Programming Practice (PRP), Coursework Exercise 4 (40%, 40 marks)

Please read the document marked 'Continuous Assessment Guidelines' carefully, before attempting any piece of coursework.

This assignment counts for 40% of your mark for PRP continuous assessment, and is the fourth of four. If you have not yet completed the first three assignments for PRP, your opportunity to receive a mark for these assignments has passed. However, you are still advised to look over these assignments, and their respective model answers, before proceeding with this one.

The release week for this assignment starts 28th November, at 23:55, and ends 5th December, at 23:55. All submissions must occur before the end of the release week.

This assignment is split into two problems. Problem A is a modelling task. Problem B uses this model to solve a potentially challenging problem. Students who just complete Problem A will receive a maximum grade of 70%. Students who complete both Problems A and B can receive up to 100%. Problem B is further subdivided into different grade boundaries. Both problems can be solved using the same set of classes.

It is recommended that you spend the majority of your week working on Problem A.

If you have any questions about the structure of this assessment, please email martin.chapman@kcl.ac.uk.

1 Problem A (70% maximum)

1.1 Problem

We want to build a simple model of a wireless network. To do this, we need to model the components involved and implement a simple protocol, the use of which will allow two components to connect. This type of protocol is known as a *handshake*. Then, we want setup our network, before finally running it.

1.1.1 The components involved

Wireless communication involves the following components:

1. Wireless communication occurs between *network devices*. All `NetworkDevice`s have an `address`, which identifies the device.
2. A `Client` (e.g. a laptop or a smartphone) and an `AccessPoint` (e.g. a home router) are special types of `NetworkDevice`. A client connects to an access point in order to use a service (e.g. to gain access to the Internet) provided by the access point. A `Client` keeps a record of which `AccessPoint` it is currently connected to, if any. An `AccessPoint` keeps a record of the `Client`s that are `authorised` to use its services.
3. Data is exchanged between network devices in the form of digital *packets*. We can imagine a packet as a (postal) letter, where each `Packet` has a `destinationAddress` (to which device the letter should be sent) and a `sourceAddress` (which device sent the letter). For an example of this, see Figure 1. We will ignore the data content in a packet for the purposes of this assignment.
4. Network devices do not exchange packets directly. Instead, they place packets into a channel. Multiple devices can use the same channel, and all devices read from that channel in order to locate packets addressed to them. `Channel`s each have a `number`, and contain `traffic`, which is a record of the `Packet`s currently in the channel.
5. Each `NetworkDevice` has the ability to `join` a `Channel`, but can only exist in one `Channel` at a time.
6. When an `AccessPoint` is `add`ed to a `Network`, it `join`s a new `Channel` upon which to communicate. It should be possible to obtain a list of the `Channel`s that are in use in a `Network`. However, for security, it should *not* be possible to access the `Channel` in

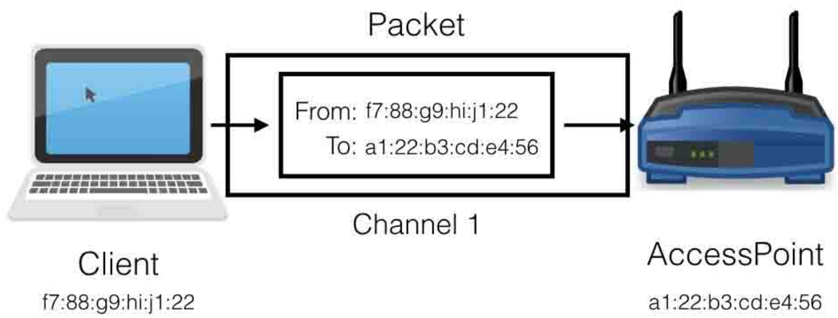

Figure 1: Communication between a `Client` and an `AccessPoint` (both with an address) takes place using a packet, which is a (postal) letter containing a 'from' and 'to' label, passed into a `Channel` that is shared by both devices.

use by a `NetworkDevice` directly. Instead, the `Network` should keep a private record of which `Channel`s are in use by different `NetworkDevices`.

