CMPUT 274 - Tangible Computing **Fall 2019**
Assignment #2

Encrypted Arduino Communication

Overview

You will be implementing a simple chat program between two Arduinos. Roughly speaking, two people will type into separate serial monitors. The message typed into the serial monitor will be sent to one Arduino, encrypted, sent to the other Arduino using using the serial port pins, decrypted, and then displayed on the other person's serial monitor. The encryption protocol we will use is RSA.

The purpose of the assignment is to develop your understanding of the following concepts:

1. Modular arithmetic
2. Bit operations
3. The notion of a time-efficient algorithm
4. Using random numbers
5. Byte-based serial communication between programs
6. ASCII character encodings
7. Reading character input and converting it into numbers

The assignment is structured into two parts with two separate deadlines. Each part is worth 50% of the total mark for the assignment.

You must work with a partner on this assignment using the "Full collaboration model". As usual, cite all sources that substantially contributed to your assignment.

For this assignment we will again be developing a considerable amount of the code as we work through it in class. **If you do not understand this assignment upon first reading (especially if it has just been released), do not be concerned**. Things should be more clear after the concepts are introduced in the lectures.

A working solution will be demonstrated in the lectures.

Part 1: Simple Proof of Concept

Due Monday, Nov 18.

In this part, a simpler version of the solution will be implemented. Even within this part, do not forget to test the individual components of the assignment one by one.

In the lectures, we will develop code that has one person type into the serial monitor, and their connected Arduino will then pass the entered byte to another Arduino using a serial port between Arduinos. The other Arduino will display the message on the screen.

You will build on this by having both Arduinos be able to chat with each other. Further, the bytes sent between Arduinos will be encrypted using the RSA encryption algorithm.

For Part 1, both Arduinos will uses the following `uint32_t` variables:

- serverPublicKey = 7
- serverPrivateKey = 27103
- serverModulus = 95477
- clientPublicKey = 11
- clientPrivateKey = 38291
- clientModulus = 84823

If you want, you can verify that both modulus variables are the product of two primes p, q and that multiplying the corresponding public/private key values for that modulus is 1 modulo $(p-1)\cdot(q-1)$ (i.e. are valid numbers for the RSA encryption scheme).

Since we have two sets of variables, each Arduino needs to know upon startup which set is its own. For this, one of the Arduinos will be configured using the digital port 13 to act as a *server*, the other will be configured to act as a *client*. In particular, connect digital port 13 to either +5V through a 550 Ω (or similar) resistor, or the ground.

When your program starts up, it reads from PIN 13 to see if it is acting as a server or client. In particular, if it reads `HIGH` from this port (i.e., the port is connected to +5V) it should act as a server, while if it reads `LOW` from the port (i.e., the port is connected to the ground) it should act as a client.

Breakdown of tasks.

- During `setup()`:
  1. Determine whether the Arduino is acting as a server or as a client.
  2. From this information, determine 4 important `uint32_t` variables that will be used to encrypt and decrypt messages:
     - d, the Arduino's private key (serverPrivateKey if acting as server, clientPrivateKey if acting as client)

- n, the Arduino's modulus (serverModulus if acting as server, clientModulus if acting as client)
- e, the other Arduino's public key (clientPublickey if acting as server, serverPublicKey if acting as client)
- m, the other Arduino's modulus (clientModulus if acting as server, serverModulus if acting as client)

We note that these do not seem like great variable names (d, n, e, m), but it is allowed in this assignment because those are common variable names used in mathematics to describe the RSA scheme. You may change them if you want, but then use a descriptive name like `myPublicKey`, etc.

- After the setup is complete, repeatedly do the following in a loop.

1. **Read** a character (if available) from the serial monitor and **send it** as an integer to the other machine **encrypted** with the help of the other Arduino's public key and modulus (for details of the encryption to be used, see below). You should also print this character back to the serial monitor (unencrypted) as well so you can see what you are typing without turning the echo on.

   Note: If the Arduino reads the carriage return character '`\r`' from the serial monitor, it should send the carriage return character and immediately follow it with the line feed character '`\n`' to the other machine. Of course, both characters should be encrypted before they are sent. Also print the unencrypted character `\n` to the serial monitor in this case as well.
2. **Receive** an **encrypted integer** (if available) from the other machine, **decrypt** it using the private key and modulus and **send** it to the serial monitor.

