COMP225 Assignment 2: The Crowds Protocol

ver 1.01

Mark Dras

May 11, 2019

1 Introduction

The ability to engage in anonymous communication over networks is important.[1] There are a number of approaches to doing this. One class of approaches is related to ONION ROUTING, where messages encapsulated in layers of encryption are transmitted through a series of network nodes (onion routers), each of which "peels away" a single layer to reveal the data's next destination; Tor is an instance of this [Dingledine et al., 2004].[2]

Another kind of approach is that of the Crowds protocol [Reiter and Rubin, 1998]. Say a node in a network (crowd) wants to send a message to a designated recipient, and doesn't want it known that it is the sender. The idea in Crowds is that the node flips a biased coin: heads means it should forward the message to another randomly chosen forwarding node; tails means it should send it direct to the designated receiver. A node that a message has been forwarded to makes the same decision, whether it should in turn forward to another relay node or send it direct to the designated receiver. (Once a path is established between two nodes, this stays fixed.) In the end, neither the receiver nor any crowd member can know for certain who the initiator of the message was, as the initiator is indistinguishable from a forwarder. There are some mathematical and computational[3] proofs about the level of anonymity: for example, PROBABLE INNOCENCE — where an attacker, a CORRUPT NODE in the network (one that uses information obtained from forwarding the message to try to determine the initiator), is unable to have greater than 50% confidence that any specific node initiated the message — can be established for given numbers of corrupt nodes and values of the probability of forwarding.[4]

There is a recent variant of the Crowds protocol known as the Deterministic Crowds Protocol [Rass and Wigoutschnigg, 2016]. In this, instead of probabilistically forwarding a message, routing becomes deterministic but pseudorandom, in such a way that a (corrupt) node (or the receiver) cannot backtrack along the path taken by the message.

Assignment Aim Your assignment will be to implement a simplified version of some of the functionality of the Deterministic Crowds Protocol.

2 The Deterministic Crowds Protocol

2.1 Network Nodes

Following is a description adapted from Rass and Wigoutschnigg [2016]. For the purposes of sending a message, there are three kinds of nodes: the INITIATOR, intermediate FORWARDING NODES, and the RECEIVER. The N nodes of a network will be identified by integers $0 \ldots N-1$.

[1] Do you want everyone to know that you are the top-ranked author of *Twilight* fanfiction on Wattpad?

[2] `https://en.wikipedia.org/wiki/Tor_(anonymity_network)`

[3] There's been work on applying a MODEL CHECKER to Crowds to verify correctness properties: `https://www.prismmodelchecker.org/casestudies/crowds.php`.

[4] The protocol is, however, still vulnerable to attack by a local eavesdropper (an attacker that can observe all incoming and outgoing messages for any proper subset of the nodes) or an attack known as the predecessor attack.

Initiator We will call the initiating node v_0. There will be a function $f : \mathbb{N} \to \mathbb{N}$, taking a non-negative integer and producing non-negative integer, that a node will use to decide which node it will send a message to. (Notationally, f^{-1} will be the inverse of this function.)

To transmit a message m to a receiver v, v_0 first chooses a path length n and calls $v_n := v$ the receiver. It then goes on to choose a value $r_n \equiv v_n \bmod N$, and backward iterates $r_{n-1} = f^{-1}(r_n)$ until it reaches a value r_1 . The first node to transmit the payload to (where the payload consists of the message m, value r and function f) is thus $v_1 = r_1 \bmod N$, and the initiator acts just as if it would have if it had computed r_1 from the data being sent from elsewhere to v_0.

Question: If there is such a function f^{-1}, how can the protocol prevent *any* node from tracking the message back? The solution is for f to be a TRAPDOOR FUNCTION, one whose inverse can only (easily) be calculated by using a TRAPDOOR SECRET. §2.2 explains these and gives an example of how it is applied here.

Forwarding Node Forwarding node v_i, upon receiving the message m, along with a value r_i and a function f, first checks if it is the designated receiver. As it is not, it computes $r_{i+1} = f(r_i)$ and thence the next hop as $v_{i+1} = r_{i+1} \bmod N$, and sends the message m together with r_{i+1} and f to v_{i+1}.

