43. UPI

Step 1: Outline Use Cases and Constraints

Use Cases:

1. **User Registration:** Users can install a UPI-enabled app, create a Virtual Payment Address (VPA), and link their bank accounts.
2. **Peer-to-Peer (P2P) Transfers:** Users can send money to other users by entering the recipient's VPA or mobile number.
3. **Merchant Payments:** Users can make payments to merchants by scanning QR codes or entering merchant VPA.
4. **Transaction Authorization:** Users authenticate transactions using their mobile PIN (MPIN) or biometric authentication.
5. **Transaction Settlement:** UPI ensures funds are transferred from the sender's bank account to the recipient's bank account in real-time.
6. **Transaction Notifications:** Both sender and receiver get notified of successful transactions.
7. **Transaction History:** Users can view their transaction history through the mobile app.

Out of Scope:

- Cross-border transactions.
- Third-party integrations with external wallets or payment providers.
- Advanced fraud detection mechanisms.
- Handling recurring payments (future versions).
- Integration with all available banks (limited to major banks only).

Constraints and Assumptions:

- **Concurrency:** The system should support a high volume of transactions (millions of users and transactions).
- **Security:** UPI transactions must comply with security standards such as encryption and two-factor authentication (MPIN, OTP).
- **Regulatory Compliance:** The design must comply with RBI and NPCI guidelines for UPI.
- **Availability:** UPI is expected to be available 24/7, with real-time settlement.
- **Performance:** The system should process each transaction in under a few seconds to provide real-time confirmation.
- **Bank Integration:** The system needs to be capable of integrating with various bank APIs through NPCI.
- **Mobile Platforms:** UPI will be implemented as a mobile-first solution (Android and iOS apps).
- **Data Persistence:** Transaction history and user details should be stored securely in databases.

Step 2: High-Level Design

Key Components:

1. **User Service:** Manages user registration, authentication, and account linking.
2. **VPA Management Service:** Manages Virtual Payment Address (VPA) creation and association with user accounts.
3. **Transaction Service:** Handles the creation and processing of transactions, including routing requests to banks and confirming settlements.

4. **Authentication Service:** Responsible for handling user authentication (e.g., MPIN or biometric) for transactions.
5. **Notification Service:** Sends notifications about the transaction status to users (e.g., SMS, email, or push notifications).
6. **Bank Service:** Interfaces with the bank’s API to verify account balances, process payments, and authorize transactions.
7. **NPCI (National Payments Corporation of India):** Coordinates the settlement of transactions between different banks.
8. **QR Code Generator/Scanner**: Handles the generation and scanning of QR codes for merchant payments.
9. **Transaction History Service**: Manages user transaction records and transaction history.

Data Flow:

1. The user registers via a mobile app and creates a VPA.
2. The user initiates a transaction by entering the recipient’s VPA or scanning a QR code.
3. The transaction service validates the transaction and sends the request to the sender’s bank service for verification.
4. The bank service checks for balance and authorization, then routes the transaction through NPCI.
5. NPCI processes the interbank transaction and sends back confirmation to the sender's and recipient's banks.
6. Once the transaction is confirmed, the notification service informs both parties.

Step 3: Design Core Components

1. User Service

- Responsible for user registration, VPA creation, and linking with bank accounts.

```
public class UserService {
```

2. Transaction Service

- Handles the creation, processing, and routing of transactions between the sender and recipient.

```
public class TransactionService {
    private BankService bankService;
    private NotificationService notificationService;

    public TransactionService(BankService bankService, NotificationService notificationService) {
        this.bankService = bankService;
        this.notificationService = notificationService;
    }

    public boolean initiateTransaction(String senderVPA, String receiverVPA, double amount) {
        User sender = UserService.getUserByVPA(senderVPA);
        User receiver = UserService.getUserByVPA(receiverVPA);

        if (sender == null || receiver == null) {
            return false;
        }

        // Check sender's balance
        if (bankService.checkBalance(sender, amount)) {
            // Deduct from sender and add to receiver
            bankService.processPayment(sender, receiver, amount);
            // Notify users
            notificationService.sendTransactionConfirmation(sender, receiver);
            return true;
        }

        return false;
    }
}
```

3. Bank Service

- Handles interactions with the bank’s API to verify and process transactions.

```
public class TransactionService {
    private BankService bankService;
    private NotificationService notificationService;

    public TransactionService(BankService bankService, NotificationService notificationService) {
        this.bankService = bankService;
        this.notificationService = notificationService;
    }

    public boolean initiateTransaction(String senderVPA, String receiverVPA, double amount) {
        User sender = UserService.getUserByVPA(senderVPA);
        User receiver = UserService.getUserByVPA(receiverVPA);

        if (sender == null || receiver == null) {
            return false;
        }

        // Check sender's balance
        if (bankService.checkBalance(sender, amount)) {
            // Deduct from sender and add to receiver
            bankService.processPayment(sender, receiver, amount);
            // Notify users
            notificationService.sendTransactionConfirmation(sender, receiver);
            return true;
        }
```

```
            return false;
        }
}
```

4. Notification Service

- Sends transaction confirmations and alerts to users.

```
public class NotificationService {
        public void sendTransactionConfirmation(User sender, User receiver) {
            // Send confirmation to both users (SMS, Push Notification, etc.)
            System.out.println("Transaction completed: " + sender.getVPA() + " -> " + receiver.getVPA());
        }
}
```

5. VPA Management Service

- Allows users to manage their VPAs.

```
public class VPAService {
        private Map < String, String > vpaRegistry;

        public VPAService() {
            vpaRegistry = new HashMap < > ();
        }

        public void registerVPA(String phone, String vpa) {
            vpaRegistry.put(phone, vpa);
        }

        public String getVPA(String phone) {
            return vpaRegistry.get(phone);
        }
}
```

Step 4: Scale the Design

Scalability Considerations:

1. **Database Sharding:** The system should use database sharding to handle user data and transaction records for millions of users.
2. **Load Balancing:** Use a load balancer to distribute incoming requests to multiple app servers, ensuring high availability.
3. **Event-Driven Architecture:** Use Kafka or another message queue to handle asynchronous notifications and event processing.
4. **Caching**: Use Redis or Memcached to store user balances and transaction history for faster access.
5. **Microservices**: The system can be broken down into smaller services such as:
   - User Management Service
   - Transaction Processing Service
   - Notification Service
   - Bank Integration Service
   - VPA Service
6. **Security**: Implement encryption for sensitive data (e.g., UPI PINs, account details) and two-factor authentication (OTP, MPIN).
7. **Monitoring and Auditing:** Use tools like Prometheus, Grafana for monitoring, and ELK stack for logging.

By considering these scalability strategies, the UPI system can handle millions of concurrent transactions and users with minimal latency.