The Encryption Scheme:
In order for Arduino B to send a character c to Arduino A, they must follow these steps:

1. Arduino B calculates the 32-bit unsigned integer $x = c^e \bmod m$. Here e is Arduino A's public key and m is Arduino A's modulus.
2. Arduino B sends x to Arduino A.
3. Arduino A calculates $y = x^d \bmod n$ and uses y as the decrypted byte. Here d is Arduino A's private key and n is Arduino A's modulus.

If these calculations are carried out correctly, y will be equal to the original byte c. That is, Arduino A would have successfully read and decrypted the byte sent by Arduino B.

To implement this encryption scheme, you will have to use a **fast modular exponentiation** algorithm that avoids overflow. To do this, we will require you to adapt the fast powmod function from the lectures to work for multiplication: this will be worked out on a worksheet we will issue.

That is, in this part and in part 2, every call to the fast powmod function will use a modulus that is at most 31 bits long (i.e. less than $2^{31} - 1$). We will have you implement a modular multiplication algorithm, described on a worksheet, to avoid using 64-bit numbers.

Note: The purpose of writing your own multiplication function is to have you realize that you can multiply two 31-bit numbers (modulo a 31-bit number) while using only 32-bit types. The function becomes almost trivial if you use 64-bit types. In principle, you could extend this idea to use 63-bit keys even though the Arduino does not have 128-bit integer types, but we will stick to 31-bit keys.

For full marks, you cannot use 64-bit integer types. If you are having a hard time completing the multiplication function this way, feel free to use 64-bit types with the understanding that you will lose marks in this category.

Comment:
With real encryption using RSA, multiple bytes would be packed together into a larger integer before that integer is encrypted. But the original primes themselves have to be big enough, and our primes are not. So this is not the most secure use of RSA because of the limited number of encrypted messages that can be seen (it could be broken by a "frequency attack"). What you are learning here is the core of the encryption scheme, other practical considerations are harder to support on an Arduino or do not contribute as much to the learning outcomes of this assignment.

Sending and Receiving Integers:
Due to the way characters are encrypted using RSA, they will turn into integers that no longer fit into one byte. Specifically, in our case, they will be turned into unsigned 32 bit integers. In order to send and receive these integers between Arduinos, you can use these functions:

```
/** Writes an uint32_t to Serial3, starting from the least-significant
 * and finishing with the most significant byte.
 */
void uint32_to_serial3(uint32_t num) {
    Serial3.write((char) (num >> 0));
    Serial3.write((char) (num >> 8));
    Serial3.write((char) (num >> 16));
    Serial3.write((char) (num >> 24));
}


/** Reads an uint32_t from Serial3, starting from the least-significant
 * and finishing with the most significant byte.
 */
uint32_t uint32_from_serial3() {
    uint32_t num = 0;
    num = num | ((uint32_t) Serial3.read()) << 0;
    num = num | ((uint32_t) Serial3.read()) << 8;
    num = num | ((uint32_t) Serial3.read()) << 16;
    num = num | ((uint32_t) Serial3.read()) << 24;
    return num;
}
```

Testing:

Part of our testing of your solution will be to have an Arduino running your solution chat with an Arduino running our solution. The same holds for the second part.

Part 2: Complete Implementation

Due Monday, Dec 2.

In the second part you will make three improvements. These will make the encrypted communication both more secure, and easier to use.

Random Key Generation:

In part 1, you were given six constants that allowed the Arduinos to communicate. In part 2, the Arduinos will each generate their own 3 variables and then transfer their public keys and moduli (keeping their private keys private) to allow for communication.
To calculate the required values, these steps must be followed:

1. Generate two random prime numbers p and q, the first lying in the range $[2^{14}, 2^{15})$ and the second lying in the range $[2^{15}, 2^{16})$. To generate a random prime in the range $[2^k, 2^{k+1})$, do the following:

   - First, generate a random k-bit number (see below) and add it to 2^k.
   - Then while the number is not prime, add 1 to it (and wrap around from $2^{k+1} - 1$ to 2^k if you increment too high).
     Alternatively, while the number is not prime, just generate another k-bit number and repeat until you generated a prime.
   - Primes are dense, you will hit one very soon.

   We will discuss how to test if a number is prime in the lectures. Feel free to tweak this algorithm to be a bit faster (eg. skip along the odd numbers only), so long as it finds the next prime (again, wrapping around the range if necessary) after the first number that is generated.

