CAS735 - Final Report

ACEM PARKING Service

Team T1:
Jingwen (Steven) Shi
Yaqi (Regana) Zhang
Ziyang (Ryan) Fang

Dec 05, 2024

1. System Architecture Diagram

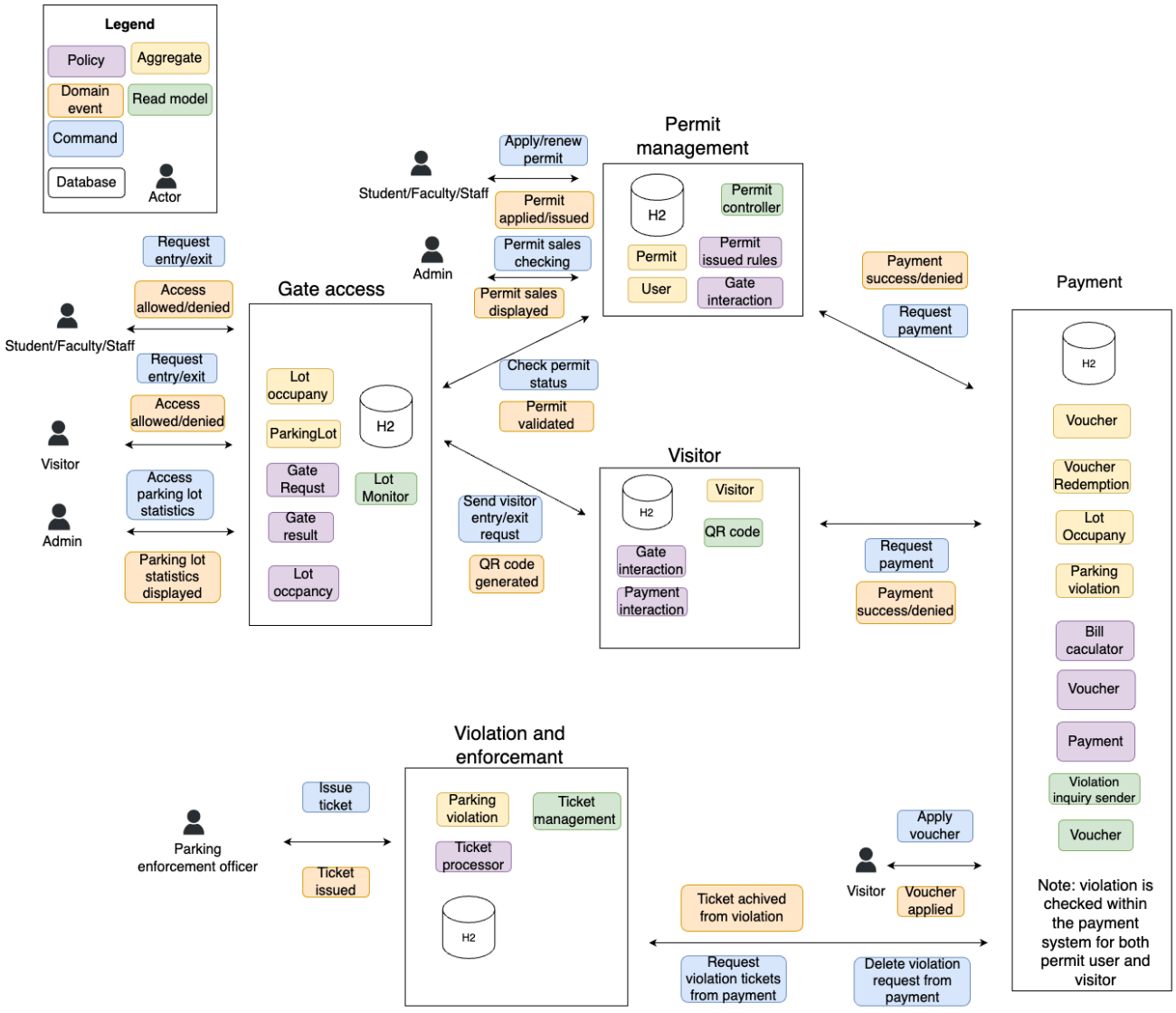

Figure 1. System Architecture diagram

1. System Workflow

- Permit:
    - **Apply Permit**: The user submits a permit application, and chosen payment method (credit card or payroll). The **Permit Service** processes the application and calls the **Payment Service** to handle the payment. Upon successful payment, the **Payment Service** notifies the Permit Service, which then stores the data and sends a confirmation response to the user.
    - **Renew Permit**: The user submits a permit renewal with, and chosen payment method (credit card or payroll). The **Permit Service** processes the application and calls the **Payment Service** to handle the payment. Upon successful payment, the **Payment Service** notifies the **Permit Service**, which then stores the data and sends a confirmation response to the user.
    - **Permit validation**: When the **Permit Service** receives an entry request from the **Gate Service**, it checks the transponder number associated with the permit and sends a validation result back to the **Gate Service** if the permit is valid.
    - **Check permit sales**: Administrators can verify valid permits through the **Permit Service**.
- Gate:
    - **Entry**: When a car enters the gate, the **Gate Service** first checks for a transponder. If a transponder is detected, the service sends the information for permit validation. If no transponder is found, the Gate Service forwards the request to the **Visitor Service** to generate a QR code and awaits the result before granting entry.
    - **Exit:** When a car exits the gate, the **Gate Service** checks for a transponder. If a transponder is detected, the car is permitted to leave. If no transponder is found, the **Gate Service** requests the **Visitor Service** to process the payment and awaits the result before granting exit. On exit, the **Monitor** will record the current parking lot occupancy.
    - **Parking lot monitor**: Monitor parking lot occupancy in real-time, including occupancy rates and peak usage times. Administrators can access detailed statistics and reports as needed.
- Visitor:
    - **Generate QR code**: When a visitor enters, the **Visitor Service** receives a request from the **Gate Service**, stores the visitor's data, generates a QR code, and sends it back to the **Gate Service** for entry.
    - **Process payment**: When the **Visitor Service** receives an exit request, it forwards the request to the **Payment Service** to process the parking fee. Once the payment is successful, the **Visitor Service** sends the result back to the **Gate Service**.
- Violation:
    - **Issue ticket**: The **Parking Enforcement Officer** can issue a ticket through the **Violation Service**, which then stores the issued ticket.
    - Retrieve tickets: When the **Violation Service** receives a request from the **Payment Service**, it returns a list of all existing tickets associated with the vehicle by verifying the license plate. The client also can use this protocol to retrieve their violations.
    - **Delete ticket**: When a ticket deletion request is received from the **Payment Service** after all fines are paid, the **Violation Service** deletes the ticket from the database.

- Payment:
    - **Process permit and visitor payment requests**: The **Payment Service** processes payment requests from both the **Permit Service** and the **Visitor Service**. For permit payments, it handles different payment methods, such as payroll deduction or credit card, based on the user's selected option. For visitor payments, the **Payment Service** calculates the appropriate price and processes the payment accordingly. Additionally, the **Payment Service** checks for any violation fees associated with both permit and visitor payments. For visitors, the service also verifies if a voucher exists and applies it if found.
