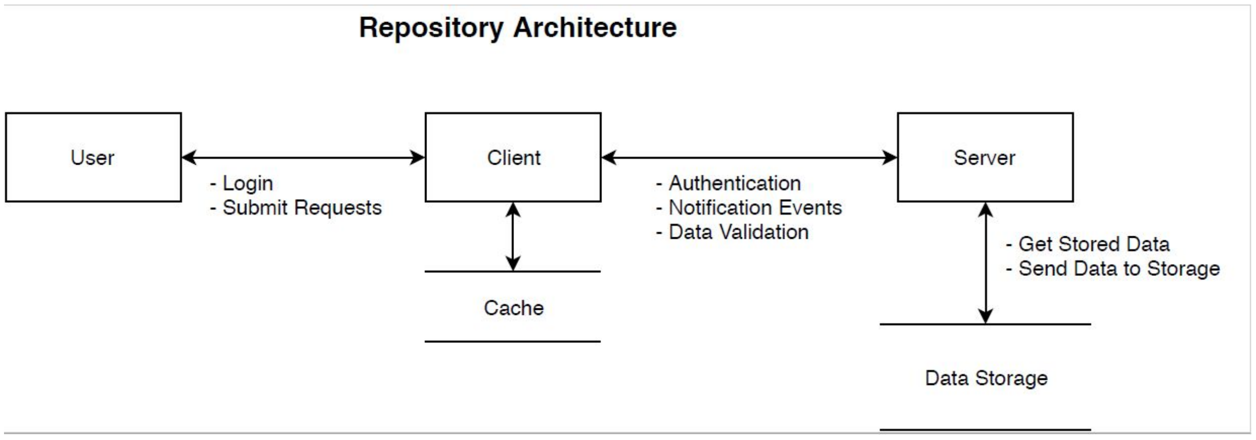
Scheduling and Time Management Application - Requirements