Receiver Node v_n similarly checks if it is the designated receiver. As it is, the process stops.

2.2 Trapdoor Functions

An Example of a Trapdoor Function Trapdoor functions are used widely in cryptography. The kind of function we'll be using comes from RSA encryption.[5] There are three values that we'll be using to define our function f and its inverse f^{-1}; we'll call these e, d and K. e and K will be used to define f, which will be public; d will be the trapdoor secret.

Our public function will be

$$f(x) = x^e \bmod K \tag{1}$$

and our inverting function will be

$$g(x) = x^d \bmod K. \tag{2}$$

There is a process (in cryptography, a KEY GENERATION ALGORITHM) to find values of e, d, K such that $f(g(x)) = x$, i.e. $g(x) = f^{-1}(x)$.

Consider $e = 3, d = 7, K = 33$. Then, $f(30) = 30^3 \bmod 33 = 6$. To compute the inverse, $f^{-1}(6) = g(6) = 6^7 \bmod 33 = 30$. This inverse can only (easily) be computed if d is known.

Applying a Trapdoor to Deterministic Crowds Consider a network (crowd) with nodes $0 \ldots 19$. Let the message initiator be node $v_0 = 1$. v_0 decides on a path length of $n = 3$ (i.e. via two intermediate forwarding nodes) to send its message to receiver node $v = v_3 = 6$. It makes the following calculations:

1. $r_3 = 6$.
2. $r_2 = f^{-1}(6) = 6^7 \bmod 33 = 30$.

   This corresponds to node $v_2 = r_2 \bmod N = 30 \bmod 20 = 10$.
3. $r_1 = f^{-1}(30) = 30^7 \bmod 33 = 24$.

   This corresponds to node $v_1 = r_1 \bmod N = 24 \bmod 20 = 4$.

So the node that v_0 forwards the message to, along with the value $r_1 = 24$ and function f (but not f^{-1} or the value d that would allow f^{-1} to be calculated), is node 4.

The full path will be $1 \to 4 \to 10 \to 6$.

[5] An accessible explanation of the maths in RSA is available at `https://www.di-mgt.com.au/rsa_alg.html`. You don't need to know any of this to do the assignment, though.

3 Assignment Code Structure

You will be working with a Java project that has 4 classes:

- `NodeTransitionFunction`: This instantiates the functions $f(\cdot)$ and $g(\cdot)$ from Equations (1) and (2) respectively.
- `Node`: This represents a node in the network; it is where the core functionality of the assignment is. Nodes receive messages, determine the next one to forward them to, and carry out the forwarding, among other functions. A node will have an integer ID $0 \ldots N-1$, where N is the number of nodes a particular network.
- `Network`: This represents a network of nodes. The key component of the network is a lookup table that is accessible to all nodes, so that nodes can look up properties of other nodes. The table has the form

  ```
  Map<Integer,Node> lookup;
  ```

  It also contains functions that will be provided for reading network specifications and messages from files.
- `MessageTrackCheck`: This represents an encoded trail of which nodes were visited in the process of passing messages. (It is mostly just used in the JUnit tests.)

For your tasks, you'll be adding attributes and methods to existing classes given in the code bundle accompanying these specs. Where it's given, **you should use exactly the method stub provided** for implementing your tasks. **Don't change the names or the parameters or exception handling.** You can add more functions if you like.

3.1 Pass Level

To achieve at least a Pass ($\geq 50\%$) for the assignment, you should do all of the following.

There will be some sample input files to be used in the JUnit tests in the code bundle. The sample input `nodedef1.in` consists of 20 nodes, each with associated trapdoor function parameters $e = 3, d = 7, K = 33$. Obviously in a real network nodes would have different parameter values, and in later input files they will; it just happens that this triple is the smallest one that works for this kind of trapdoor function, and so the easiest to handle numerically when getting started.

