Application Level Protocol Design

Objectives

In this lecture we will discuss issues related to application level protocol design. We start by discussing the atomic units used by any protocol (which we term "messages"), revisit our discussion of encoding and finish by presenting a reusable, protocol independent, TCP server, accompanied by a LinePrinting protocol implementation.

Protocol Definition

A **protocol** is a set of rules, governing the communication details between two parties (in our case, processes). Protocols exist in a myriad of different forms and levels; from protocols governing how to exchange bits across a wire to protocols governing administration of super computers. In this lecture we focus on application level protocols, which define the interaction between two computer applications.

Each protocol defines the following sets of rules:

- The **syntax** of the communication: how do we phrase the information we exchange.
- The **semantics** of the communication: what is the proper response for each information datum received.
- The **synchronization** of the communication: whose turn it is to speak (given the above defined semantics).

Almost all protocols follow a very simple skeleton. The parties involve exchange information by using messages, which define the syntax. The communication usually begins when one party sends an initiation message (a hand-shake) to the other party. The synchronization used is usually very simple, where each party sends one message in a round robin fashion. The difference between most protocols is the syntax used for messages, and the semantics of the protocol.

An example for a protocol is HTTP, which stands for Hyper Text Transfer Protocol, and govern the details of exchanging special text files over the network. We give a brief and simple (but not complete) description of the protocol:

- synchronization: the client initiates the connection, sends a single request, and receive the reply from the server.
- syntax: text based, see rfc2616 .
- semantics: the server either sends to the client the page asked for, or returns an error.

In the rest of this lecture we will discuss the syntax and semantics aspects of protocols. We will assume that synchronization works in a round robin fashion, e.g., each party sends one message at a time.

Message Format

To define the syntax of a protocol, we employ the concept of **messages**. A message is the atomic unit of data exchanged throughout the protocol. We can think of a message as a letter, which contain all the information one party needs to send to the other party. We leave the discussion of what exactly is written inside the letter to later, and first concentrate on the delivery mechanism.

Framing

When using a streaming protocol such as TCP, the need to separate between different messages arises; as all messages are sent on the same stream, one after the other, the receiver should be able to distinguish between different messages. The usual solution is to use message *framing*. Framing a message is taking the content of the message, and encapsulating it in a frame (continuing our metaphor of a letter, think of an envelop). The sender and receiver agree on the framing method beforehand (the framing is part of the message format/protocol). The framing used should enable the receiver to easily discover, given a stream of bytes, where a message starts and where it ends.

The simplest example of a framing protocol, when sending strings, is using a special character, the `FRAMING` character (e.g., a line break). In other words, each message is framed by two `FRAMING` characters, one at the beginning and one at the end. Care should be taken that each message will not contain a `FRAMING` character in its text!

A different framing protocol may be achieved by adding a special tag at the start of the message, and a special tag at the end. When sending strings of text, a message can be framed using `<begin>` and `<end>` strings. Again, care must be taken to avoid having `<begin>` and `<end>` as part of the message body.

Yet another framing protocol can be achieved by employing a variable length message format, namely a special tag or character are used to mark the start of a frame while the message itself contains the information on the message's length.

Binary Data

Many standard protocols exchange data in textual form – that is, they send and receive strings of characters, in an agreed upon character encoding, often UTF-8. The advantage of using text in a protocol is that it is very easy to document and to debug - just print the messages exchanged by the client and server and you understand what is happening. The limitation of text-based protocols is that it becomes difficult to send non-textual data. For example: how do we send a picture? a video? an audio file? In this context, non-textual data is called

binary data. (Of course, all data is eventually encoded in "binary" format, as a sequence of bits - so this usage of "binary data" is special – it means data that cannot be encoded as a readable string of characters).

Sending binary data in raw binary format in a stream protocol such as TCP is dangerous. As the binary data may contain any byte sequence, the binary data may corrupt our framing protocol. There are at least two solutions you may employ:

- Encoding: you can encode the binary data by using a well established encoding algorithm. For example, Base64 encoding is one of the better alternatives. Base64 encoding encodes binary data into a string, converting every 2 bytes sequence from the binary data into 3 ASCII characters. Base64 is used by many "standard" protocols (such as email protocols to encode file attachments which may contain any type of data).