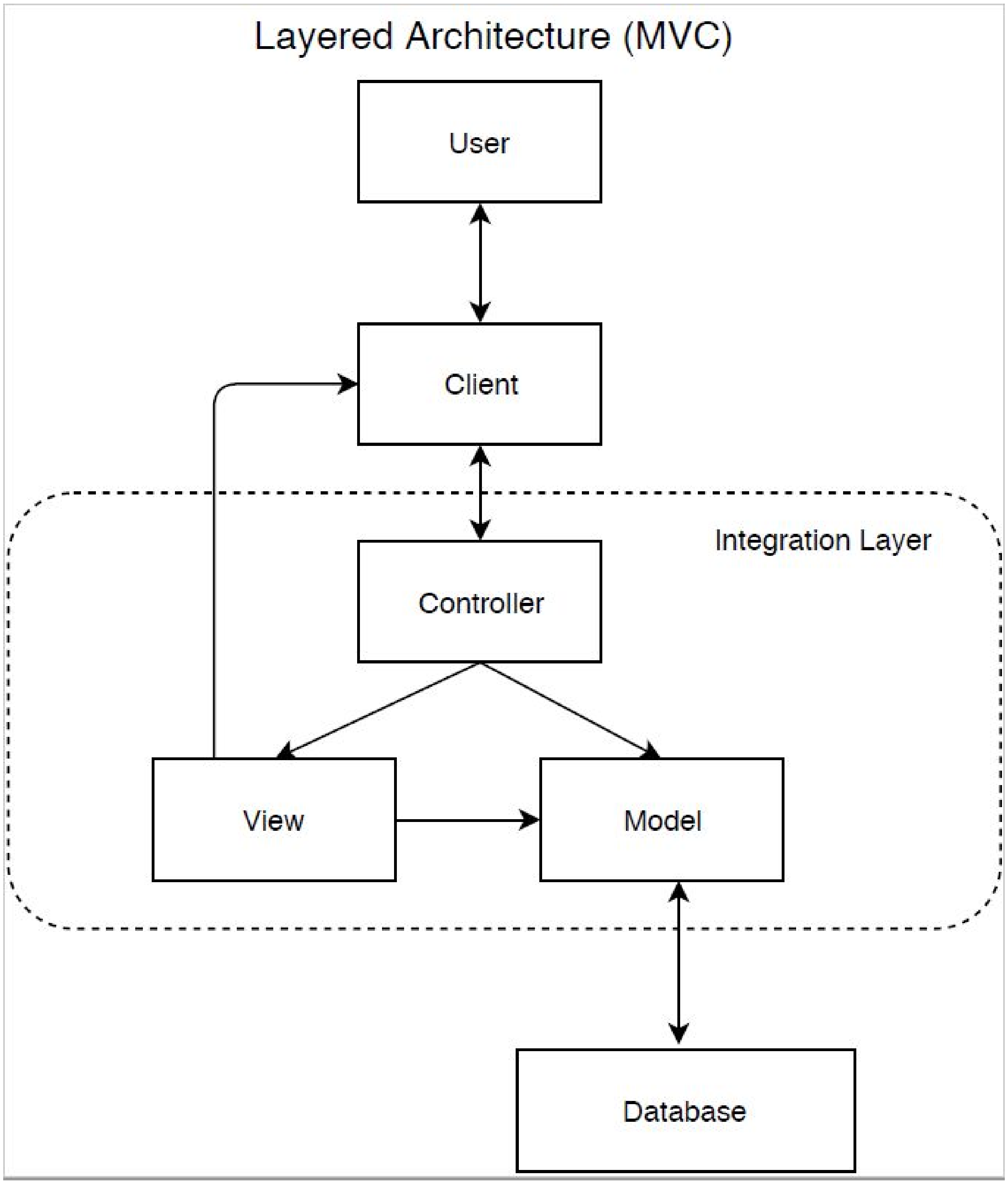
2019/11/3

Group 20:
Sunghoon Cho
Michael Czapary
Christopher Elliott
Alex Kolstad
Zachary Wetekamm

Repository Architecture



Layered Architecture:



Key Quality Attributes:

Reliability

Architecture 1: The repository architecture would rely on the server for transmitting most information between the client and storage. In the event of a server failure, the system will need to use a secondary server as a backup—which will run on the current release of the software—to ensure reliability. Some basic functionality will be available if the server is offline, but data may not be the most recent; for example, a user may record timestamps locally, but the information will not be sent until it is back to a working state.

Architecture 2: The layered architecture is also reliant on the server for transmitting most information between the client and storage. Layers communicate between one another, and the server must be active to send correct data to different layers. A backup storage is necessary for the application to be reliable. Some basic functionality will be available if server is offline, but data may not be the most recent; for example, a user may record timestamps locally, but the information will not be sent until it is back to a working state.

Efficiency

Architecture 1: Since the application features time management, it is largely dependent on efficiency and accuracy. The repository architecture will allow users to store and access data from and immediately affect how others view that data. However, since this architecture is dependent upon server status, it will need to be reliable to be efficient. Client and server will have external dependencies—such as user internet speed.

Architecture 2: A layered architecture may be susceptible to inefficiency as more layers are added to the application. In terms of speed, this could potentially lead to a redesign depending on the scale and longevity of the application. Layers will communicate between client and server and have external dependencies—such as user internet speed.

Integrity

Architecture 1: The repository architecture will use the central server to authenticate users. However, a user will need to keep their own computer secure, such as with antivirus software and physically secure login credentials or other sensitive information. The client side would more likely be exposed to security compromises, but the server will be less exposed with validation between client and server. The application will require users to periodically update password credentials.

Architecture 2: Layered architecture is excellent because of separation of concerns between layers; components will interact but will not necessarily know specifics about that neighboring component. Authentication will occur on the server. This application would benefit from strong security, such as an MVC design, where data is protected by classes. The application will require users to periodically update password credentials.

Usability

Architecture 1: The repository architecture will reflect the most recent data for users since all components interact with the same database. A responsive web GUI used with the client will give the user an eased experience. However, the architecture would struggle in the event of a server failure. A running backup of the most recent iteration of data will be necessary.