1. There will be no public `Node` method to find out which node sent a message to the current node. (If there was, there would be no anonymity. However, we will change this slightly for corrupt nodes in the Distinction level tasks.) On the other hand, we want some way of recording which nodes were visited.[6]

   The `MessageTrackCheck` class acts a like a parity check,[7] and works as follows. A MessageTrackCheck instance t will sum the IDs of each node that is involved in passing a message, including the initiator and receiver; it will also add a specified offset. It will then internally calculate the sum mod 26, and make available the corresponding CHECK CHARACTER from the lower-case alphabet $a \ldots z$, assuming they are indexed by the values $0 \ldots 25$ respectively.

   So if a message track check has an offset of 3, and a message is passed along a path of nodes $1 \to 4 \to 10 \to 6$, the check character would be the character corresponding to $(3 + 1 + 4 + 10 + 6) \bmod 26 = 24$, which is y.

   Write the following methods for class `MessageTrackCheck`.

   ```
   public MessageTrackCheck(Integer offset) {
   // CONSTRUCTOR: Argument is offset to initialise the running sum
   }

   public void add(Integer n) {
   // PRE: -
   // POST: Adds n to the running sum
   }
   ```

[6] In particular, so I can automark this assignment.

[7] `https://en.wikipedia.org/wiki/Parity_bit`

```
public char check() {
// PRE: -
// POST: Returns the character that corresponds to the running sum mod 26;
//       0..25 correspond to a..z
}

public void reset(Integer offset) {
// PRE: -
// POST: Re-initialises the running sum to the given offset
}
```

2. Write the following methods for the `NodeTransitionFunction` class.

```
public NodeTransitionFunction(Integer exp, Integer KVal) {
// CONSTUCTOR: Sets the class to calculate f(x) = (x ^ exp) mod KVal
}

public Integer apply(Integer val) {
// PRE: -
// POST: Implements f(val)
}
```

   Note that if you implement `apply()` in a straightforward way, you will almost certainly exceed Java's `Integer.MAX_VALUE`: try it for yourself with an instance of `NodeTransitionFunction` instantiated with values 3 and 33, comparing it against the value obtained from e.g. a spreadsheet.

   Instead, there is a useful identity that lets you make the calculation without this risk:

$$(ab) \bmod p = ((a \bmod p)(b \bmod p)) \bmod p \tag{3}$$

   Implement `apply()` using this identity.[8]

3. Implement a constructor for class `Node` with the following signature:

```
public Node(Integer n, Integer e, Integer d, Integer K, Boolean encrypt, Boolean useBI,
            Map<Integer,Node> m, MessageTrackCheck t) {
// CONSTUCTOR:
//      n is node ID,
//      e is the exponent for the function f()
//      d is the exponent for the function g()
//      K is the divisor in f() and g()
//      encrypt is true if messages are encrypted, false otherwise
//      useBI is true if BigInteger should be used for NodeTransitionFunction, false otherwise
//      m is a non-null map of node IDs to node objects
//      t is an instance of MessageTrackCheck
}
```

   For the Pass level, `encrypt` and `useBI` will be set to false. The parameters `e`, `d` and `K` here correspond to the e, d, K from §2.2.

4. This task and those below all involve adding to class `Node`.

   For the Pass level, when a message is passed through the network it will have appended to the end a 3-character string that indicates the destination; we will call this combination of the original message and destination the AUGMENTED MESSAGE. A node will therefore know that it is the destination for a message from the final three characters of the augmented message. An augmented message that has the value `hello006` is thus destined for node 6.

   Implement the following method in class `Node`:

[8] *Hint*: There's some relevant code you can adapt at `https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/`. If you use this, cite the source in a comment.

```
public Boolean isDestinationNode(String msg) {
// PRE: msg is an augmented message (i.e. containing 3 characters at the end
//      indicating destination node)
// POST: Returns true if this is the destination node, false otherwise
//       E.g. For node 6, will return true for "hello006"
}
```

5. There are several getter methods for class `Node`:

```
public Integer getID() {
// PRE: -
// POST: Returns node ID
}

public Integer getE() {
// PRE: -
// POST: Returns value of e in this node's function f()
}

public Integer getK() {
// PRE: -
// POST: Returns value of K in this node's function f()
}

public Boolean transmittedMessage() {
// PRE: -
// POST: Returns true if this node has transmitted a message, false otherwise
}

public String getMsg() {
// PRE: -
// POST: Returns the current received (non-augmented) message, null if no received message
}
```