  In C++, the Boost library includes a module to perform encoding/decoding of arbitrary byte arrays into/from Base64 encoded ASCII data. This functionality is modeled as a sort of stream "filter" that performs encode/decode on all data flowing through the stream in the serialization package.

  In Java, the standard java library includes a Base64 encoder/decoder you can use.

  The advantage of this encoding is that any stream of bytes can now be "framed" as ASCII data - regardless of the character encoding used by the protocol. The disadvantage is that there is a cost in the size of the message, which is increased by 50%.

- Devising a variable length message format.

Encoding

An integral part of the message format is the encoding used to send strings. There are many standard ways to encode a string into a byte sequence. However, in this course we will use UTF-8 as our encoding scheme.

Protocol and Server Separation

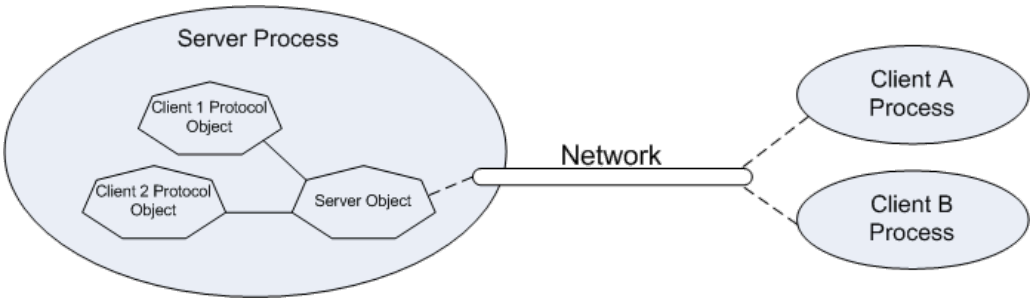
We have already established that when designing complex systems, code reuse is one of our design goals. In the context of networking, it would be especially useful to have a generic implementation of a server, which handles all the communication details, and a generic protocol interface, which handles incoming messages, implements the protocol's semantics and generates the reply messages. It follows that the protocol object is the object in charge of implementing the expected behavior of our server, namely what actions should be performed upon the arrival of a request. Note that requests may be correlated one to another, meaning the protocol should save an appropriate state per client.

For example, protocols often require user authentication (login), so that only authorized users can perform certain actions. In this case, the protocol is stateful - this means that the protocol serving the requests of a client can be in at least 2 distinct states: authenticated (user has already logged in) or non-authenticated (user has not yet provided his login and password). Depending on the state of the protocol object, the behavior of the protocol object will be different (if an authenticated user asks the protocol object to perform an action, it will be done, while a non-authenticated user asking the same action will receive an error code).

The key to producing such a generic server implementation is to carefully separate the different tasks the server must perform. The following actions can be distinguished:

- Accept new connections from new clients.
- Receive new bytes from connected clients.
- Parse incoming bytes from connected clients into separate messages (an operation known as "de-serialization" or "unframing" or "decoding").
- Dispatch a message to the right method on the server side to execute the requested operation.
- Send back an answer to a connected client after an action has been executed.

We now describe a software architecture that separates these various tasks into separate interfaces:



The key participants in this architecture are:

- the `MessageEncoderDecoder`, which implements the protocol's syntax, encoding and decoding messages from and to bytes.

- the `MessagingProtocol`, which implements the protocol's semantics; e.g., handling the received messages and generating the appropriate responses.

Using these interfaces, we can create a generic server - i.e, a protocol agnostic server. Note though, that for simplicity the general protocol that we defined above assumes that clients will send messages to the server and the server can respond. We can think about protocols which has the server sending messages to the clients and the clients respond or even mixed protocols. Extending the interfaces to support these types of protocols is fairly simple once you understand their implementation and usage.

Interfaces

We implement the separation between the protocol and the server by employing the following design. First, we define a message. A message can be encoded in various ways: Base64 message, XML message or text message. For simplicity, we will describe messages encoded as plain UTF-8 text.

Next, we define the framing of messages, that is, the delimiters between messages when messages are sent through a common stream.

Finally, we define the protocol interface which handles each individual message.

MessageEncoderDecoder

The MessageEncoderDecoder interface is in charge of parsing a stream of bytes into a stream of messages and backwords.