Architecture 2: The layered architecture would likely follow an MVC design and render views for the user. The architecture is dependent on server status and may not be the most recent if a server failure occurs. Users may access a limited functionality, such as recording timestamps locally, but will not be able to submit that data until the server resumes working status.

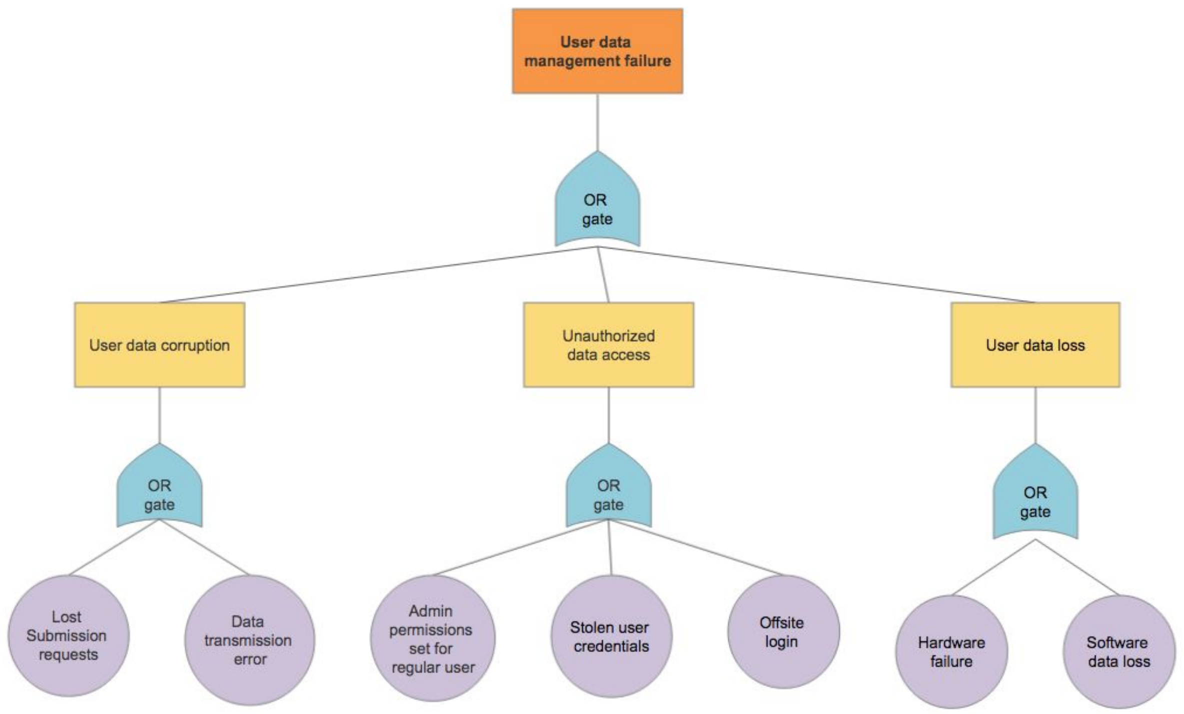
Maintainability

Architecture 1: The user would not need to update the application, as the server will handle updating data on the server side. However, if some data were stored or updated incorrectly, this would affect the experience for every user because all users access the same data.

Architecture 2: Layered architecture can prove costly depending on scale. Scaling upwards would require more layers to the design. The user would not need to update the application, as the server will handle updating data on the server side. Changes to an individual layer will not affect a neighboring layer, which is a huge benefit.