6. A node will have its own `NodeTransitionFunction` that corresponds to function $f(\cdot)$ as defined by its own parameters e, K.

```
public NodeTransitionFunction createForwardNodeTransitionFunction() {
// PRE: -
// POST: Creates a NodeTransitionFunction using this node's public function f()
//       with parameters e, K
}
```

7. All nodes have the core functionality of receiving and sending messages. For a node at step i in a path of length n, in sending a message it passes along the payload consisting of the (augmented) message, the value r_i, and the node transition function representing a particular $f(\cdot)$ that the next node will use. In our example from §2.2, for the node 1 at the first step in the path, it will be sending to node 4, the value of r is 24, and the node transition function is $f(\cdot)$ with $e = 3, K = 33$.

```
public void sendMsgToNode(Node n, String msg, Integer r, NodeTransitionFunction f) {
// PRE: n is a non-null node,
//      msg is an (augmented) message,
//      r is the current value of r from the forward transition function.
//      f is the forward transition function
// POST: invokes receiveMsgFromNode on node n
}
```

The following method specifies what happens when a node receives a message. Continuing the example, when node 4 receives the message from node 1, say the augmented message is `hello006`: node 4 is then not the destination. So it takes the value $r = 24$ and calculates the next node to forward the message to, $f(24)$.

```
public void receiveMsgFromNode(String msg, Integer id, Integer r, NodeTransitionFunction f) {
// PRE: msg is an augmented message,
//      id is the ID of the sending node,
//      r is the current value of r from the forward transition function,
//      f is the forward transition function
// POST: If this is the destination node, stop;
//       otherwise, send the message onwards.
//       Add ID of current (receiving) node to local MessageTrackCheck
}
```

8. An initiator has some additional functionality. First, there is a method to construct an augmented message from the original message:

```
public String addDestIDToMsg(String msg, Integer v) {
// PRE: msg is a message, v is a node ID
// POST: Returns a string that concatenates v as a 3-character string to the end of msg.
//       E.g. for msg="hello", v=6, returns "hello006"
}
```

Then, it has to calculate the value r_1 to determine which the first node is to send the message to, as described in §2.2.

```
public Integer firstRForInitiatingMessage(Integer k, Integer v) {
// PRE: v is destination node ID, k is number of steps
// POST: Uses the trapdoor function inverse, applied to destination node v with number of steps k,
//       to calculate the node path;
//       returns value of r that determines first step on node path
}
```

Then, it has to initiate the sending of the message; this involves creating the function $f(\cdot)$ using its own parameters e, K that can be passed along to later nodes in the path.

```
public void initiateMessage(String msg, Integer k, Integer v) {
// PRE: msg is an original message, v is destination node ID, k is number of steps
// POST: Adds destination ID to msg;
//       sends augmented msg to the next node, as determined by firstRForInitiatingMessage(k, v),
//       along with new forward transition function
}
```

3.2 Credit Level

To achieve at least a Credit ($\geq 65\%$) for the assignment, you should do the following. You should also have completed all the Pass-level tasks.

For Credit-level tasks and above, it will be up to you to discover how some more advanced Java features work. It will be OK to ask questions, but you'll be expected to read Java documentation yourself first.

In the Credit-level tasks, you'll be using Java's `BigInteger`[9] for calculations of $f(\cdot)$ and $g(\cdot)$. This class has been specifically designed for dealing with problems like the `Integer.MAX_VALUE` one noted in the Pass-level tasks.

1. In class `NodeTransitionFunction`, add a new method `apply()` that takes its argument as a `BigInteger` rather than an `Integer`.

```
public BigInteger apply(BigInteger val) {
// PRE: -
// POST: Implements f(val), with val as a BigInteger
}
```

You can, and should, still keep your earlier `apply()` method. From Java's point of view, the type signatures of these methods are different, and it will know which one to invoke by the argument.

