Payment System Socket Communication Protocol

Overview

This document outlines the WebSocket communication protocol between the Android client application and the backend server for the payment system. It specifies message formats, expected responses, and the flow of communication during different stages of the payment process.

Connection Details

- **Connection Type**: WebSocket (RFC 6455)
- **Client Implementation**: OkHttp WebSocket Client (version 4.11.0)
- **Message Format**: JSON
- **Transport Security**: Secure WebSocket (WSS) recommended for production

Connection Establishment

1. The Android client connects to the backend server using WebSocket
2. Upon successful connection, the server should immediately send the initial screen state (typically the amount selection screen)
3. The client maintains a single persistent connection for the duration of the payment process

Message Format

All messages follow this standard JSON structure:

```
{
  "messageType": "[SCREEN_CHANGE | USER_ACTION | ERROR | STATUS_UPDATE]",
  "screen": "[SCREEN_IDENTIFIER]",
  "data": {
    // Optional data specific to the screen or action
  },
  "transactionId": "unique-transaction-identifier",
  "timestamp": 1619456789000
}
```

Message Type Values

- `SCREEN_CHANGE`: Server instructs client to display a specific screen
- `USER_ACTION`: Client notifies server of user interaction
- `ERROR`: Server notifies client of an error condition
- `STATUS_UPDATE`: Server sends non-screen-changing status information

Screen Identifier Values

- `AMOUNT_SELECT`: Select payment amount
- `KEYPAD`: Custom amount entry via keypad
- `PAYMENT_METHOD`: Select payment method
- `PROCESSING`: Processing payment
- `SUCCESS`: Transaction successful
- `FAILED`: Transaction failed
- `LIMIT_ERROR`: Amount limit exceeded
- `PRINT_TICKET`: Printing ticket
- `COLLECT_TICKET`: Collect printed ticket
- `THANK_YOU`: Final thank you screen
- `DEVICE_ERROR`: Terminal error condition

Transaction Flow and Messages

1. Initial Connection

Upon successful connection, server sends:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "AMOUNT_SELECT",
  "data": {
    "amounts": [20, 40, 60, 80, 100],
    "currency": "£",
    "showOtherOption": true
  },
  "transactionId": "initial",
  "timestamp": 1619456789000
}
```

2. Amount Selection

When user selects a predefined amount, client sends:

```
{
  "messageType": "USER_ACTION",
  "screen": "AMOUNT_SELECT",
  "data": {
    "selectedAmount": 60,
    "selectionMethod": "PRESET_BUTTON"
  },
  "transactionId": "T123456",
  "timestamp": 1619456790000
}
```

If user selects "Other" option, client sends:

```
{
  "messageType": "USER_ACTION",
  "screen": "AMOUNT_SELECT",
  "data": {
    "selectedAmount": -1,
    "selectionMethod": "OTHER"
  },
  "transactionId": "T123456",
  "timestamp": 1619456790000
}
```

Server responds with keypad screen:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "KEYPAD",
  "data": {
    "currency": "£",
    "minAmount": 10,
    "maxAmount": 300
  },
  "transactionId": "T123456",
  "timestamp": 1619456791000
}
```

3. Custom Amount Entry

When user enters a custom amount via keypad, client sends:

```
{
  "messageType": "USER_ACTION",
  "screen": "KEYPAD",
  "data": {
    "selectedAmount": 75,
    "selectionMethod": "KEYPAD"
  },
  "transactionId": "T123456",
  "timestamp": 1619456792000
}
```

4. Payment Method Selection

Server responds with payment method selection screen:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "PAYMENT_METHOD",
  "data": {
    "methods": ["DEBIT_CARD", "PAY_BY_BANK"],
    "allowCancel": true,
    "currency": "£"
  },
  "transactionId": "T123456",
  "timestamp": 1619456793000
}
```

When user selects payment method, client sends:

```
{
  "messageType": "USER_ACTION",
  "screen": "PAYMENT_METHOD",
  "data": {
    "selectedMethod": "DEBIT_CARD"
  },
  "transactionId": "T123456",
  "timestamp": 1619456794000
}
```

5. Processing Payment

Server responds with processing screen:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "PROCESSING",
  "data": {},
  "transactionId": "T123456",
  "timestamp": 1619456795000
}
```

6. Transaction Result

For successful transactions, server sends:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "SUCCESS",
  "data": {
    "showReceipt": true
  },
  "transactionId": "T123456",
  "timestamp": 1619456796000
}
```

For failed transactions, server sends:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "FAILED",
  "data": {
    "errorMessage": "Card declined by issuer",
    "errorCode": "CARD_DECLINED"
  },
  "transactionId": "T123456",
  "timestamp": 1619456796000
}
```

7. Limit Error

If amount exceeds daily limit, server sends:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "LIMIT_ERROR",
  "data": {
    "limit": 300,
    "remaining": 200,
    "currency": "£"
  },
  "transactionId": "T123456",
  "timestamp": 1619456796000
}
```

8. Ticket Printing

For ticket printing, server sends:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "PRINT_TICKET",
  "data": {},
  "transactionId": "T123456",
  "timestamp": 1619456797000
}
```

9. Collect Ticket

After printing completes, server sends:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "COLLECT_TICKET",
  "data": {},
  "transactionId": "T123456",
  "timestamp": 1619456798000
}
```

10. Thank You Screen

Final screen in the flow, server sends:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "THANK_YOU",
  "data": {},
  "transactionId": "T123456",
  "timestamp": 1619456799000
}
```

11. Device Error

If terminal has an error, server sends:

```
{
  "messageType": "SCREEN_CHANGE",
  "screen": "DEVICE_ERROR",
  "data": {
    "errorMessage": "Printer out of paper",
    "errorCode": "PRINTER_ERROR"
  },
  "transactionId": "T123456",
  "timestamp": 1619456800000
}
```

Transaction Timeout Handling

If a transaction times out, server sends:

```
{
  "messageType": "ERROR",
  "screen": "CURRENT_SCREEN",
  "data": {
    "errorMessage": "Transaction timed out",
    "errorCode": "TIMEOUT"
  },
  "transactionId": "T123456",
  "timestamp": 1619456800000
}
```

Connection Error Handling

- The client implements automatic reconnection with exponential backoff
- If connection is lost, client attempts to reestablish connection
- Server should maintain transaction state and allow resumption when possible

Security Considerations

1. **Authentication**:
    - Initial connection should include authentication if required
    - Consider using token-based authentication in WebSocket handshake
2. **Data Validation**:
    - Server must validate all client-sent data, particularly amounts and transaction IDs
    - Client should validate server responses before displaying screens
3. **Transport Security**:
    - Use WSS (WebSocket Secure) in production environments
    - TLS 1.2 or higher is recommended

Implementation Notes for Backend Developer

WebSocket Server Setup

The backend should implement a WebSocket server that:

1. Accepts connections from the Android client
2. Maintains session state for each connected client
3. Processes messages according to the protocol defined above
4. Sends appropriate responses based on the payment processing flow

Example Server Implementations

Recommended frameworks:

- Node.js: `ws`, `socket.io`
- Java: Spring WebSocket, Jetty
- Python: `websockets`, Django Channels
- .NET: SignalR

Testing

A test client is available in the Android app repository that simulates the client-side WebSocket connections. Use this for development and testing.

Logging Requirements

The backend should log:

1. Connection events (connect, disconnect)
2. Message receipt (inbound from client)
3. Message transmission (outbound to client)
4. Errors and exceptions
5. Transaction lifecycle events

Client Technical Details

The Android client uses:

- OkHttp WebSocket implementation (version 4.11.0)
- JSON serialization with Moshi (version 1.14.0)
- Kotlin coroutines for asynchronous handling
- StateFlow for UI state management
- Animation transitions between screens
- GIF animations for various process stages

This document covers the essential communication protocol between the Android client and backend server. Adjustments may be needed based on specific backend implementation details or additional business requirements.