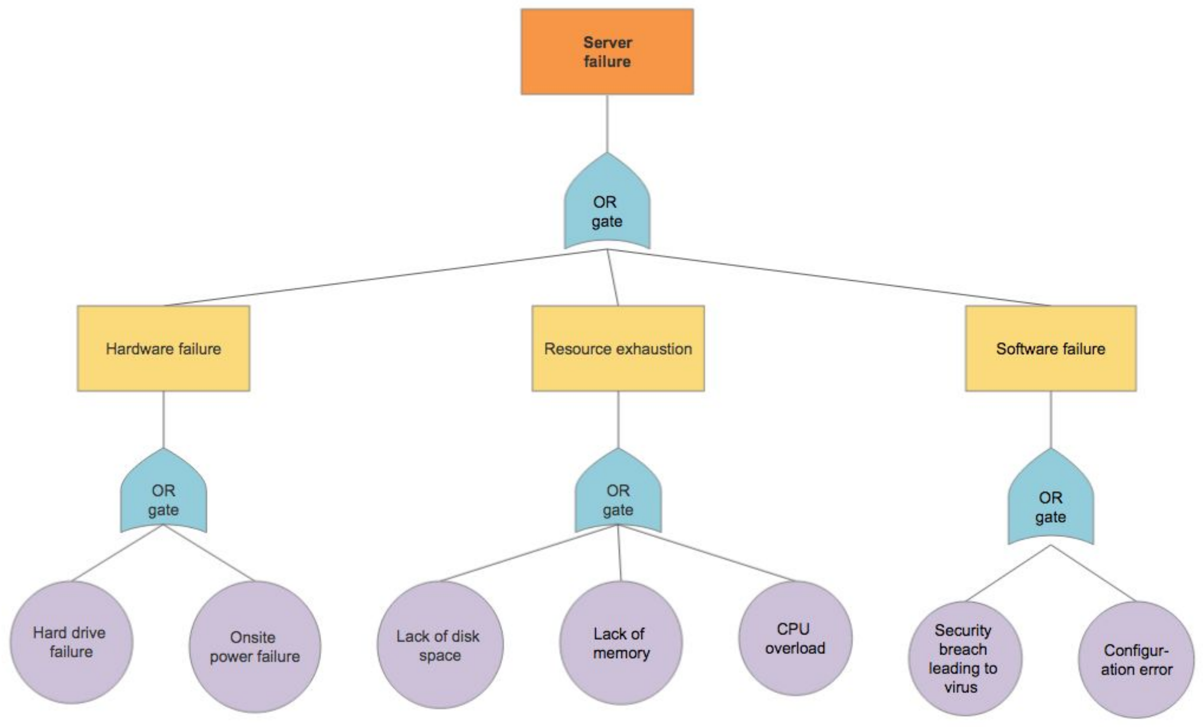
Failure Modes:

User data management failure (fault tree)



The architecture more prone to failure for the above failure mode is the repository architecture. Because of its central data store where there is no middle layer, the reliance on the server makes it more prone to failure. For the three cases leading to user data management failure, all of them occur during the read and write process with the client during usage. Authentication also occurs on the server and any unauthorized access will lead to this failure mode and periodic updates of credentials will be required to prevent security compromises. A layered architecture may separate each concern and deal with it slightly better at each layer (as the functions may be hosted on several machines or clusters) than the repository architecture, although not vastly because it is also server dependent.

