**System Architecture Overview**

As the Software Architect, I'll outline a software architecture plan for the described client/server application which includes serialisation of data, file handling, encryption, and network communication. This plan will follow a modular approach for ease of development, testing, and maintenance. Here is a high-level architecture breakdown:

1. **Features:**

* **Dictionary Serialisation**: The client can serialise dictionaries into multiple formats, including JSON, binary, and XML.
  + Example of dictionary:
  + my\_dict = {"artist\_name": "Rihanna", "real\_name": "Robyn Rihanna Fenty", "nationality": "Barbadian", "age": 36}
* **File Encryption**: Optionally encrypt text files before sending them to the server to ensure data security.
* **Configurable Server**: The server can be configured to print or save the processed data based on user preferences.
* **Comprehensive Testing**: Unit tests and feature tests ensure the system's reliability and correctness.

1. **Utility Modules**:
   1. **encrypter.py**: Handles encryption and decryption of text files.
   2. **file\_handler.py**: Provides functions to read from and write to text files.
   3. **serialiser.py**: Contains functions to serialise and deserialise dictionaries in binary, JSON, and XML formats.

**3. Data Flow:**

1. **Client Side(client.py):**
   * User selects the serialisation format (JSON, binary, XML) for the dictionary.
   * Optionally, the user decides whether the text file should be encrypted.
   * Serialise the dictionary using the chosen format.
   * Encrypt the file content if required.
   * Send serialised data and file content to the server using the network module.
2. **Server Side(server.py):**
   * Receive the serialised dictionary and file content.
   * Deserialise the dictionary if required.
   * Decrypt the file content if it's encrypted.
   * Print the contents to the console and/or write them to a file based on the configuration.

**4. Error Handling:**

1. Use try-except blocks to handle exceptions from file operations, serialisation, encryption, and network communications.
2. Validate user inputs for serialisation format and encryption choice.

**5. Testing:**

1. **Unit Tests**: Ensure the correctness of individual components.

**Unit Test Design**

**Unit Tests for client, server, and Utility Modules:**

• Test\_client:

o Test serialisation of dictionaries into JSON, binary, and XML formats.

o Test the encryption of text files.

o Test the sending of data to the server.

• Test\_server:

o Test the server's ability to receive and process serialised data.

o Test the deserialisation of data in JSON, binary, and XML formats.

o Test the server's ability to handle unsupported format.

o Test the server's ability to handle unencrypt content.

o Test the server's ability to handle encrypted content.

o Test the server's ability to handle missing of content.

o Test the decryption of received text files.

o Test the decryption of received text files.

o Validate the server's functionality to print or save data.

• Test\_encrypter:

o Test encryption and decryption functions.

o Ensure that encrypted text can be correctly decrypted.

• Test\_file\_handler:

o Test file read and write operations.

o Ensure data integrity during file operations.

• Test\_serialiser:

o Test serialisation and deserialisation of dictionaries in different formats.

o Verify the correctness of the serialised output.

1. **Feature Tests**: Validate the end-to-end functionality of the system.

**Feature Test Design**

The feature tests validate the complete client/server interaction. The tests cover scenarios with different serialisation formats and encryption options to ensure the system behaves as expected.

**Feature Test Scenarios:**

* Test Case 1: JSON Format with Encryption
  + Client: Create and serialise a dictionary to JSON. Encrypt the text file.
  + Server: Receive, decrypt, and deserialise the JSON data. Verify the content.
  + Result: Success
* Test Case 2: JSON Format without Encryption
  + Client: Create and serialise a dictionary to JSON. Do not encrypt the text file.
  + Server: Receive and deserialise the JSON data. Verify the content.
  + Result: Success
* Test Case 3: Binary Format with Encryption
  + Client: Create and serialise a dictionary to binary. Encrypt the text file.
  + Server: Receive, decrypt, and deserialise the binary data. Verify the content.
  + Result: Success
* Test Case 4: Binary Format without Encryption
  + Client: Create and serialise a dictionary to binary. Do not encrypt the text file.
  + Server: Receive and deserialise the binary data. Verify the content.
  + Result: Success
* Test Case 5: XML Format with Encryption
  + Client: Create and serialise a dictionary to XML. Encrypt the text file.
  + Server: Receive, decrypt, and deserialise the XML data. Verify the content.
  + Result: Success
* Test Case 6: XML Format without Encryption
  + Client: Create and serialise a dictionary to XML. Do not encrypt the text file.
  + Server: Receive and deserialise the XML data. Verify the content.
  + Result: Success

**Example Feature Test (FeatureTest.docx):**

Test Case 1: JSON Format with Encryption

Client:

Create a dictionary: {"name": "John", "age": 30}

Serialise the dictionary to JSON.

Encrypt the text file content.

Send the serialised data and encrypted text file to the server.

Server:

Receive the serialised JSON data.

Decrypt the text file content.

Deserialise the JSON data.

Verify that the data matches the original dictionary.

Print or save the data based on the configuration.

Result: Success

**Proposed test chart presentation:**

Feature Test:

|  |  |  |  |
| --- | --- | --- | --- |
| Case | Format | Encryption | Result |
| 1 | JSON | YES | Success |
| 2 | JSON | NO | - |
| 3 | BINARY | YES | - |
| 4 | BINARY | NO | - |
| 5 | XML | YES | - |
| 6 | XML | NO | - |

**5.Documentation and Version Control:**

**Readme.md:** Overview of the project, how to set it up and run it.

**requirements.txt:** List of all external libraries used.

* **Code Comments and Docstrings:** Detailed explanations of the functionality of code segments and modules.
* Use Git for version control with clear commit messages reflecting changes.

**7. Collaboration Tools:**

* **GitHub:** Host the repository, manage branches for different development stages, merge requests for code review.
* **Microsoft Teams:** Communicate within the team, share files, and conduct meetings.

**Conclusion:**

This plan should be robust enough to manage the development of a scalable client-server application that handles data serialisation, encryption, and network communication efficiently.