2. Calculate $n = pq$, which becomes this Arduino's modulus. Note n lies in the range $[2^{29}, 2^{31})$.

3. Calculate the totient $\phi(n) = (p-1)(q-1)$.

4. Choose an integer e such that $1 < e < \phi(n)$ and e is co-prime to ϕ. (Equivalently, $gcd(e, \phi(n)) = 1$). This value e becomes this Arduino's public key. Specifically, we want you to generate a random 15-bit number e and increment it until $\gcd(e, \phi(n)) = 1$ **or** continue generating random 15-bit numbers k until you reach one that has $\gcd(e, \phi(n)) = 1$ (either is acceptable).

5. Find an integer d such that $e \cdot d \equiv 1 \pmod{\phi(n)}$. Equivalently, compute d as a **modular inverse** of e. d becomes this Arduino's private key. You can find this value using **Euclid's Extended Algorithm**. This algorithm will be presented in the lectures.

Generating Random Numbers:

To generate random numbers, use analogRead with analog pin 1, while the pin is not connected to anything. This will cause the number read to fluctuate. However, not all bits out of the 10 bits of the number returned by `analogRead` fluctuate randomly. To be on the safe side, use only the least significant bit of the value returned by `analogRead`. Hence, to get, say, a 15 bit random number, you will need to call `analogRead` 15 times, while waiting a little (say, for 5ms) in between the calls to allow the voltage on the pin to fluctuate.

Test this function! A very common mistake is having it return only one bit random numbers. **Note that you should not call random()**! This function is predictable and is thus unsuitable for cryptographic purposes. All random numbers in this assignment should obtained using the method described above.

Automatic Key Exchange and Handshake

The exchange of public keys and modulis will be done automatically. In addition to displaying the public keys on the serial monitor, each Arduino will also exchange their public keys over their serial link. The tricky part of this will be doing a *handshake* between the two Arduinos, to make sure they are both ready to exchange keys and begin their encrypted exchange.

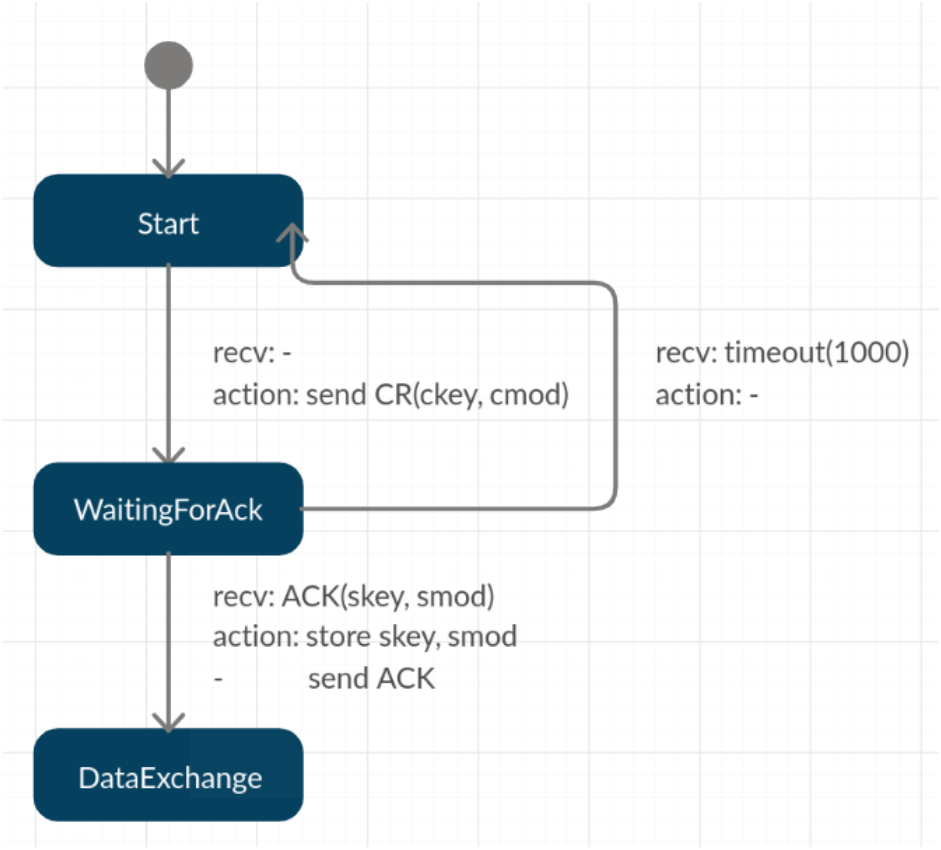
For this, one of the Arduinos will be configured using the digital port 13 to act as a *server*, the other will be configured to act as a *client* (in exactly the same way as Part 1).