[9] `https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html`

2. Add the following methods to class `Node` that duplicate several methods from the Pass-level tasks but that use `BigInteger` rather than `Integer`.

```
public void sendMsgToNode(Node n, String msg, BigInteger r, NodeTransitionFunction f) {
// PRE: n is a non-null node,
//      msg is an augmented message,
//      r is the current value of r from the forward transition function.
//      f is the forward transition function
// POST: invokes receiveMsgFromNode on node n
}

public void receiveMsgFromNode(String msg, Integer id, BigInteger r, NodeTransitionFunction f) {
// PRE: msg is an augmented message,
//      id is the ID of the sending node,
//      r is the current value of r from the forward transition function.
//      f is the forward transition function
// POST: If this is the destination node, stop;
//       otherwise, send the message onwards.
//       Add ID of current (receiving) node to local MessageTrackCheck
}

public BigInteger firstRForInitiatingMessage(Integer k, BigInteger v) {
// PRE: v is destination node ID, k is number of steps as a BigInteger
// POST: Uses the trapdoor function inverse, applied to destination node v with number of steps k,
//       to calculate the node path;
//       returns value of r that determines first step on node path
}
```

Note that you'll also want to change the internals of `initiateMessage()`, depending on whether `useBI` is set to true or false in the node constructor.

4 (High) Distinction Level

To achieve at least a Distinction ($75-100\%$) for the assignment, you should do the following. You should also have completed all the Credit-level tasks.

There are two aspects to these advanced level tasks.

Corrupt Nodes The first is implementing a simplified version of how a corrupt node might try to guess an initiator. For this, a corrupt node will assume that there is only one initiator in the network.[10] A corrupt node forwards and receives messages just like any other node, but keeps track of which node sent it a message. Further, a corrupt node v_{c_1} can find out from another corrupt node v_{c_2} which node v_p most recently sent v_{c_2} a message; that node v_p could likewise be corrupt, and v_{c_1} could then query v_p as well.

We will stipulate that a corrupt node v_{c_1} guesses another node v_g to be a message initiator if it discovers that v_g has transmitted messages that have reached v_{c_1} along *two different paths*. (v_{c_1} does not have to be the receiver; it may have just forwarded the message.)

Consider an example that extends from §2.2. We have the same network with nodes $0 \ldots 19$, and initiator node $v_0 = 1$. In this example, v_0 sends a first message with path length 2 to node $v_2 = 8$; the path will be $1 \to 2 \to 8$. Then it sends a second message with path length 4 to node $v'_4 = 7$; the path will be $1 \to 13 \to 19 \to 8 \to 7$.

Assume that nodes 2, 8, 13 and 19 are corrupt. Node 8 can then guess that node $v_0 = 1$ is the initiator. If the set of nodes that are corrupt were instead $\{2, 8, 19\}$, it could not be guessed.

[10] Imagine the scenario of a single whistleblower who is trying to anonymously send messages to others.

1. Write the following methods in class `Node`

```
public void setCorrupt() {
// PRE: -
// POST: Sets a node to be corrupt
}

public Integer lastSender() {
// PRE: -
// POST: If a node is not corrupt, returns -1;
//       if a node is corrupt, returns ID of node that last sent it a message,
//       -1 if it has not been sent any messages
}

public Integer guessInitiator() {
// PRE: -
// POST: Guesses a node to be the initiator if it can track back through corrupted nodes
//       along two separate paths;
//       returns this node ID, or -1 if no guess
}
```

Encryption In the second part of the advanced level, you'll be working with actual encrypted messages. Specifically, we will use ElGamal encryption[11] as specified in Rass and Wigoutschnigg [2016]. As with encryption in general, this involves a public key and a private key. When an initiator node v_0 wants to send a message to a receiver node v, it will use v's public key to encrypt the message; then only v can decrypt it, using its private key.

Determining whether a node is the destination or not is thus different from the earlier levels: there is no longer an augmented message of type `String` where the last 3 characters give the destination. Instead, we will implement it as described in Rass and Wigoutschnigg [2016]. First, the initiator uses a hash function on the original message, and then concatenates this hash value (which for us will be 3 characters long) to form an augmented message; it then encrypts this augmented message using the receiver's public key. It sends this encrypted message as usual.

