Hands-On Exercise: Class Based Communication Library

Scenario

You have inherited a piece of functional code that performs inter-process communication (IPC).

Note: In this context we are going to assume inter-process means either separate loops/threads on the same application or separate applications that may be running on different computers.

The existing code implements two libraries for IPC. These libraries are designed to communicate lossless commands from one process to another (1:1 bidirectional communication). The two provided libraries are:

- *Local.lvlib*: It relies on LabVIEW Queues to exchange packets/commands between two loops running on the same application.
- *Network.lvlib*: It relies on LabVIEW Network Streams to exchange packets/commands between two processes using TCP/IP. This allows communication of different applications that may be on separate computers.

The commands are being sent as a part of a packet. The raw representation of this packet is an array of U8 elements with fixed size. Initially, there is only one packet type with the next format:

- The counter value that increments for each packet that is sent.
- Timestamp captured before sending the packet.
- Command ID is provided by the user and identifies the packet.

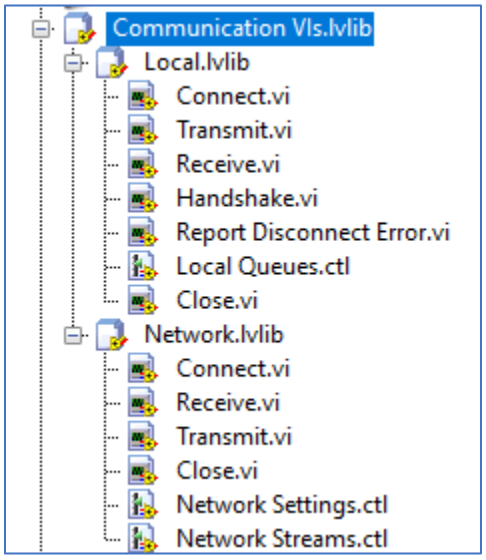
Counter (2 bytes)	Timestamp (8 bytes)	Command ID (2 bytes)

You are being asked to take this existing code and migrate it to use an object-oriented approach. The new code can use inheritance to define a parent library with a generic interface that can be reused and extended by specific implementations (local, network and others in the future).

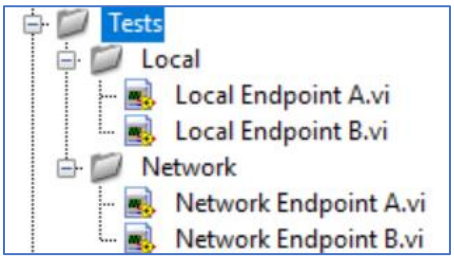
Steps to Follow

Explore Provided Code

1. Fork the existing github repository with the files for this exercise.
    a. Repo: https://github.com/agomez08/lvoop_exercise
2. Clone your repo and explore the *Communication Vis.lvlib* that contains the code that will be migrated. You will notice that both libraries already have a somehow generic interface with the next actions:
    a. *Connect:* The two processes to communicate setup their connections and execute some sort of handshake. Provide an input parameter with *timeout* in milliseconds that configures the maximum time to wait for connection.
    b. *Transmit:* A process sends a packet of data to the peer process.
    c. *Receive:* A process attempts to receive a packet of data from the peer process.
    d. *Close:* Disconnect and destroy connections with the peer.



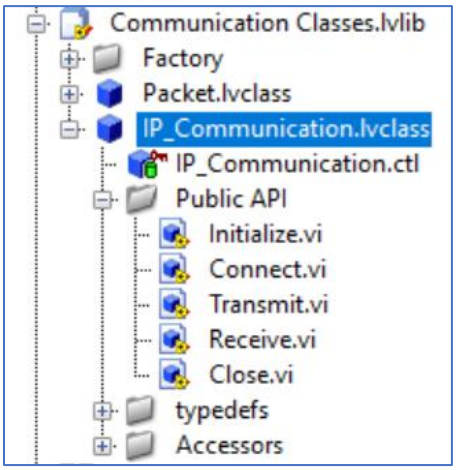
3. Test the provided communication libraries.
    a. The tests for Local and Network can be found on the *Tests* virtual folder.