The context of this interface is the following: the server accepted a new connection from a client. The server creates an instance of a `BlockingConnectionHandler` object that will handle all incoming messages from this client. The `ConnectionHandler` object maintains the state of the connection for the specific client which it serves (for example, if the user performed "login", the `ConnectionHandler` object will remember this in its state). The `ConnectionHandler` also has access to the `Socket` connecting the server to the client process.

Since we are describing a TCP server, the `Socket` connection is viewed as a pair of `InputStream` and `OutputStream`. These streams are streams of bytes – that is, as far as TCP is concerned, the client and the server exchange a bunch of bytes.

The `MessageEncoderDecoder` interface is a filter that we put between the `Socket` input stream and the protocol. The protocol does not access the input/output stream directly - it only handle application level messages while the `MessageEncoderDecoder` responsible to translate them to and from bytes. This way, one can use the same protocol under different message formats and reuse message formats for different protocols.

The decoding process of the `MessageEncoderDecoder` works in a byte-by-byte fashion. Every time we receive a new byte from the server we will give it to the `MessageEncoderDecoder` if this byte, together with the previous bytes which were passed to the `MessageEncoderDecoder` represents a full message, the `MessageEncoderDecoder` will return it to us and the decoding process will restart.

```
                                          17 lines ...
1. public interface MessageEncoderDecoder<T> {
2.
3.     /**
4.      * add the next byte to the decoding process
5.      *
6.      * @param nextByte the next byte to consider for the currently decoded message
7.      * @return a message if this byte completes one or null if it doesnt.
8.      */
9.     T decodeNextByte(byte nextByte);
10.
11.    /**
12.     * encodes the given message to bytes array
13.     * @param message the message to encode
14.     * @return the encoded bytes
15.     */
16.    byte[] encode(T message);
17.
18. }
```

Messaging Protocol

We define next the protocol interface. The `MessagingProtocol` interface operates in the following context:

A `ConnectionHandler` instance wraps together: the socket connected to the client; the MessageEncoderDecoder which splits incoming bytes from the socket into messages. The next step is to pass the incoming messages from the client to the `MessagingProtocol` which will now execute the action requested by the client. The task of the `MessagingProtocol` is to look at the message and decide what should be done. This decision may depend on the state of the connection (remember the example of the "authenticated" protocol). Once the action is performed, we will need to send an answer to the client. So we expect to get an answer back from the `MessagingProtocol`.

We model this behavior in the following interface:

```
                                                14 lines ...
1. public interface MessagingProtocol<T> {
2.
3.     /**
4.      * process the given message
5.      * @param msg the received message
6.      * @return the response to send or null if no response is expected by the client
7.      */
8.     T process(T msg);
9.
10.    /**
11.     * @return true if the connection should be terminated
12.     */
13.    boolean shouldTerminate();
14.
15. }
```

Note that we allow the protocol to use message any type of message (the type argument T). This means that the operation of Serialization and Deserialization (encode/decode complex parameters to/from Strings) will be performed by the MessageEncoderDecoder - which yield a good separation of concerns.

Implementations

The Connection Handler

We now put things together into the `ConnectionHandler`. `ConnectionHandler` is designed to run by its own thread. It handles one connection to one client for the whole period during which the client is connected (from the moment the connection is accepted, until one of the sides decides to close the connection). It therefore is modeled as a `Runnable` class.

The `ConnectionHandler` holds references to the TCP socket connected to the client, a `MessageEncoderDecoder` and an instance of the `MessagingProtocol`.

The following runnable connection handler is generic, and works flawlessly with any implementation of a messaging protocol.

```
                                                35 lines ...
1. public class ConnectionHandler<T> implements Runnable {
2.
3.     private final MessagingProtocol<T> protocol;
4.     private final MessageEncoderDecoder<T> encdec;
5.     private final Socket sock;
6.
7.     public ConnectionHandler(Socket sock, MessageEncoderDecoder<T> reader, MessagingProtocol<T> protocol) {
8.         this.sock = sock;
9.         this.encdec = reader;
10.        this.protocol = protocol;
11.    }
12.
13.    @Override
14.    public void run() {
15.
16.        try (   Socket sock = this.sock; //just for automatic closing
17.                BufferedInputStream in = new BufferedInputStream(sock.getInputStream());
18.                BufferedOutputStream out = new BufferedOutputStream(sock.getOutputStream())) {
19.
20.            int read;
21.            while (!protocol.shouldTerminate() && (read = in.read()) >= 0) {
22.                T nextMessage = encdec.decodeNextByte((byte) read);
23.                if (nextMessage != null) {
24.                    T response = protocol.process(nextMessage);
25.                    if (response != null) {
26.                        out.write(encdec.encode(response));
27.                        out.flush();
28.                    }
29.                }
30.            }
31.
```

```
32.         } catch (IOException ex) {
33.             ex.printStackTrace();
34.         }
35.     }
36. }
```

To implement a TCP server on the basis of this design, we now only need to implement our specific framing handler (the message encoder/decoder) and the specific protocol we wish to use. To continue our example of echo server, we will now illustrate how to implement the echo printing protocol in this architecture.