Each node along the path to the receiver tries to decrypt the encrypted message with its own private key. Treating the decryption as a `String`, it hashes the first $n-3$ characters and compares it to the last 3 characters of the decryption. If it is the correct private key, these will match; if it is not, the decryption will look like gobbledygook rather than a valid string, and the hashes will not match.

There are a number of sample programs you can find on the Internet which use the same mechanisms we will use.[12] The cryptography libraries are part of `javax.crypto.Cipher`; to import this, you'll have to add some external libraries (made available in iLearn) to the project.[13]

You then need to implement the following methods in class `Node`

2. The following are some getter methods.

```
public Boolean hasMsgEncryption() {
// PRE: -
// POST: Returns true if messages are encrypted, false otherwise
}

public Key getPublicKey() {
// PRE:
// POST: Returns the node's public key (null if hasMsgEncryption() is false)
}
```

[11] `https://en.wikipedia.org/wiki/ElGamal_encryption`

[12] I suggest looking at `http://www.java2s.com/Tutorial/Java/0490__Security/ElGamalexamplewithrandomkeygeneration.htm`.

[13] Do this by right-clicking on your project in Eclipse and selecting `Properties -- Java Build Path -- Libraries -- Add External JARs`.

3. The following methods are related to the new definition of augmented messages.

```
public String basicHashFunction (String m) {
// PRE: -
// POST: Sums the numeric value of each character using Character.getNumericValue(),
//       takes mod 100 of the total; returns as a 3-char string
}
```

An an example, the basic hash function when applied to string `hello` should return the string `097`.

```
public String addCheckToMsg(String msg) {
// PRE: msg is a message
// POST: Returns a string that concatenates the basicHashFunction of msg
//       E.g. for msg="hello", returns "hello097"
}
```

4. The following methods are the heart of the message encryption process.

```
public byte[] encryptedMsg(String msg, Key chosenPubKey) {
// PRE: msg is a message, chosenPubKey is a public key
// POST: Returns msg encrypted with chosenPubKey
//       (null if hasMsgEncryption() is false or chosenPubKey is null)
}
```

```
public byte[] decryptedMsg(byte[] msg) {
// PRE: msg is an encrypted message as a byte array
// POST: Returns msg decrypted using node's private key
//       (null if hasMsgEncryption() is false)
}
```

```
public Boolean isDestinationNode(byte[] msg) {
// PRE: msg is an augmented encrypted message (i.e. containing 3 check digits at the end)
// POST: Returns true if this is the destination node, false otherwise.
//       Determines if this is the destination by decrypting msg,
//       then comparing the hashed decrypted core msg (i.e. up to the last 3 characters)
//       against the last 3 chars of the decrypted msg
}
```

Note that to fully implement encrypted messages, you'd also have to define new functions `sendMsgToNode()` and `recMsgFromNode()` so that their message arguments were of type `byte[]` rather than `String`, and change the internals of several other functions. However, you don't have to do this for the assignment. Only the above functions will be tested in the JUnit tests.

5 What To Hand In

In the submission page on iLearn for this assignment you must include the following:

Submit a zip file consisting of all the Java classes in the package from the original assignment code bundle.

Instructions on how to create the zipfile are available in iLearn: you'll find them with all the assignment 2 material.

Your file must leave unchanged the specification of already implemented functions, and include your implementations of your selection of method stubs outlined above.

Do not change the names of the method stubs because the auto-tester assumes the names given. Do not change the package statement. You may however include additional auxiliary methods if you need them.

Please note that we are unable to check individual submissions and so it is very important to abide by the above submission instructions.

6 Changelog

- **2/5/19**: Draft released (ver 1.0).
- **11/5/19**: Corrected postcondition on `receiveMsgFromNode()` to specify adding *current (receiving) node ID* to MessageTrackCheck, in specs and in code bundle (ver 1.01).

References

R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320, 2004.

Stefan Rass and Raphael Wigoutschnigg. Arguable Anonymity from Key-Privacy: The Deterministic Crowds Protocol. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS 2016)*, pages 571–576, 2016.

Michael Reiter and Aviel Rubin. Crowds: Anonymity for Web Transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1(1):66–92, 1998.