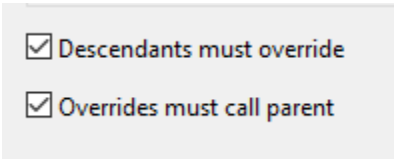
    b. Run *Local Endpoint A.vi* along with *Local Endpoint B.vi.* Press the button to send commands and notice how they get updated on the receiving end of the other endpoint.
    c. What happens when you stop one of the VIs? You will notice that the other endpoint will throw an error indicating the connection has been closed.

d. What happens if you run only one of the Vis? You will notice that an error will be thrown after the timeout expires indicating the *Connect.vi* was not successful.
e. Perform similar experiments running the *Network Endpoint A.vi* along with *Network Endpoint B.vi.*

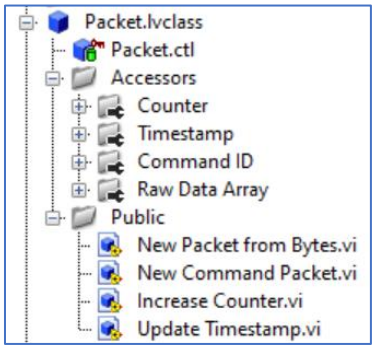
4. Another programmer has already created a generic class that defines the interface to follow for the IPC libraries. Take some time to explore this class.



a. Explore the private data of the class.
b. Explore the Public API VIs. As you will see, the class implements some generic functionality that can be reused by children.
c. Explore the accessor provided to access the *General Settings* from the private data.
d. Explore the properties of the class (right click >> Properties).
    i. Review the *Item Settings* for the elements in the Public API. You will notice that *Connect.vi, Transmit.vi, Receive.vi* and *Close.vi* will force the children to override them and to call back the parent method to ensure the generic functionality is executed.

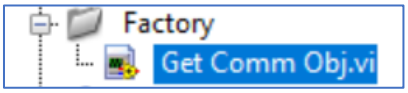


5. An abstraction of the packet has also been created through the *Packet.lvclass*.



a. Explore the private data of the class.
b. Explore the Public methods exposed in this class and its accessors.

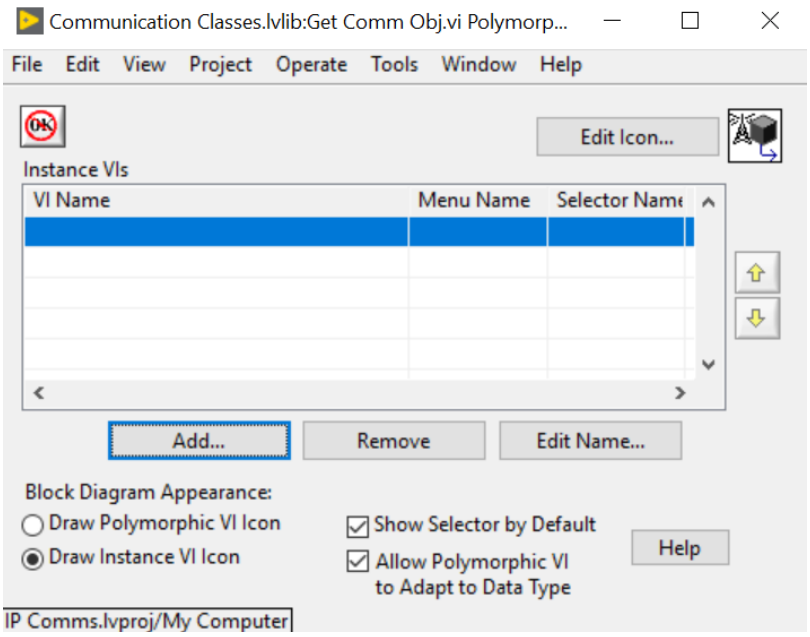
6. The library also contains a *Factory* virtual folder with a polymorphic VI. The polymorphic VI can be configured with Vis that can provide a communication object for a given type (similar to a constructor).



Implementing IPC Libraries using OOP

You will now migrate the existing libraries to use an OOP approach.

1. Create one class for each of the libraries: Network and Local.
   a. Configure these classes to inherit from the *IP_Communication.lvclass*.
   b. Configure a custom icon template for each of these classes.
   c. Configure a custom wire appearance for each of these classes.
2. Use the existing code to implement the *Connect.vi, Transmit.vi, Receive.vi* and *Close.vi* overrides for each of these classes.
3. Create a new VI that can provide a communication object for each of the classes (similar to a constructor) from the settings provided by the user (connection name, endpoint tag, IP address, etc). You can use the parent method *Initialize.vi* from this VI.
   a. Add this constructor VI to the factory polymorphic VI *Get Comm Obj.vi*.



4. Create tester VIs like the ones used to test the original libraries to verify your OOP implementations work as expected.