The handshaking will work as follows:

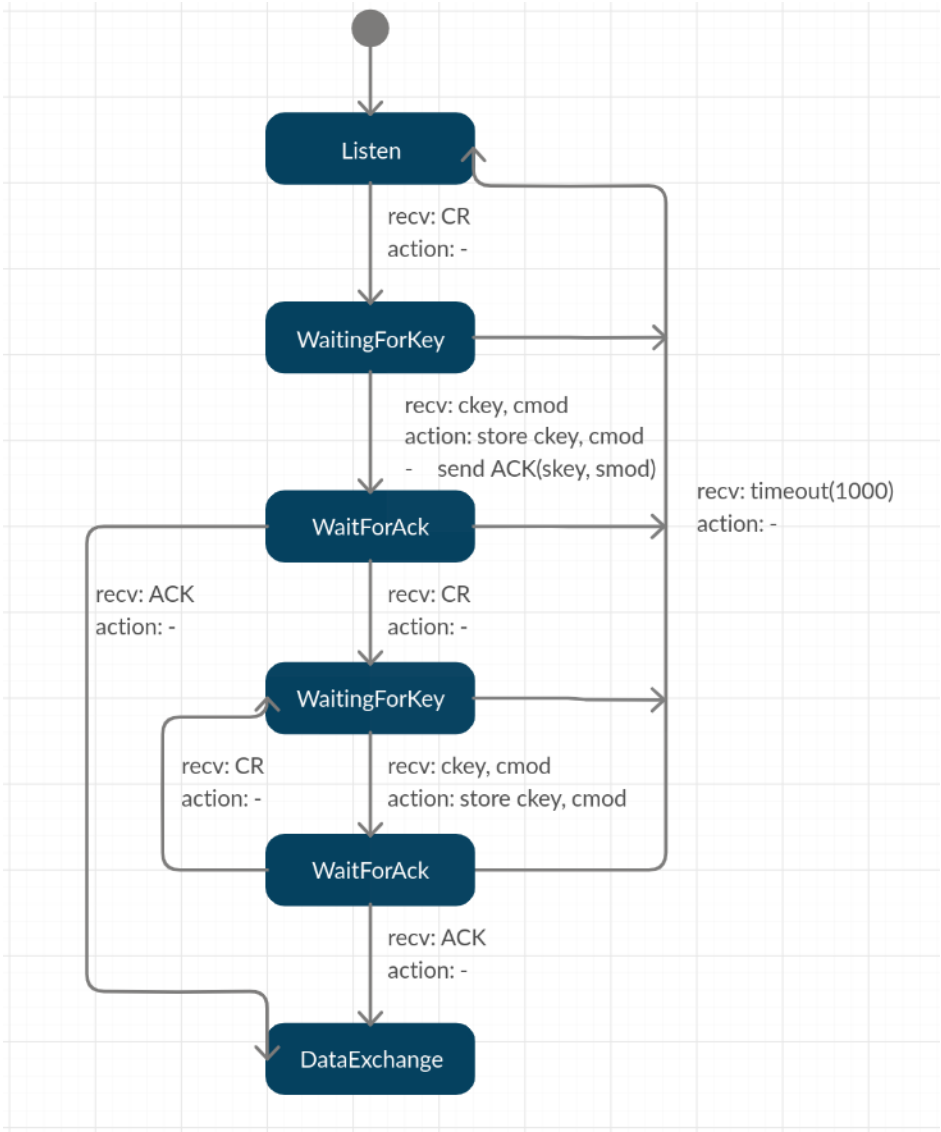
- The Arduino configured to act as a client will keep sending *connection requests* to the server. When the server captures one of these requests, it will *acknowledge* the receipt of the message so that the client knows that its partner is there and that it can move on to the data exchange phase.
- However, before moving to this phase, the client sends an *acknowledgement of the acknowledgement*. This is needed because the server may have received multiple connection requests that needs to be discarded before it moves to the data exchange phase, otherwise these requests will show up at the beginning of the data exchange as messages from the client.
- While waiting for the acknowledgement of the acknowledgement from the client, the server consumes all the outstanding connection requests so that when the acknowledgement arrives, it can move on to the data exchange phase, knowing that all the outstanding connection requests have been consumed.

More on the next page!