1.1.2 Handshake Authentication

Once a network has been setup with at least one access point, and that access point has a channel, a `Client` can `join` the same `Channel` as an access point, in order to address packets to and receive packets from the access point (for the purpose of using a service, as previously described). Remember that an arbitrary number of access points and an arbitrary number of clients can share the same channel.

However, as the network has the only (accessible) record of which channel is in use by an access point (and, conceivably, there could be a vast number of channels), a client cannot join an access point's channel directly. Instead, a `Client` must undertake a process known as a `handshake`, facilitated by the `Network`, before being permanently joined to an `AccessPoint`'s channel.

A `handshake` simply means that a set of packets are sent by each device in a certain order in order to verify that a `Client` is authorised to use an `AccessPoint`. Therefore, for a handshake to be successful, both an `AccessPoint` and a `Client` must have the same `key` (e.g. a WiFi password). A `handshake` proceeds between a `Client` and an `AccessPoint` in a `Network` as follows:

1. The `Network` `join`s the `Client` to the `AccessPoint`'s `Channel`. Note that the client's connection to this channel is (initially) *temporary*, and *only* for the purpose of the handshake. The connection will be *revoked* if the handshake fails (indeed, the connection can only be considered real once the `AccessPoint` has listed the `Client` as `authorised`).

2. The `Client` `add`s a special type of `Packet` called a `HandshakePacket` to the `Channel` now shared with the `AccessPoint`, where the `destinationAddress` of this packet is the address of the `AccessPoint`, and naturally the `sourceAddress` is the address of the `Client`. A `HandshakePacket` contains the `key`.

3. The `AccessPoint` `get`s the `traffic` in its `Channel`, and if it finds a `HandshakePacket` it checks whether the `destinationAddress` on this packet matches its own address. We will assume that all devices, when they `get` the traffic from a `Channel`, only *inspect* each `Packet`, as opposed to removing it, so if a `HandshakePacket` is not addressed to the `AccessPoint`, reading it (and ignoring it) has no consequence. Assuming a `HandshakePacket` addressed to the `AccessPoint` is found, the `AccessPoint` then extracts the `key` from the packet, and checks whether this matches its own `key`.

4. If the `AccessPoint` extracts a `key` from a `HandshakePacket` and finds it to match its own, it lists the `Client` from whom the `HandshakePacket` originated as an `authorised` `Client`, and to confirm it has done this, it places *another* `HandshakePacket` into its `Channel`, where the `destinationAddress` is that of the now authorised `Client` (and the `sourceAddress` is its own address).

5. The `Client` now `get`s the `traffic` from the `Channel`, in order to locate the second `HandshakePacket` sent from the `AccessPoint`. If it finds it, the `Client` checks that the `key` from the packet still matches its own, and if it does, it knows it has been authorised to connect, and thus lists the `AccessPoint` as the device it is currently connected to. Only two handshake packets are transferred in total during the exchange.

6. If the `Network` observes that the `Client` has been authorised to connect (i.e. the handshake was successful), the `Client` remains connected to the `AccessPoint`'s `Channel`, and the `Network` records that the `Client` is using that `Channel`. If the `handshake` is not successful, then the `Client` is `disconnected` from the `AccessPoint`'s `Channel`. As a further layer of security, an `AccessPoint` should *only* respond to `Packet`s from `authorised` `Client`s.

1.1.3 Network Activity

Once a network is setup, and at least two devices have engaged in a handshake, the network `starts`. Once a network has started, all network devices on the `Network` (i.e. `AccessPoint`s and those `Client`s that have engaged in a successful `handshake`) `communicate` within the `Network` every 100 milliseconds. We refer to this as *burst* of *normal* `networkActivity` (as opposed to `handshake` activity). During a burst of `networkActivity`, all `NetworkDevices` are asked to `communicate`.