        - Request violation ticket: The **Payment Service** will request the **Violation Service** to check if a violation ticket exists for the vehicle by verifying the license plate. If a violation is found, the **Payment Service** will add the violation fee to the total payment. For visitors, the violation fee will be added to the visitor fee, while for permits, it will be added to the permit fee before processing the payment.
        - Voucher: The user will apply for a voucher associated with their license plate. The **Payment Service** will store the voucher, and when a visitor payment is requested, it will verify if a voucher exists. If a voucher is found, the **Payment Service** will apply it and calculate the visitor's payment accordingly.

2. System Advantages:

- Hexagonal Architecture
    - The system is designed to follow hexagonal architecture with ports and adapters design patterns. All services have input and output ports as interfaces that isolate the interaction between services and external infrastructures.
    - This allows more flexibility and scalability in code by replacing adapters without changing the main logic of the business module.
- Load Balancing
    - RabbitMQ and Docker built-in load balancer ensures the traffic will be automatically and dynamically redirected to other running containers and services to minimize downtime.
    - Nginx listens on port 80 for all incoming HTTP traffic and proxies the requests to the corresponding backend service port based on the URL path. This extra abstraction layer allows the client to only use the default HTTP port for communication and does not need to change the client side's URL if the backend service's port has changed.
        - For example, gate service runs on port 8081, but the user can directly send a request to port 80 (i.e. http://localhost/gate/monitor/lookup?lotId=1) instead of port 8081 (i.e. http://localhost:8081/gate/monitor/lookup?lotId=1)
- Auto Restart on Failure
    - By using the restart policy with docker, any crash due to application errors will automatically restart containers.

3. System Limitations:

- Horizontal Scalability
    - The system is not horizontally scalable as the H2 database is bound within each service. In other words, the database is not an independent service that all other services can directly communicate with.
    - If we scale up the number of replications of each service with docker-compose, multiple replications can read from the same exchange at the same time while each replication has its database, which will lead to data inconsistency.
- Single Point of Failure
    - Since each service has only one replication running at the same time, if a node or process fails, the entire system will break as no other replications can handle the user's request.
- Availability
    - Due to a single point of failure, the system cannot continue to operate and respond to user requests despite failures of processes or nodes, which means low availability.
- Partition Tolerance (Limited)
    - The system is split into independent services that can continue functioning even if other services are unreachable (e.g. Apply/renew for a new permit).
    - However, due to the issue mentioned above, once a service fails in the critical path (e.g. gate service, visitor service), it will break the entire flow. Therefore, there are no mechanisms that exist to handle partial system failures gracefully unless the scalable issue has been solved.

