CO633 Computer Networks 2019-20 : Programming Assignments #1 Sender and #2 Receiver

Introduction

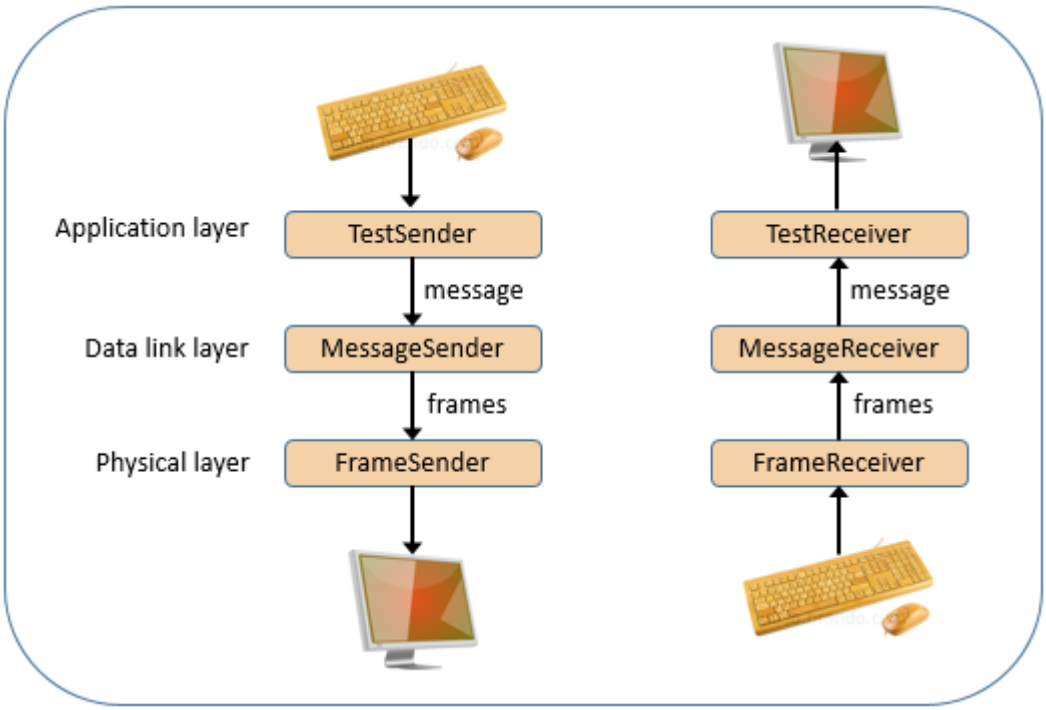
Quick links in this document

Appendices and other external links

Overview

The programming assignments involve implementing parts of a simple data link protocol using Java. On completion you'll have gained a more concrete understanding of several important design issues including framing, error detection, message segmentation and reassembly. The coursework also serves to consolidate problem-solving and software development skills.

The data link classes you're working with are components of a simple protocol stack:



Layer	Description and associated classes
Physical	• The physical layer is implemented by classes FrameSender and FrameReceiver. • These provide a primitive frame stream service with no error detection or flow control. • The key methods are sendFrame and receiveFrame. • The stream can carry a series of zero or more frames. • Frames are implemented as Java Strings for convenience.