Here is the state-transition diagram of what the client should do:



Here is what the server should do:



Every arrow on the diagram shows a condition that triggers the transition after "recv" and the action that happens during the transition. The messages will be encoded as follows:

- `CR` is the single character 'C'.
- `ACK` is the single character 'A'.
- `ckey` is the 4 byte (32 bit) public key when the Arduino is acting as a client.
- `cmod` is the 4 byte (32 bit) modulus when the Arduino is acting as a client.
- `skey` is the 4 byte (32 bit) public key when the Arduino is acting as a server.
- `smod` is the 4 byte (32 bit) modulus when the Arduinos as acting as a server.

- The message `CR(ckey, cmod)` is 9 bytes long: 'C' followed by the 4 bytes of the public key followed by the 4 bytes of the modulus.
- The message `ACK(skey, smod)` is also 9 bytes long: 'A' followed by the 4 bytes of the public key followed by the 4 bytes of the modulus.

You should still use **serial port 3** for the communication between the Arduinos.

Note that while the Arduinos are waiting, they should consume all the characters until a sequence of characters is encountered which can be interpreted as a message that they were waiting for.

By design, by the time the Arduino moves to the `DataExchange` phase, it is guaranteed to get to the other Arduino's public key, so that it can compute the shared secret to be used during data exchange.

Do not forget to implement the timeout transitions. The numbers there are in milliseconds. You can use the `millis()` function to implement the timeouts.

One last detail is that when you read the letter 'C' in the `Listening` mode of the server, your code will assume that the 8 bytes, describing the public key and the modulus, will follow it. However, it may happen that the client did not succeed in sending these 8 bytes (because it was reset in the meanwhile) or that the letter 'C' was some leftover in the buffer. Thus, if the server cannot read 8 bytes in a short period of time, it should return to the Listening state (while it consumed all the bytes in the buffer).

The following functions will be essential to properly implement the protocol, together with the necessary timeouts (note that the bottom two functions are not changed from the ones used in part 1):

```
/** Waits for a certain number of bytes on Serial3 or timeout
 * @param nbytes: the number of bytes we want
 * @param timeout: timeout period (ms); specifying a negative number
 *                 turns off timeouts (the function waits indefinitely
 *                 if timeouts are turned off).
 * @return True if the required number of bytes have arrived.
 */
bool wait_on_serial3( uint8_t nbytes, long timeout ) {
  unsigned long deadline = millis() + timeout;//wraparound not a problem
  while (Serial3.available()<nbytes && (timeout<0 || millis()<deadline))
  {
    delay(1); // be nice, no busy loop
  }
  return Serial3.available()>=nbytes;
}

/** Writes an uint32_t to Serial3, starting from the least-significant
 * and finishing with the most significant byte.
 */
void uint32_to_serial3(uint32_t num) {
  Serial3.write((char) (num >> 0));
  Serial3.write((char) (num >> 8));
```

```
  Serial3.write((char) (num >> 16));
  Serial3.write((char) (num >> 24));
}

/** Reads an uint32_t from Serial3, starting from the least-significant
 * and finishing with the most significant byte.
 */
uint32_t uint32_from_serial3() {
  uint32_t num = 0;
  num = num | ((uint32_t) Serial3.read()) << 0;
  num = num | ((uint32_t) Serial3.read()) << 8;
  num = num | ((uint32_t) Serial3.read()) << 16;
  num = num | ((uint32_t) Serial3.read()) << 24;
  return num;
}
```

In particular, you MUST use the same ordering of the 4 bytes of the public keys and moduli as above when sending these keys, or your program will not be interoperable with the other programs, including the one used for marking. This is considered a violation of the protocol. The simplest solution of course is to use the above functions to send the public keys as this will ensure the correct ordering of bytes.

Testing:

As with part 1, if you are using the agreed handshake protocol and encryption scheme, you should be able to communicate properly with an Arduino running any other (correct) solution. This is indeed how your code will be tested, using an implementation that the instructors developed.

Submission Details (for both parts):

Compress all the following files in a compressed archive called `encrypted_communication_partX.tar.gz` or `encryped_communication_partX.zip` (where in both cases, `X` refers to the assignment part number (1 or 2)). Submit only this `.tar.gz` or `.zip`.

- All C++ files required to solve the assignment. You are permitted to use multiple files if you feel it helps, but it is not required (nor is it expected, as we have not yet discussed how to use multiple files in a single project). Just make sure to describe the role of each file in the `README`.

  The file with `main()` should be called `encrypted_communication_partX.cpp`, just like the compressed archive name (substituting 1 or 2 for `X`, as appropriate).

- The provided Arduino `Makefile`
- Your `README`, which specifies the **full names** of both group members.
- Submit the archive to the group on eClass you and your partner joined.

Make sure to follow the Code Submission Guidelines!