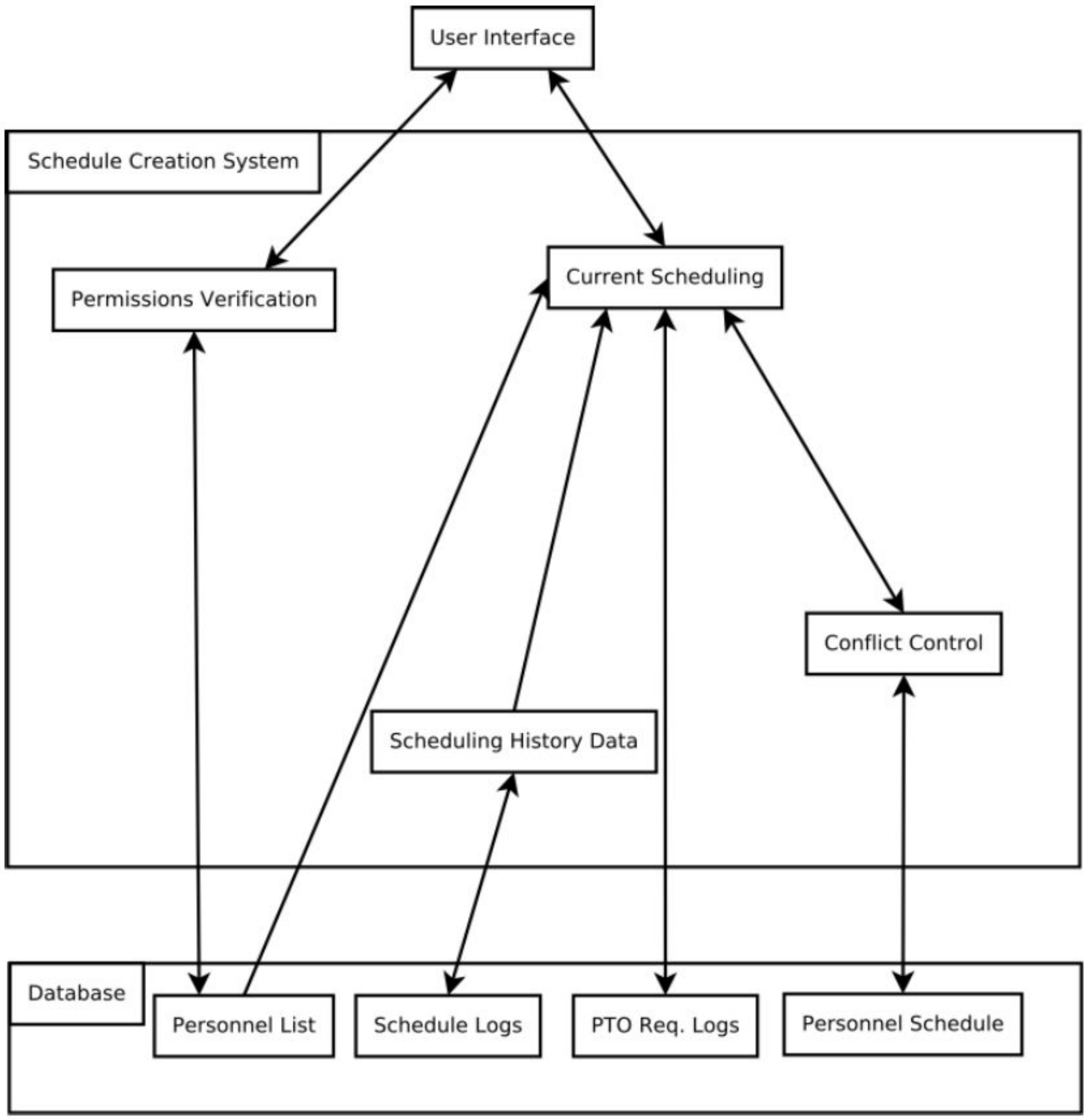
Server failure (fault tree)