MessageEncoderDecoder

For our protocol, we use a framing method based on a single character delimiter. We assume that we have a stream of messages, delimited by `FRAMING`, specifically, we will use the character `'\n' (newline)`.

```
                                          37 lines ...
1. public class LineMessageEncoderDecoder implements MessageEncoderDecoder<String> {
2.
3.     private byte[] bytes = new byte[1 << 10]; //start with 1k
4.     private int len = 0;
5.
6.     @Override
7.     public String decodeNextByte(byte nextByte) {
8.         //notice that the top 128 ascii characters have the same representation as their utf-
   8 counterparts
9.         //this allow us to do the following comparison
10.        if (nextByte == '\n') {
11.            return popString();
12.        }
13.
14.        pushByte(nextByte);
15.        return null; //not a line yet
16.    }
17.
18.    @Override
19.    public byte[] encode(String message) {
20.        return (message + "\n").getBytes(); //uses utf8 by default
21.    }
22.
23.    private void pushByte(byte nextByte) {
24.        if (len >= bytes.length) {
25.            bytes = Arrays.copyOf(bytes, len * 2);
26.        }
27.
28.        bytes[len++] = nextByte;
29.    }
30.
31.    private String popString() {
32.        //notice that we explicitly requesting that the string will be decoded from UTF-8
33.        //this is not actually required as it is the default encoding in java.
34.        String result = new String(bytes, 0, len, StandardCharsets.UTF_8);
35.        len = 0;
36.        return result;
37.    }
38. }
```

EchoProtocol

We now implement a specific protocol on the server side. This server, when it receives a message, prints it on the screen (on the server side) together with the time it received and then return it back to the sender while repeating the last two chars a couple of times. That is, if a client send to the server the line "hello" it will be responded with the line "hello .. lo .. lo .."

The protocol also support the "bye" message which causes the server to close its connection to the client.

```
                                          20 lines ...
```

```java
public class EchoProtocol implements MessagingProtocol<String> {

    private boolean shouldTerminate = false;

    @Override
    public String process(String msg) {
        shouldTerminate = "bye".equals(msg);
        System.out.println("[" + LocalDateTime.now() + "]: " + msg);
        return createEcho(msg);
    }

    private String createEcho(String message) {
        String echoPart = message.substring(Math.max(message.length() - 2, 0), message.length());
        return message + " .. " + echoPart + " .. " + echoPart + " ..";
    }

    @Override
    public boolean shouldTerminate() {
        return shouldTerminate;
    }
}
```

A Client

Before we see how to put together the `ConnectionHandler` into a running server process, let us review the code of a very basic compatible TCP client for the protocol we have just described.

```
                                          28 lines ...
```

```java
public class EchoClient {

    public static void main(String[] args) throws IOException {

        if (args.length == 0) { //set default values
            args = new String[]{"localhost", "hello"};
        }

        if (args.length < 2) {
            System.out.println("you must supply two arguments: host, message");
            System.exit(1);
        }

        //BufferedReader and BufferedWriter automatically using UTF-8 encoding
        try (   Socket sock = new Socket(args[0], 7777);
                BufferedReader in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
                BufferedWriter out = new BufferedWriter(new OutputStreamWriter(sock.getOutputStream()))) {

            System.out.println("sending message to server");
            out.write(args[1]);
            out.newLine();
            out.flush();

            System.out.println("awaiting response");
            String line = in.readLine();
            System.out.println("message from server: " + line);
        }
    }
}
```

By now we created the protocol, message encoder/decoder, and client for our echo protocol. We know that our client will get treated in the server using its own connection handler, all that is left is to write the actual server - which is just an object that listen to new connection and assigned them to connection handlers.