	• The frame format is defined by the data link layer which also constrains the length. • Each send/receiveFrame call communicates one frame. • To facilitate testing, sendFrame outputs frames to the screen and receiveFrame inputs frames from the keyboard. • The keyboard and screen are accessed via a helper class TerminalStream (see below).
Data Link	• The data link layer is implemented by classes MessageSender and MessageReceiver. • These provide a more flexible message stream service with error detection (flow control and error correction/recovery are **not** supported by this protocol and should **not** be added). • The key methods are sendMessage and receiveMessage. • The stream can carry a series of zero or more messages. • Messages are implemented as Java Strings for convenience. • A message can be any length (including empty/zero) and can contain any sequence of characters supported by the String class. • Each send/receiveMessage call communicates one message. • sendMessage encodes a message as a sequence of one or more frames based on the frame format specification. • Several frames may be needed for a long message because a single frame must not exceed the *maximum transfer unit* (**MTU**). • receiveMessage reverses the process by decoding a sequence of frames and reassembling the message segments into a single string. • send/receiveMessage should check for errors and throw a ProtocolException if an error is detected.
Application	• The application layer is implemented by the test harness classes TestSender and TestReceiver. • In both classes, the runTest methods are designed for use with BlueJ and the main methods are designed for command line execution. • In TestSender, these methods input messages from the keyboard and call sendMessage to transmit them. • In TestReceiver, they call receiveMessage to obtain messages and output them to the screen. • The keyboard and screen are accessed via TerminalStream, as for the physical layer. • The class documentation for each test harness contains instructions on its use. • Illustrations of the test harnesses in operation are included in the testing appendix.
Other classes	• ProtocolException is used to signal protocol and other network-related errors. • The physical and data link layer class documentation indicates when and how this exception should be used. • TerminalStream is a helper class that handles access to standard input (normally the keyboard) and standard output (normally the screen/terminal window). • TerminalStream also supports diagnostic features to assist with testing and debugging. Again, further information can be found in the class documentation.

What you need to do

Download the ZIP archive containing source files for the above classes. A BlueJ package file is included for the convenience of those using BlueJ but you do **not** need to use BlueJ for this assignment.

The following classes are already complete. **Do not alter these classes!**

- TestSender, TestReceiver
- FrameSender, FrameReceiver
- ProtocolException
- TerminalStream

The following classes are incomplete. The interfaces are defined but most of the code for sendMessage and receiveMessage is missing. **Do not alter the interfaces of these classes!**

- MessageSender, MessageReceiver

The programming coursework is divided into two parts with different deadlines. #1 Sender involves completing MessageSender and #2 Receiver is to complete MessageReceiver. Although both are based on the same protocol, the classes are designed so the sender and receiver can be developed and tested independently.

To score a high mark you will need to:

- Study the frame format specification, class documentation and marking criteria (see below) carefully to understand the requirements and how they'll be assessed. If you have difficulty interpreting the formal language of the specification, it may help to look at the examples provided at the end of the frame format spec and testing appendices.

- Design and code solutions for sendMessage and receiveMessage that function correctly in all cases, including the proper detection and handling of errors. All the information about the protocol necessary for this purpose is provided in the specification and accompanying documentation. However, developing a working solution will involve reasoning and problem-solving skills plus a good understanding of network concepts. Advanced knowledge of Java is **not** required. If your programming skills are rusty, the advice on tackling the implementation appendix may help you get started.

- Test your implementations thoroughly. This is an important part of the exercise. To help you get started, sample test data and examples of valid operation can be found in the frame format spec and testing appendices. Check your solutions give the same results. Note the examples provided are **not** comprehensive tests. Additional checks may be needed to verify a full solution rigorously. This is especially true for the receiver due to its greater complexity (decoding is inherently harder than encoding) and potential data corruption during transmission (which could affect any characters).

Notes on implementation:

- **Java 11** will be used when the assignments are marked. Please avoid new language/API features only supported by Java 12 or later.
- The Java version incorporated into BlueJ (Java 8) is suitable.
- Standard Java API classes (e.g. from java.util) can be imported and used within MessageSender and MessageReceiver, if you wish.
- Do **not** alter the interface of any class or change the constructors.
- Do **not** put the supplied classes into a package or create any additional classes or sub-classes of your own.

How your work will be marked

Marks will be based entirely on functionality. We will **not** inspect your Java source files.

Your implementations of send/receiveMessage will be marked with the help of a partly automatic test harness that replaces the other classes. It will invoke send/receiveMessage with a variety of messages/frames, varying the MTU and simulating borderline and error conditions. Marks will be awarded according to how well your code performs in these tests. An implementation that functions correctly for all tests will score 100%. We will

not be releasing the test data used for marking because, as explained earlier, working out how to verify your implementation thoroughly is part of the exercise.

Beware of critical bugs!