A `Client` `communicate`s by addressing a `Packet` to its connected `AccessPoint` and placing it in its `Channel`, and an `AccessPoint` `communicate`s by `get`ting the `traffic` in its `Channel` in order to check whether any packets are addressed to it. If the `AccessPoint` finds a packet addressed to it, it responds with an additional packet addressed to the sender. Remember that the `AccessPoint` should only respond to packets from `authorised Clients`. Recall that for simplicity, we assume that packets do not have any data content (i.e. they are blank). Before a burst of `networkActivity` occurs, all channels are `cleared`.

If we were to *log* they key activities involved in both setting up and running a network, this log may look something like `log1.txt`, available on KEATS.

1.2 Requirements

Implement and setup a wireless network, according the task given above, in order to produce an activity log similar to the one shown in `log1.txt`. You can assume that only a single `Client` and a single `AccessPoint` communicate within a `Network`, however your code should be able to accommodate multiple devices, and you can add these devices to create additional traffic if you wish. No matter how many devices you choose to implement **at least one of your access points should have the `address` "68:a4:69:dd:80:6e" and the `key` "password"**.

Tip: The code `Thread.sleep(N)`, *pauses* the execution of your code, where `N` is the number of milliseconds to pause for. This command pauses the *main* thread of your program. Multi-threaded code is not required.

Tip: Skip Problem B, read the Mark Scheme and the Tips, and then implement Problem A before reading Problem B.

2 Problem B (70%+)

2.1 Problem:

For up to 80%:

A hacker, `Mr Robot`, wishes to gain access to an access point, without knowing the key to that access point.

To do this, Mr Robot knows that he can intercept a handshake packet exchanged by the target access point and a client, and extract the key. For this purpose `Mr Robot` has a `targetClient` and a `targetAccessPoint`. To be certain that he has an accepted key, Mr Robot needs to *observe both* of the handshake packets exchanged by two devices (a client and an access point). He can then reliably extract the key contained in either packet.

However, because the evidence of the initial handshake is cleared before the first activity burst (Section 1.1.3), Mr Robot needs to wait for devices to *reconnect*. This is known as `listening` for a handshake. For a reconnect to occur, we will assume that, 10% of the time, before communicating, a `Client disconnects` from an `AccessPoint` (perhaps due to going out of range of the access point). During a burst of network communication therefore, before a `Client` is asked to `communicate`, the `Network` checks whether a `Client` is `connected`. If it is not, then the `Network` automatically performs another `handshake` between the `Client` and the *last* `AccessPoint` it was connected to. In order to understand which `AccessPoint` a `Client` was last connected to, every time a `handshake` occurs, the `Network` records, as a form of `history`, which `AccessPoint` the `Client` is currently connecting to.

For up to 90%:

To exploit this repeated exchange of handshake packets, after each burst of network activity `Mr Robot gets` *all* the traffic in *all* the `Channels` of a `Network`, and performs two out of three possible checks:

1. Is the packet a `HandshakePacket` **and**
2. Has the `HandshakePacket` been sent by the `targetAccessPoint` to the `targetClient` **or**
3. Has the `HandshakePacket` been sent by the `targetClient` to the `targetAccessPoint`.

If, for example, a handshake packet has been located and it has been sent from the `targetAccessPoint` to the `targetClient`, then Mr Robot knows that half a handshake has been captured. Two handshake packets need to be captured before the key can be reliably extracted from a complete handshake. Because a client will not always disconnect prior to each burst of communication, Mr Robot may have to examine the packets exchanged in multiple bursts of network activity before discovering two handshake packets.

Once `Mr Robot` has the `key`, he should call the `Network` in order to handshake his own, new `Client` (to which he attributes this `key`) with the `AccessPoint`, in order to start using the services offered without permission.

An example log of this activity is shown on KEATS, in `log2.txt`.