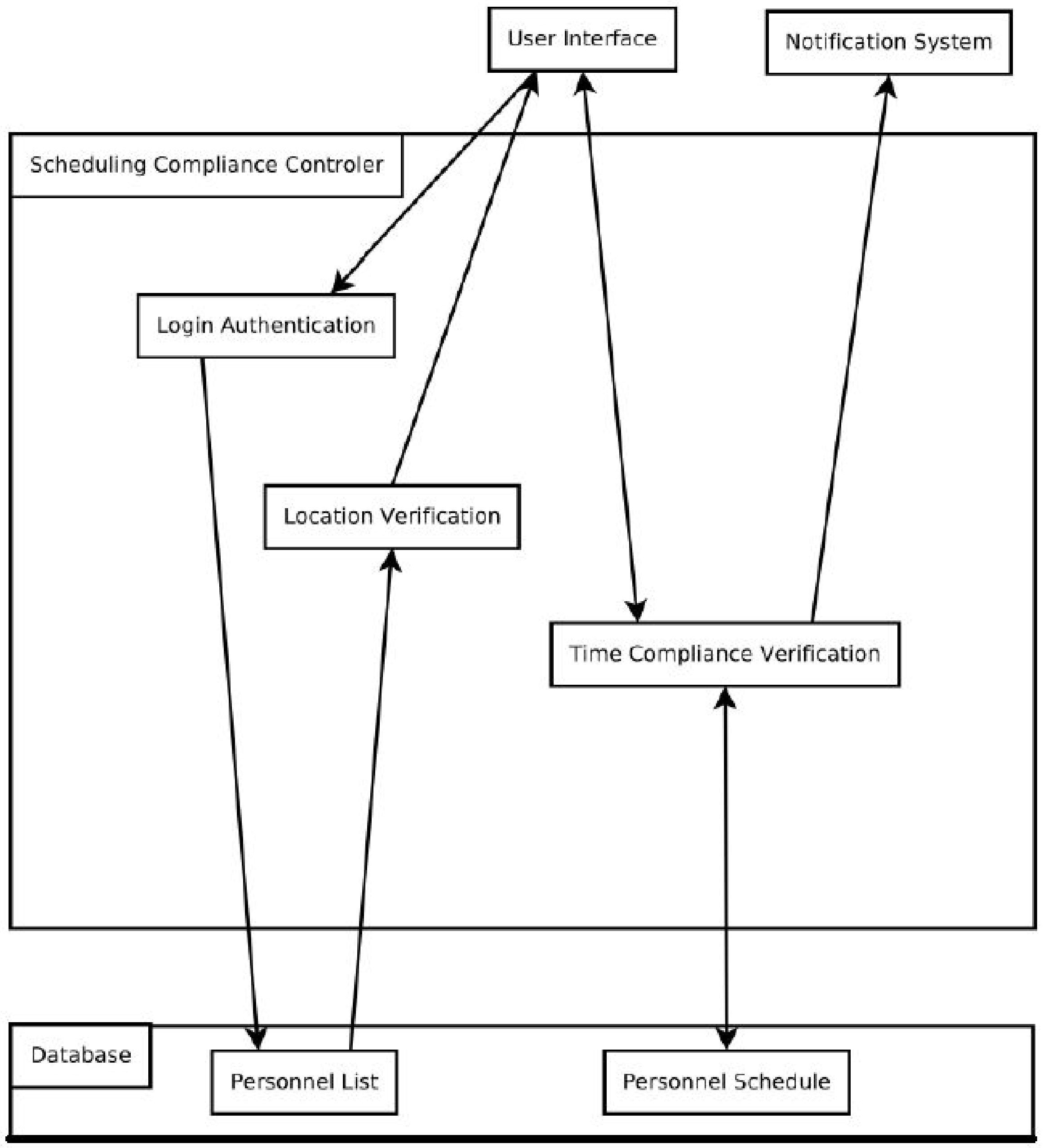
The architecture more prone to failure for the above failure mode is the repository architecture. The data centered style which relies on a server makes server failure its most significant failure mode. All aspects of the service would cease to function. The layered architecture, while also server dependent, would likely offer a slight advantage as the buffering and data marshaling at each layer may slow down the causes for server failure. However, it still is not a better choice because any faults in one layer will spread upwards to calling layers and not handle errors and exceptions very well.

Lower level Data Flow Diagrams:

Schedule Maker:



Clock In Controller:



Use Case Walkthroughs:

USE CASE #1: Employee Timestamp

- The user will be able to login to the system by connecting to the client. The client will verify the user's credentials by comparing them to what is stored in the database via the server. The client will also verify that the user is connecting from the worksite.
- The client's user interface will allow the user to complete a number of timestamp functions including clocking in, clocking out, taking a break, or going to lunch.
- Whether the client is clocking in, clocking out, taking a break, or going to lunch the system will send this information through the client to the server and database where it will be stored.
- The server will return a success or failure message based on whether the action was successful or not to the client. The server will generate this message based on a response from the database.
- For clocking in and out the server will compare the current time to the scheduled work time. If the user is late then the server will send a message to the appropriate manager through the client. (More on this in use case #2)
- Managers will be able to edit timestamp functions through the client's user interface. All changes will be sent to the server and stored in the database along with the original timestamp action.