An implementation that yields incorrect results for most tests will score a low mark, regardless of how many lines of code the Java file contains! It's therefore better to submit a partial solution handling a subset of test cases correctly (e.g. just short messages) than a more comprehensive solution containing a critical bug causing it to fail every time.

Here are common causes of critical bugs observed in previous years:

- Making an improvement/bug-fix in a hurry near the deadline without thorough retesting. Sometimes a tiny tweak to the code can lead to a catastrophic fault.
- Trying to implement too many features at the same time. The resulting code is likely to contain multiple bugs which can combine in strange ways that are hard to isolate. It's better to build the implementation incrementally, testing each new part as it's added.
- Not reading the frame format spec carefully and/or not checking your output matches the examples provided. If they don't match exactly there may be a serious problem.
- Confusing the end of the input stream with the end of a single message, for example receiveMessage keeps grabbing frames until the end of the input stream is reached rather than stopping when the last frame of the current message has been received. Compare your output with examples provided that involve several messages.

Marking criteria

The marks for each assignment are divided into several categories, each corresponding to a group of related tests. The categories and approximate breakdown of marks are listed below. More marks are allocated for the receiver because it's more difficult to implement.

Assignment	Test category	Approx marks
#1 Sender	Short message encoding (single frame only)	15
	Longer message encoding (segmentation and MTU compliance)	15
	Robustness (errors and borderline conditions)	10
	Subtotal for sender assignment	*40*
#2 Receiver	Short message decoding (single frame only)	15
	Longer message decoding (segmentation and MTU compliance)	15
	Robustness (errors and borderline conditions)	30
	Subtotal for receiver assignment	*60*

The tests check the completeness, correctness, robustness and efficiency of your implementation:

- Completeness relates to the proportion of protocol features implemented.
  It's possible to achieve a pass mark without implementing the full protocol.
- Correctness means how accurately those features that are implemented comply with the specification.
  It's better to omit a feature entirely than to implement it incorrectly in a way that makes the whole system fail.
- Robustness refers to the detection and handling of errors and borderline cases.
  This criterion represents a large part of the marks for the receiver because there are many potential cases to consider.

- Efficiency in this context is concerned solely with how efficiently the protocol is used and not how fast it executes.
  For example, does sendMessage make the best possible use of the MTU?

Deadlines and submission

Your work must be submitted electronically by the method described here. Folders have been created on raptor for each student with separate subfolders for the sender and receiver. Please check you can access your submission folders well before the deadline so any technical problems can be fixed in good time.

Make sure the correct file is placed in each folder. Only the Java files named below should be submitted. The .class files are not required. Do not create any new subfolders or change the names of the Java files.

In the folders column below *login* refers to your public PC username. Linux users can find these folders under /proj.

Assignment	Estimated work	File to be submitted	Folder where file to be stored	Deadline	Weight (towards final module mark)
#1 Sender	12 hours	MessageSender.java	\\raptor\files\proj\co633c\sender*login*	23:55 Mon Week 9 (**18 Nov 2019**) - *TBC*	8%
#2 Receiver	18 hours	MessageReceiver.java	\\raptor\files\proj\co633c\receiver*login*	23:55 Fri Week 11 (**6 Dec 2019**) - *TBC*	12%

If you're unable to upload the file on the evening of the deadline then it can be emailed directly to Peter Kenny. However, please only use email as a last resort.

Help and advice

See the appendices for further advice, including suggestions for how each assignment can be broken down into smaller steps. If you get stuck or something needs clarification please check the CO633 moodle forum but do **not** post questions about the programming assessments directly to the forum, instead email Peter Kenny.

Plagiarism

You are reminded of the University's rules on plagiarism and duplication of material. **The rules for these assignments are stricter then those you may have encountered previously.** Designing, implementing and testing the solution are all part of the exercise. This includes working out what kinds of error could arise. Hence please do **not** collaborate with anyone else on the design, implementation or testing.

Your work will be checked for originality using a plagiarism detection tool. We also reserve the right to use submitted code to test and refine our plagiarism detection methods during the course and in the future.