Tip: Skip to the requirements and implement everything you have read so far, before reading the rest of Problem B.

For up to 100%:

So far we have assumed that keys are sent in handshake packets without some kind of protection, for example by *scrambling* them. One way to scramble a key is to associate every key with a unique number, and then only send this unique number during communication, rather than the original key. This process is known as *hashing*. In order to make a network more secure, a `HashFunction` should be implemented. In order to setup a hash function, it should be possible to provide a series of keys in `plaintext` (i.e. as unencrypted Strings), associate each key with a unique number, and then store this information.

With this information stored, for security, it should *only* be possible to `hash` a key (translate it from plaintext to its unique integer using the association established when the function was setup) *not* the other way around. Evidently this hash function is a simplification because it can only hash the keys it keeps associations for[1].

We will assume that the only keys used by the devices in the network are specified in the file `keys.txt`, available on KEATS. Therefore, when a network is `setup`, all these keys should be `add`ed to the `HashFunction`. From then onwards, whenever a `key` is used by a `NetworkDevice` in communication, it should be passed to the `HashFunction` and `hash`ed before being placed into a `HandshakePacket`. Moreover, during a handshake (Section 1.1.2), in order to check whether the `key` in a `HandshakePacket` matches the key held (Points 3 and 5, Section 1.1.2), both an `AccessPoint` and a `Client` must now `hash` their

[1]You are welcome to implement your own hashing function that works for arbitrary Strings, should you wish to

own keys, before checking whether they match the hashed key in the `HandshakePacket`.

Hashing keys makes the Mr Robot's job more difficult because he can now only see the encrypted version of an access point's key. However, we assume that the `HashFunction` is also known to his (imagine it is a standard algorithm), and, moreover, we imagine that he has stolen `keys.txt` and can thus narrow down which key is being used by the access point. What Mr Robot must therefore do is to use the `HashFunction` to `hash` each key from `keys.txt` and check whether each key matches the hashed `key` from both of the `HandshakePackets`[2]. When the key does match, he knows it is correct, and can connect, unauthorised, as before, to the `AccessPoint` using the correct key from `keys.txt`.

2.2 Requirements

Alter the classes you implemented for Problem A, in order to accommodate the reconnection behaviour, and the behaviour of a hacker.

3 Mark Scheme

Marks for this assignment will be awarded as follows:

For 0 - 16 marks (0% - 40%):

1. Correctly decomposing the problem into a set of classes relevant to the problem.
2. Using these classes to both store and provide access to all information relevant to the problem. This information should be of an appropriate type.
3. Correctly encapsulating all information.
4. Taking the appropriate steps to ensure that all information that is required by each class is always present (e.g. every `NetworkDevice` has an address).

[2] Conceivably Mr Robot could *bruteforce* the `AccessPoint` directly, but bruteforcing captured handshakes is often preferred, so that the task can be done offline, and out of range of the access point.

For 16 - 20 marks (40% - 50%): All of the above, and:

1. Maximising efficiency through abstraction (i.e. collecting the common features of all `NetworkDevices`).
2. Setting up a `Network` and its associated `NetworkDevices` (without handshaking).

For 20 - 24 marks (50% - 60%): All of the above, and implementing each stage of the handshake:

1. Handshake Stage 1 (Section 1.1.2, Point 2)
2. Handshake Stage 2 (Section 1.1.2, Points 3 and 4)
3. Handshake Stage 3 (Section 1.1.2, Point 5)
4. Handling the result of a handshake (Section 1.1.2, Point 6)

For 24 - 28 marks (60% - 70%) All of the above, and creating network activity:

1. Implementing how an `AccessPoint` and a `Client` communicate. Including ensuring that an `AccessPoint` only responds to `Packets` from `authorised Clients`.
2. Implementing a burst of network activity, in which all `NetworkDevices` communicate.
3. Running a network, and having the bursts of activity at appropriate intervals.
4. Clearing all channels when appropriate.

