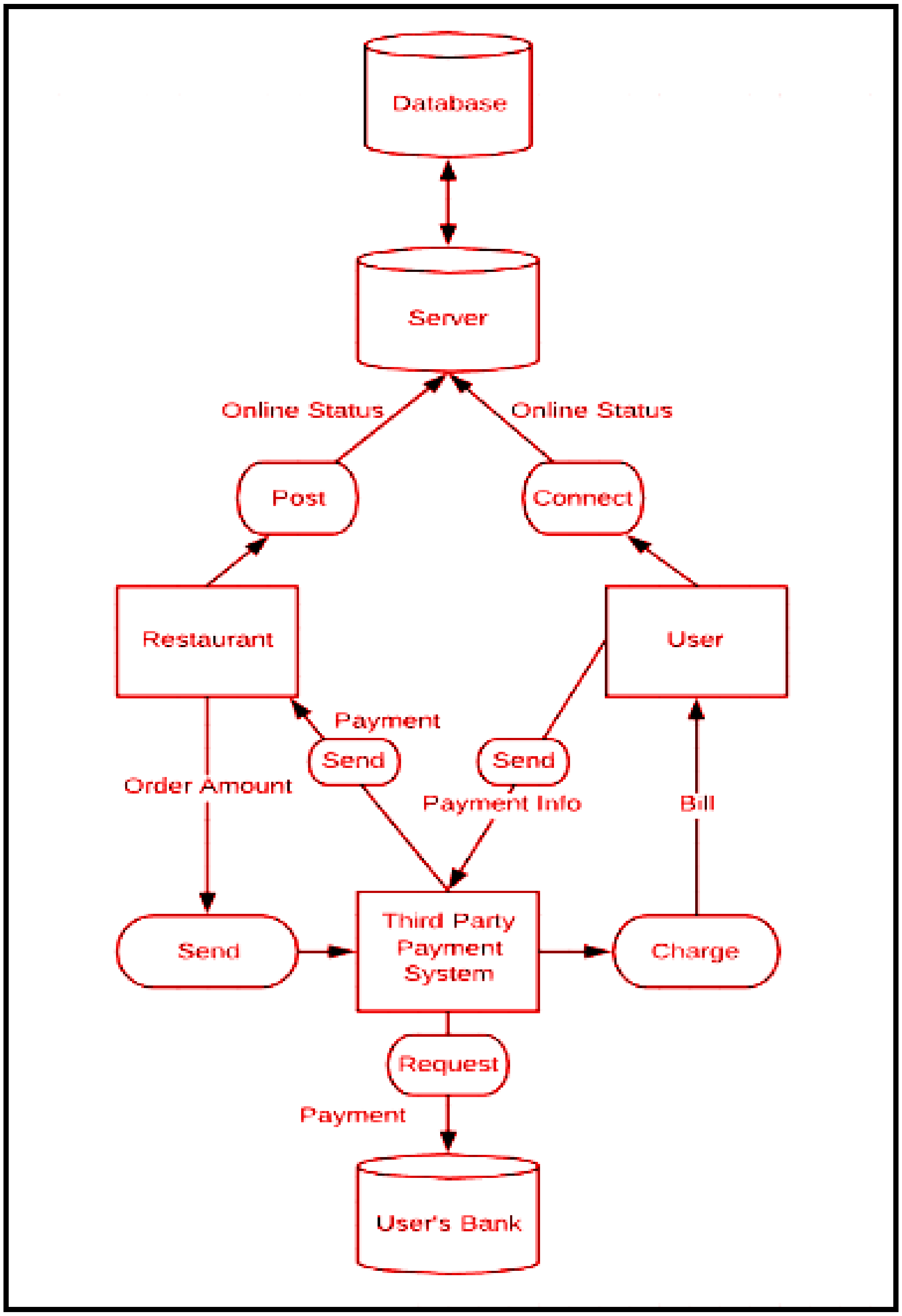
➢Order Processing



➢

❖Key Element 2

➢ **Payment Processing:**



Architecture Validation:

❖Use Case 1 - Customer places an order and pays:

1. The user will access the server through the user side portal. The user will first authenticate their account to gain access to their user information through the portal.
2. The client portal will make a call to the server API to get the required information to populate the client portal.
3. The server will make the SQL calls to the server to get the user data to send back to the client portal.
4. The server will make a request for current wait time from the restaurant, if non responsive the server will provide current DB data to client portal, if less than an application defined time period.
5. The client side portal renders the JSON data in either a Swift based iOS application or JavaScript web app.
6. The client then interacts with the client side portal to select a restaurant
7. The same process as above occurs with calls to the server API by the client side portal and the JSON rendered by the portal.
8. The client selects items for the order on the portal, Those items are associated with the client in the DB by a call to the server. That association is sent to the restaurant portal after initial payment confirmation.
9. The user is prompted by the client side portal to pay. Payment is handled by the user selected third party application, ie ApplePay, PayPal, etc
10. Payment status data is sent to the server, stored in the DB and pushed to the restaurant.
11. Upon delivery of the food, restaurant user updates delivered status on the restaurant portal, that makes an API call to the server, updates the DB and pushes a request to the user side client portal for final confirmation and tip.

❖Use Case 2 - Customer creates an account:

1. The user will access the create new user page through the user side portal. The user will enter their first name, last name, email, phone number, and password.
2. The user application will test if the user's input matches the requirements for email, and passwords. The user application will indicate to the user if the input is valid or no, based on preset criteria through the user side portal.
3. The user will select their preferred third party payment option.
4. The user side portal will send the information to the server and the server will query the DB to see if that email is already registered. The server will return the result to the user side portal.
5. The user is prompted to read and then either accept or decline UEat's User Agreement and Privacy Policy, but the application will not proceed unless the box has been checked.
6. The user side portal will send the confirmation to the server. The server will then pass the user's information to the DB to have a new entry created in the DB for this new user.
7. The server will send a confirmation email to the user's email and a text message to the user to confirm both methods of contact.

❖Use Case 3 - Restaurant adds items to menu:

1. The restaurant user will input login info into the restaurant application.
2. The restaurant application will send that information to the server. The server will query the information from the DB and confirm. The server after confirmation will request the restaurant page information from the database in preparation for display.
3. The server will send the confirmation to the restaurant application. The server will send the page information to the restaurant application to be displayed. The restaurant application will display the connected restaurant page.
4. The restaurant user will view the displayed information on the restaurant application. The restaurant user will input and or upload the desired menu changes and submit them.

5. The restaurant application will validate the input. The restaurant application will send the updated menu information to the server.
6. The server will send the changes to the menu to the DB, and wait for confirmation from DB. The server will relay the confirmation to the restaurant application that the changes were successful.

Architectural Implications:

Based on the results of our validation and verification activities, we can deduce several appropriate revisions of our selected architecture. The most important revisions of the client-server architecture to consider are those that reinforce the statefulness of the server in relation to the clients and those that increase the resilience of the server and database. Revisions to the architecture that will reinforce the statefulness of the server include implementing a REST interface with the client applications and utilizing an object-relational model for the interface-application-database architecture decomposition. Additionally, revisions to the architecture that will improve the resilience of the server and database include implementing strong error handling systems and server/database distribution across a cloud platform.

Reinforcing the statefulness of the server by implementing a REST interface is a well-established practice for creating web-based application reliability. An object-relational model will allow changes in the application states to be immediately reflected not just in the server state, but also in the recoverable database state. Improving the resilience of the server and database by error handling can be accomplished with good server side code handling and use of an ACID compliant database. Improving server and database resilience can also be achieved by deployment across multiple servers. For an application intended to serve clients around the world, this is best achieved by deploying cloud-based servers and databases. A cloud-based architecture is more easily achievable than ever before through the various web-hosted platform services, which often have the benefit of integrated scaling, distributed server caching, and automated server management.

These revisions would have a significant impact on both the quality attribute evaluation and failure modes analysis results. By centralizing communication with external clients and systems into the server, the quality attributes and failure modes become centered around the quality of the server model--hence the server-heavy descriptor for our chosen architecture. Making the server stateful and resilient to error in turn makes the clients highly recoverable where the competing architecture would fail.

❖ Team Contributions:

- ➢ All: Communication via private Slack channel and collaborative Google Doc.
- ➢ Timothy Tseng - Failure Mode 1 & 2 Diagrams and Key Element 1 & 2 Decomposition Diagrams.
- ➢ Miles McCoy - Architecture A Dataflow Diagram, Architecture A Quality Attribute Comparison, Use Case 1 Validation.
- ➢ Ibrahim Mahmoud - Architecture B Diagram.
- ➢ Luke Carter - Use Case 2 Validation, Use Case 3 Validation.
- ➢ Mathew McDade - Document Preparation and Submission, Architecture A & B Dataflow Diagram, Architecture B Quality Attribute Comparison, Failure Mode 1 & 2 Diagrams and Analyses, and Architectural Implications.