USE CASE #2: Notification of Tardy or Absent Employees

- As defined in use case #1 a user will be able to log in to the system through the user interface.
- The database will store a record of all employee schedules. When a user successfully clocks in for, the work client will send the timestamp of that action to the server. The server will then compare the current clock in timestamp with the scheduled clock in timestamp in the database.
- If the current clock in timestamp is later than the scheduled clock in time the server will generate a message and send it to the client. The client will push this notification to the appropriate member of management.
- After receiving this notification, the manager will be able to attach a message to it through the user interface. This message will then be sent back to the server and stored in the database with the timestamp action.
- The server will also inform the employee of their tardiness when sending a successful clock in confirmation.

- Periodically throughout the day the client will send a message to the server to check if all currently scheduled employees have clocked in. The server will compare the scheduled timestamps with the received clock in timestamps to find any missing employees.
- If an employee that was scheduled to work has not clocked in at all then the server will generate a message and send it to the appropriate member of management via the client.
- Periodically (based on user settings) the server will compile data on frequency of employee tardiness by checking the database. It will then send these reports to the appropriate manager though the client.

USE CASE#3: Request Paid Time Off

- As defined in use case #1 an employee will be able to log into the system.
- The user interface will present the option for the user to request paid time off.
- The server will parse the information in the request and check with the database to see if the employee as sufficient paid time off remaining to cover the requested time.
- If the employee does not have enough paid time off remaining the server will generate a message stating so and send it to the employee through the client.
- The employee will then be able to make a special request, if this happens then the client will send this message to the server which will relay the message back to the appropriate member of management through the client.
- If the server confirms that the employee has sufficient time off remaining then it will send a message to a member of management through the client.
- If the manager denies the request then a message will be sent back to the employee stating the request was denied and a reason given by the manager.
- If the request is granted then the client will send a message to the server to reduce the remaining paid time off stored in the database. The server will then send a message to the scheduler to notify them of the time the employee will be unavailable. Finally, server will send a message to the employee notifying them that their request has been granted.

Implications:

- The biggest issue of having all of the services available through a web portal or app is what happens when the users are unable to connect to the server for whatever reason. A possible solution for clients that use the app is to cache information locally on the client's device in case the user could not connect to the server, as well as to fall back to other services that do not use the server, like email for messaging. The information the client could cache locally could be:
    - The user's schedule of shifts to work
    - The user's prior clock-in and clock-out times
    - The user's currently scheduled Vacation days and PTO days
    - The contact information of their administrator, including the administrator's phone number and email in case they cannot connect to the server

  Whenever the client connects to the server, the client could fetch the updated information from the server to ensure the cached information is current. This way, the user could still see the times of their shifts, and if they need to contact their boss or administrator, they have their boss or administrator's contact information to get a hold of them, even without the server's built in messaging tools. The client could even handle the emailing of administrators internally, without having to use a separate email client, though the user would have to use their phone's texting client if they wanted to text their boss or administrator.

- The client that the employees use in the workplace would run on the same machine as the server itself, so when it is offline, it loses the ability to notify administration when an employee clocks in or out, or if an employee is late or absent that day. A potential solution to this could be to use a wireless protocol other than WiFi to notify administrators. Bluetooth could be a viable option, so long as the administrators remain within the range of Bluetooth to the server.

- Because the client and server talk to each other, another potential option is to have the client ping the server at a defined interval, and if there is no ping sent for a certain amount of time, then the administrator or server could reach out with any new information. This information could be sent using means other than the messaging built into the server, like email or SMS, as even if the user did not have internet access, if they had the bare minimum of cellular reception, they could still get in contact with the boss or administrator and continue receiving up to date information. A downside of this implementation would mean the app running in the background to be able to ping the server when the app is not open, but depending on the ping interval, the loss of performance and battery life to the user could be very negligible. The app could also make it clear to the user that it is in “Offline Mode” and is only using the cached data on the device, so the user knows that information may have changed since the app last cached data.

Team Member Contributions:

Zachary Wetekamm: Made both high level data flow architecture diagrams and wrote key quality attributes.

Sunghoon Cho: Made both failure mode diagrams.

Christopher Elliott: Made both lower level data flow diagrams.

Mike Czapary: Wrote all three use case walkthroughs.

Alex Kolstad: Wrote the implications