Concurrency Models of TCP Servers

We now address the question of the concurrency model the TCP server should implement. A TCP server should strive to optimize the following quality criteria:

- **Scalability**: the capability to server a large number of concurrent clients.
- **Low accept latency**: do not make clients a long time before they are accepted.
- **Low reply latency**: send a reply to the client as fast as possible after it has been received.
- **High efficiency**: for a given number of concurrent connections and a given level of latency, use as little resources on the server host as possible (as measured by RAM, number of threads and CPU usage).

We can actually define an abstract server that allow its derivatives to implement different concurrency models Since we want the server to be generic we will supply it with suppliers of MessageEncoderDecoder and MessagingProtocol, as you can see in the following code.

```
                                                38 lines ...
1. public abstract class BaseServer {
2.
3.     private final int port;
4.     private final Supplier<MessagingProtocol> protocolFactory;
5.     private final Supplier<MessageEncoderDecoder> encdecFactory;
6.
7.     public BaseServer(
8.             int port,
9.             Supplier<MessagingProtocol> protocolFactory,
10.            Supplier<MessageEncoderDecoder> encdecFactory) {
11.
12.        this.port = port;
13.        this.protocolFactory = protocolFactory;
14.        this.encdecFactory = encdecFactory;
15.    }
16.
17.    public void serve() {
18.        try (ServerSocket serverSock = new ServerSocket(port)) {
19.
20.            while (!Thread.currentThread().isInterrupted()) {
21.
22.                Socket clientSock = serverSock.accept();
23.                ConnectionHandler handler = new ConnectionHandler(
24.                        clientSock,
25.                        encdecFactory.get(),
26.                        protocolFactory.get());
27.
28.                execute(handler);
29.            }
30.        } catch (IOException ex) {
31.            ex.printStackTrace();
32.        }
33.
34.        System.out.println("server closed!!!");
35.    }
36.
37.
38.    protected abstract void execute(ConnectionHandler handler);
39. }
```

First note that `Supplier` is an interface in java that has one non default function called get `get`. A factory is a supplier of objects. Our TCP server needs to create a new Protocol and EncoderDecoder for every connection it receives but since it is a generic server, it does not know what and how to create such objects. This problem is solved using factories, the server receives factories in its constructor that create those objects for it.

To obtain good quality, a TCP server will most often use multiple threads. We will now investigate three simple models of concurrency for servers, i.e., three implementations of preparing the ServerConcurrencyModel interface.

Server Model 1: Single Thread

The following server uses the same (main) thread for accepting a new client and for dealing its requests, by applying the run method of the passive ConnectionHandler object. Clearly, this implementation will have:

- No scalability: at any given moment in time, it can serve at most one client.
- Very high accept latency (a second client must wait until the first client disconnects to be served).

- Very low reply latency: all the resources of the server are concentrated on serving one client.
- Good efficiency: the server uses exactly the resources needed to serve one client at a time.

Such a solution may be considered if the time it takes to the server to process a full connection from one client is guaranteed to remain very small. For example, consider the case of a server that provides clients with the date and time value on the server machine. Such a server simply sends one string to the client then disconnects. In this case, it is perfectly fine to use this concurrency model (single thread). For any other type of connections, the model is not appropriate.

```
                                          15 lines ...
1. public class SingleThreadedServer extends BaseServer {
2.
3.     public SingleThreadedServer(
4.             int port,
5.             Supplier<MessagingProtocol> protocolFactory,
6.             Supplier<MessageEncoderDecoder> encoderDecoderFactory) {
7.
8.         super(port,protocolFactory,encoderDecoderFactory);
9.     }
10.
11.    @Override
12.    protected void execute(ConnectionHandler handler) {
13.        handler.run();
14.    }
15.
16. }
```

Server Model 2: Thread per Client

The following server assigns a new thread, for each connected client, by invoking the 'start' method over the runnable ConnectionHandler object.