Problem B marks begin here.

For 28 - 36 marks (70% - 80%) All of the above, and:

1. Setting up a `Hacker` with appropriate fields.
2. Modifying a `Client` so that, before it communicates, there is a 10% chance of it disconnecting rather than communicating.

3. Modifying a `Network` so that a record is kept of `Client` and `AccessPoint` history.

4. Modifying a `Network` so that any disconnected device are reconnected by performing another `handshake` between them

For 28 - 36 marks (80% - 90%) All of the above, and:

1. Implementing a means by which a `Hacker` can recognise when two handshake packets, belonging to its `targetAccessPoint` and its `targetClient` are in a `Channel`, and thus from where to extract a key.

2. Using an extracted `key` to perform a `handshake` in a `Network` between a new `Client`, owned by the `Hacker`, and the `targetAccessPoint`.

3. Modifying the running of a network so it does not just include `networkActivity` but also `hackerActivity`.

For 36 - 40 marks (90% - 100%) All of the above, and:

1. Implementing `HashFunction`, including automatically reading all `keys` from `keys.txt`.

2. Altering all uses of `keys` by the `Client` and the `AccessPoint`, so that they are `hash`ed prior to being added to a `Packet`.

3. Implementing the means for a `Hacker` to `crack` a `HandshakePacket` by hashing all possible `keys` from `keys.txt`, and comparing them to the hashed `key` in the `HandshakePacket`.

In addition, note that:

1. Code that does not compile will receive a maximum mark of 40%. As with all previous coursework, you must test whether your code compiles on one of the lab computers *outside* of any IDE (i.e. by compiling it from the command line), even if you intend to demonstrate your code to your examiner on your own laptop.

   (a) Do *not* adopt a non-standard packet structure for your work; it must be possible for your code to compile simply by compiling the class in which your main method resides.

(b) Do *not* submit any classes that do not compile directly, even if they are not involved in the running of your program (**Tip:** Running `java *.java` in your source directory checks whether all your classes compile.)

2. The mark scheme for 36 marks and above is not exhaustive. It is at the discretion of us as examiners to reward those students who demonstrate an understanding of reusability, encapsulation, abstraction and decomposition, and who produce their solutions in the most efficient manner.

3. You are not required to implement your own exceptions in this assignment, but you may have to handle exceptions.

4. **As in previous assignments, your final grade is based upon both the quality of your code, and your ability to describe your code to your examiner.**

4 Tips

Tips and useful information for this assignment are as follows:

1. You should spend the majority of your week working on Problem A. Only attempt Problem B once you are certain you have a sufficient solution to Problem A.

2. It is essential that you discern which elements of the problem statement require implementation, and which are provided for clarity. Only sentences containing `verbatim` font require direct implementation. Sentences not containing `verbatim` font do not require any direct implementation, but support your understanding of the problem.

3. It is recommended that you at least maintain a separate version of your code for Problem A and Problem B. In reality, you should work with many more self-contained versions, using the method discussed in lectures.

4. Start off with the easy parts of the problem first. Build what you can and what you are familiar with, and then focus on how these components can be combined and developed in order to solve the harder parts of the problem.

5. The focus of this assignment is not to replicate the functionality of network devices in detail nor entirely accurately. Instead, focus on how the problem enables you to exhibit examples of coding practice you have learnt in the course so far. Consulting the lecture exercises is the *best* way to gain insight into how this problem can be solved.

Once you have completed this assignment, you must place all the code you have produced into a folder, name this folder 'Exercise4', compress it (to one of a .zip, .rar or .tar.gz file, no other formats will be accepted for grading) and submit it to KEATS. Please note that you should only submit plain text files with a .java extension for assessment (so no proprietary formats such as PDF or Rich Text).

In addition, please note that Martin Chapman's office hours this week, originally scheduled for 30th November, 10am - 12pm, have been moved to 1st December, at the same time.