4. Future Improvements:

- Database Decoupling
    - Extract database as an independent service that all other services can communicate to. Then, all services will be able to scale horizontally and solve the limitations mentioned above. Note that, there still might have a single point of failure if the database itself is not a distributed database.
- Consistent Hashing
    - All the limitations mentioned above are mainly due to tight coupling between services and databases. This scalability issue can be solved with consistent hashing for data partitioning and load balancing.
    - Consistent Hashing at Load Balancer Level
        - Each request is hashed to determine its corresponding destination node. This works around the issue of tight coupling between services and the database, as it ensures each request is directed to the appropriate node responsible for storing and retrieving its data.
        - However, when a service/node is added to the swarm, data migration may needed since the hash ring may change depending on your implementation. Then, it may result in longer downtime as the system has to wait for the data migration step to finish first before functioning.

    - Consistent Hashing at Database Level
        - Before storing or retrieving data from the database, requests are hashed to identify the appropriate database and node for processing. This functions like a load balancer but at the database level rather than the service level. Consistent hashing for load balancer still allows tight coupling between services and database while consistent hashing for database requires the database itself to be a distributed system.
        - An implementation for the above is to use Cassandra, where each node can handle incoming requests directly. Cassandra manages data replication, partitioning, and consistency internally, simplifying system architecture.

5. Testing:

- Testing Coverage:
    - Individual business logic for each service.
    - Data transformations and mappings within each service.
    - Return values of function calls.
    - Error handling for each service.
    - Limited workflow steps within a service.
- Testing Limitations:
    - Unit tests are effective for validating individual components but do not work well for testing the interactions between different microservices. For instance, RabbitMQ message flows between services cannot be fully tested through unit tests. While mock tests can be used to test different ports and adapters, they can only confirm that methods like convertAndSend are invoked. However, they cannot verify whether the message was sent to the correct exchange or detect configuration issues within RabbitMQ.
    - Similarly, unit tests cannot verify the interactions between services are correct within the docker environment.
    - No testing for concurrent operations.
    - No stress testing and load testing can be used, these kinds of tests need to use tools like Grafana K6.

6. Architecture Justification:

HTTP and AMQP:

- All inter-service communications will be using RabbitMQ for decoupling services, asynchronous processing and scalability.
- All outer-service communications (e.g. to user, admin, and clients) will be using HTTP for ubiquity and ease of use.

Async Request for Inter-service Communication:

- Each service will include at least a pair of sender and receiver ports and adapters for handling requests and responses via RabbitMQ.
- This design ensures a non-blocking system, where services can continue processing independently without waiting for synchronous responses. If services are required to wait for responses, a single bottleneck in one service could slow down the entire system, causing delays across the entire chain. By avoiding this, the rest parts of the system remain resilient and efficient, even in the presence of bottlenecks.

Sync Request between Payment and Violation (The only exception):

- The inter-service communication between the Payment and Violation services bypasses RabbitMQ for two reasons:
    1. The Violation service already has an HTTP adapter that allows users to check their issued parking tickets. Reusing this adapter can avoid redundant implementation and leverage the existing infrastructure.
    2. The Payment service needs an immediate response from the Violation service to fetch the list of parking tickets. This list is used for calculating the total amount the visitor needs to pay. Using HTTP ensures the Payment service can retrieve the required data synchronously, fulfilling this dependency relationship.

Why is Monitor implemented within Gate Service?

- Since all the entry and exit requests and responses will finally returned through the Gate service, Gate is the only service which knows the current occupancy. Therefore, Monitor functionality implemented within Gate can reduce communication overhead.

Why is payment sales not implemented in Monitor as required in the documentation?

- Since Permit service provides RESTful ports and adapters for applying for and renewing permits. All the permit data are stored within the database associated with the Permite service. Therefore, it will be easier to retrieve the overall permit sales statistics from Permite service and can reduce overheads.

Why is Voucher implemented within the Payment Service module?

- The Voucher service allows the user to apply voucher issued by admins for free parking during a certain timeframe. The Payment service will always interact with the Voucher to check if the parking fee can be waived.
- Therefore, implementing Voucher and Payment in the same module can reduce communication overhead as Payment can directly access Voucher’s database.

Why is Violation NOT implemented within Payment Service like Voucher?

- Even though the Payment also always requires data from Violation when a visitor exists, the main consideration is that if Violation is implemented within the same module as the Payment service, when the Payment service fails, officers cannot issue tickets.
- This does not make sense as issuing tickets should not be affected by the failure of payment from the user's perspective.

Why does each business model within each service work like an independent service?

- Example:
    - In Gate service, only Gate Result Handler will make a call to the Monitor to update the current occupancy. In all other cases, Gate Request Handler, Gate Result Handler and Monitor do not interact with each other but each acts like an individual service.
- Loose Coupling
    - Each component operates independently and does not need to know about the implementation details of others and components interact only through well-defined interfaces/ports.
    - By doing so, changes to one component (e.g., how Dashboard stores data) won't affect the others which makes the system more maintainable and flexible.
- Single Responsibility Principle
    - Each component within a service has a clear, focused purpose:
    - For example:
        - Gate Request Handler: Handles incoming gate requests
        - Gate Result Handler: Send validation results (gate opening command) back to the corresponding gate and call the Monitor to update occupancy.
        - Dashboard: Manages and retrieves parking lot monitoring and status
    - This makes the code easier to understand, test, and modify and follows good software design principles.