This implementation will have:

- **Scalability**: the server can serve several concurrent clients, up to the point where there are too many threads running in the process. This happens when there are so many threads that all the RAM of the host is used (each thread allocates a stack and thus consumes RAM) and the scheduler of the host is overwhelmed. Practically, this happens when about 500 to 1000 threads become active within a single process. After this limit, the process itself does not defend itself – it keeps creating new threads for new incoming connections, and thus may become dangerous for the host.
- **Low accept latency**: the time from one accept to the next is approximately the time it takes to create a new thread - which is short compared to the delay between incoming client connections. Thus accept latency is good.
- **Reply latency**: the resources of the server are spread among all the concurrent connections. As long as a reasonable number of connections are active (no more than a few hundreds), and that the load requested by each connection is relatively low in CPU and RAM, then the server architecture will produce good reply latency.
- **Low efficiency**: the server creates a full thread for each connection, even if each connection may be mainly bound to Input/Output operations. That is, most of the time the ConnectionHandler thread will be blocked waiting for input data, but will still use the resources of the thread (RAM and Thread). We therefore expect efficiency to be low - and will find that the Reactor architecture will provide better efficiency.

```
                                          15 lines ...
1. public class ThreadPerClientServer extends BaseServer {
2.
3.     public ThreadPerClientServer(
4.             int port,
5.             Supplier<MessagingProtocol> protocolFactory,
6.             Supplier<MessageEncoderDecoder> encoderDecoderFactory) {
7.
8.         super(port, protocolFactory, encoderDecoderFactory);
9.     }
10.
11.    @Override
12.    protected void execute(ConnectionHandler handler) {
13.        new Thread(handler).start();
14.    }
15.
16. }
```

Server Model 3: Constant Number of Threads

The following server uses a constant number of threads, instead of a thread per client, by adding the runnable ConnectionHandler object to the task queue of a thread pool executor.

The key advantage of this design is that it avoids the danger described above of the server causing a complete host crash when too many clients connect at the same time to the server. The difference is that up to N concurrent client connections, the server behaves the same way as the "thread-per-connection" one; above this number, accept latency will grow. In other words, scalability is limited on purpose to the amount of concurrent connections we believe we can support.

```
                                                24 lines ...
1. public class FixedThreadPoolServer extends BaseServer {
2.
3.     private final ExecutorService pool;
4.
5.     public FixedThreadPoolServer(
6.             int numThreads,
7.             int port,
8.             Supplier<MessagingProtocol> protocolFactory,
9.             Supplier<MessageEncoderDecoder> encoderDecoderFactory) {
10.
11.         super(port, protocolFactory, encoderDecoderFactory);
12.         this.pool = Executors.newFixedThreadPool(numThreads);
13.     }
14.
15.     @Override
16.     public void serve() {
17.         super.serve();
18.         pool.shutdown();
19.     }
20.
21.     @Override
22.     protected void execute(ConnectionHandler handler) {
23.         pool.execute(handler);
24.     }
25. }
```

The Command Invocation Protocol

In the previous sections we learnt about a very simple echo protocol. In this section we are going to create a new generic protocol that will allow clients to execute remote commands on the server. Many complex applications can be described using this generic protocol and we will see an example of a news feed server that allow clients to publish and read news in multiple channels.

The Protocol

```
1. public interface Command<T> extends Serializable {
2.
3.     Serializable execute(T data);
4. }
```

A command is a generic interface with one method: `execute`. A command is sent by a client but executed by the server (on the server process). Depending on the server type, the server may pass a single argument to the command. This argument can hold different services that the command can interract with (as we will see in the later example). This process can be easily expressed by our `MessagingProtocol`

```
                                                18 lines ...
1. public class RemoteCommandInvocationProtocol<T> implements MessagingProtocol<Serializable> {
2.
3.     private T data;
4.
5.     public RemoteCommandInvocationProtocol(T data) {
6.         this.data = data;
7.     }
8.
9.     @Override
```

```
    public Serializable process(Serializable msg) {
        return ((Command) msg).execute(data);
    }

    @Override
    public boolean shouldTerminate() {
        return false;
    }

}
```

Java-Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

A serialized object (i.e., a `byte[]`) can be deserialized back into a copy of the original object that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes ObjectInputStream and ObjectOutputStream are high-level streams that contain the methods for serializing and deserializing any object, they are able to deserialize any Serializable object. A class is Serializable if it:

- Implements the Serializable interface or it super class is Serializable
- Its first non-serializable super class has a no-args constructor
- All its non-transient fields must be serializable

For the purpose of this lecture this information is sufficient, you can read more about the serialization mechanism in the following links: serializable javadoc , basic serialization tutorial and java object serialization .

With the above knowledge we can actually design a message encoder decoder that can handle arbitrary serializable objects and especially our commands:

```
                                                85 lines ...
public class ObjectEncoderDecoder<> implements MessageEncoderDecoder<Serializable> {

    private final byte[] lengthBytes = new byte[4];
    private int lengthBytesIndex = 0;
    private byte[] objectBytes = null;
    private int objectBytesIndex = 0;

    @Override
    public Serializable decodeNextByte(byte nextByte) {
        if (objectBytes == null) { //indicates that we are still reading the length
            lengthBytes[lengthBytesIndex++] = nextByte;
            if (lengthBytesIndex == lengthBytes.length) { //we read 4 bytes and therefore can take the length
                int len = bytesToInt(lengthBytes);
                objectBytes = new byte[len];
                objectBytesIndex = 0;
                lengthBytesIndex = 0;
            }
        } else {
            objectBytes[objectBytesIndex++] = nextByte;
            if (objectBytesIndex == objectBytes.length) {
                Serializable result = deserializeObject();
                objectBytes = null;
                return result;
            }
        }

        return null;
    }

    private static void intToBytes(int i, byte[] b) {
        b[0] = (byte) (i >> 24);
```

```
        b[1] = (byte) (i >> 16);
        b[2] = (byte) (i >> 8);
        b[3] = (byte) i;
    }

    private static int bytesToInt(byte[] b) {
        //this is the reverse of intToBytes,
        //note that for every byte, when casting it to int,
        //it may include some changes to the sign bit so we remove those by anding with 0xff

        return ((b[0] & 0xff) << 24)
                | ((b[1] & 0xff) << 16)
                | ((b[2] & 0xff) << 8)
                | (b[3] & 0xff);
    }

    @Override
    public byte[] encode(Serializable message) {
        return serializeObject(message);
    }

    private Serializable deserializeObject() {
        try {
            ObjectInput in = new ObjectInputStream(new ByteArrayInputStream(objectBytes));
            return (Serializable) in.readObject();
        } catch (Exception ex) {
            throw new IllegalArgumentException("cannot desrialize object", ex);
        }

    }

    private byte[] serializeObject(Serializable message) {
        try {
            ByteArrayOutputStream bytes = new ByteArrayOutputStream();

            //placeholder for the object size
            for (int i = 0; i < 4; i++) {
                bytes.write(0);
            }

            ObjectOutput out = new ObjectOutputStream(bytes);
            out.writeObject(message);
            out.flush();
            byte[] result = bytes.toByteArray();

            //now write the object size
            intToBytes(result.length - 4, result);
            return result;

        } catch (Exception ex) {
            throw new IllegalArgumentException("cannot serialize object", ex);
        }
    }

}
```

The ObjectEncoderDecoder is the first binary encoder decoder that you encountered in this course. It is actually much more simple than it looks like, the encoding of a message that contains the object `O` which can be serialized to a byte array `b1,b2,b3,...,bN` will be `N,b1,b2,b3,...,bn` (i.e., the message will start with the number of bytes that need to be read in order to deserialize an object). The number of bytes `N` is sent using a binary representation - note `byteToInt` and `intToByte`. Finally, when an object received, the `decodeNextByte` method first checks if it belongs to `N` or that we already started to read `b1,b2,...,bN` and fill the correct byte arrays.

We can now create a generic client for our generic protocol

```
39 lines ...
```

```
public class RCIClient implements Closeable{

    private final ObjectEncoderDecoder encdec;
    private final Socket sock;
    private final BufferedInputStream in;
    private final BufferedOutputStream out;

    public RCIClient(String host, int port) throws IOException {
        sock = new Socket(host, port);
        encdec = new ObjectEncoderDecoder();
        in = new BufferedInputStream(sock.getInputStream());
        out = new BufferedOutputStream(sock.getOutputStream());
    }


    public void send(Command<?> cmd) throws IOException {
        out.write(encdec.encode(cmd));
        out.flush();
    }

    public Serializable receive() throws IOException {
        int read;
        while ((read = in.read()) >= 0) {
            Serializable msg = encdec.decodeNextByte((byte) read);
            if (msg != null) {
                return msg;
            }
        }

        throw new IOException("disconnected before complete reading message");
    }

    @Override
    public void close() throws IOException {
        out.close();
        in.close();
        sock.close();
    }

}
```

These 4 classes (the Command, RemoteCommandInvocationProtocol, ObjectEncoderDecoder and RCIClient) can be serve as the basis for many advanced servers as we will see next.

NewsFeed Server

Lets utilize our generic command invocation protocol to create a NewsFeed server. This server will allow clients to execute two commands:

- Publish news to a category by its name
- Fetch all the news which were published to a specific category

The main object that is manipulated by the server is the NewsFeed

```
public interface NewsFeed {

    void clear();

    List<String> fetch(String category);

    void publish(String category, String news);

}
```

The client commands receives the `NewsFeed` and manipulate it

```
                                                13 lines ...
public class FetchNewsCommand implements Command<NewsFeed> {

    private String category;

    public FetchNewsCommand(String category) {
        this.category= category;
    }

    @Override
    public Serializable execute(NewsFeed feed) {
        return feed.fetch(category);
    }

}
```

```
                                                16 lines ...
public class PublishNewsCommand implements Command<NewsFeed> {

    private String category;
    private String news;

    public PublishNewsCommand(String category, String news) {
        this.category= category;
        this.news = news;
    }

    @Override
    public Serializable execute(NewsFeed feed) {
        feed.publish(category, news);
        return "OK";
    }

}
```

Note that the client works with the interface of news feed while the server will have the actual implementation. Since the news feed can be manipulated by different connection handlers in the server on the same time (as they will respond to concurrent client requests) it must be implemented as a thread safe object.

```
                                                24 lines ...
public class NewsFeedImpl implements NewsFeed {

    private ConcurrentHashMap<String, ConcurrentLinkedQueue<String>> newsPerCategory = new ConcurrentHashMap<>
();

    @Override
    public List<String> fetch(String category) {
        ConcurrentLinkedQueue<String> queue = newsPerCategory.get(category);
        if (queue == null) {
            return new ArrayList<>(0); //empty
        } else {
            return new ArrayList<>(queue); //copy of the queue, arraylist is serializable
        }
    }

    @Override
    public void publish(String category, String news) {
        ConcurrentLinkedQueue<String> queue = newsPerCategory.computeIfAbsent(category, (k) -
> new ConcurrentLinkedQueue<>());
        queue.add(news);
    }

    @Override
    public void clear() {
```

```
        newsPerCategory.clear();
    }
}
```

This is actually all that is needed in order to implement our protocol, we can now start the server as follows:

```
                                          11 lines ...
public class NewsFeedServerMain {

    public static void main(String[] args) {
        NewsFeed feed = new NewsFeedImpl(); //one shared object

        new ThreadPerClientServer(
                7777, //port
                () -> new RemoteCommandInvocationProtocol<>(feed), //protocol factory
                () -> new ObjectEncoderDecoder<>()                 //message encoder decoder factory
        ).serve();
    }
}
```

And we can use the following code to test our server

```
                                          51 lines ...
public class NewsFeedClientMain {

    public static void main(String[] args) throws Exception {
        if (args.length == 0) {
            args = new String[]{"localhost"};
        }

        System.out.println("running clients");

        runFirstClient(args[0]);
        runSecondClient(args[0]);
        runThirdClient(args[0]);
    }

    private static void runFirstClient(String host) throws Exception {
        try (RCIClient c = new RCIClient(host, 7777)) {
            c.send(new PublishNewsCommand(
                    "jobs",
                    "System Programmer, knowledge in C++, Java and Python required. call 0x134693F"));

            c.receive(); //ok

            c.send(new PublishNewsCommand(
                    "headlines",
                    "new SPL assignment is out soon!!"));

            c.receive(); //ok

            c.send(new PublishNewsCommand(
                    "headlines",
                    "THE CAKE IS A LIE!"));

            c.receive(); //ok

        }

    }

    private static void runSecondClient(String host) throws Exception {
        try (RCIClient c = new RCIClient(host, 7777)) {
            c.send(new FetchNewsCommand("jobs"));
```

```
            System.out.println("second client received: " + c.receive());
        }
    }

    private static void runThirdClient(String host) throws Exception {
        try (RCIClient c = new RCIClient(host, 7777)) {
            c.send(new FetchNewsCommand("headlines"));
            System.out.println("third client received: " + c.receive());
        }
    }
}
```

which will print:

```
second client received: [System Programmer, knowledge in C++, Java and Python required. call 0x134693F]
third client received: [new SPL assignment is out soon!!, THE CAKE IS A LIE!]
```